Using Python for Introductory Econometrics
1st edition

Florian Heiss

Daniel Brunner

Using Python for Introductory Econometrics

Companion website: **`http://www.UPfIE.net`**

Address:
Universitätsstraße 1, Geb. 24.31.01.24
40225 Düsseldorf, Germany

Contents

List of Tables

List of Figures

Preface

An essential part of learning econometrics is the application of the methods to real-world problems and data. The practical implementation and application of econometric methods and tools helps tremendously with understanding the concepts. But learning how to use a software package also has great benefits in and of itself. Nowadays, a vast majority of our students will have to deal with some sort of data analysis in their careers. So a solid understanding of some serious data analysis software is an invaluable asset for any student of economics, business administration, and related fields.

But what software package is the right one for learning econometrics? That's a tough question. Possibly the most important aspect is that it is widely used both in and outside of academia. A large and active user community helps the software to remain up to date and increases the chances that somebody else has already solved the problem at hand. And fluency in a software package is especially valuable on the job market as well as on the job if it is popular. Another aspect for the software choice is that it is easily (and ideally freely) available to all students.

Python is an ideal candidate for starting to learn econometrics and data analysis. It has a huge user base, especially in the fields of data science, machine learning, and artificial intelligence, where it arguably is the most popular software overall. These are very exciting areas and there is a lot of cutting edge research in the integration of their tools into the econometrics toolbox. So why not kill two birds with one stone and master a powerful and important software package while learning econometrics at the same time? Because Python must be hard to learn and to apply to econometrics? It is not at all, as this book shows.

And Python is completely free and available for all relevant operating systems. When using it in econometrics courses, students can easily download a copy to their own computers and use it at home (or their favorite cafés) to replicate examples and work on take-home assignments. This hands-on experience is essential for the understanding of the econometric models and methods. It also prepares students to conduct their own empirical analyses for their theses, research projects, and professional work.

A problem we encountered when teaching introductory econometrics classes is that the textbooks that also introduce Python do not discuss econometrics. Conversely, our favorite introductory econometrics textbooks do not cover Python. Although it is possible to combine a good econometrics textbook with an unrelated introduction to Python, this creates substantial hurdles because the topics and order of presentation are different, and the terminology and notation are inconsistent.

This book does not attempt to provide a self-contained discussion of econometric models and methods. Instead, it builds on the excellent and popular textbook "Introductory Econometrics" by Wooldridge (2019). It is compatible in terms of topics, organization, terminology, and notation, and is designed for a seamless transition from theory to practice.

The first chapter provides a gentle introduction to Python, covers some of the topics of basic statistics and probability presented in the appendix of Wooldridge (2019), and introduces Monte Carlo simulation as an additional tool. The other chapters have the same names and cover the same material as the respective chapters in Wooldridge (2019). Assuming the reader has worked through the material discussed there, this book explains and demonstrates how to implement everything in Python and replicates many textbook examples. We also open some black boxes of the built-in functions for estimation and inference by directly applying the formulas known from the textbook to reproduce the results. Some supplementary analyses provide additional intuition and insights. We want to thank Lars Grönberg providing us with many suggestions and valuable feedback about the contents of this book.

The book is designed mainly for students of introductory econometrics who ideally use Wooldridge (2019) as their main textbook. It can also be useful for readers who are familiar with econometrics and possibly other software packages. For them, it offers an introduction to Python and can be used to look up the implementation of standard econometric methods. Because we are explicitly building on Wooldridge (2019), it is useful to have a copy at hand while working through this book.

Note that there is a sister book *Using R for Introductory Econometrics*, just published as a second edition, see `http://www.URfIE.net`. We based this book on the R version, using the same structure, the same examples, and even much of the same text where it makes sense. This decision was not only made for laziness. It also helps readers to easily switch back and forth between the books. And if somebody worked through the R book, she can easily look up the pythonian way to achieve exactly the same results and vice versa, making it especially easy to learn both languages. Which one should you start with (given your professor hasn't made the decision for you)? Both share many of the advantages like having a huge and active user community, being widely used inside and outside of academia and being freely available. R is traditionally used in statistics, while Python is dominant in machine learning and artificial intelligence. These origins are still somewhat reflected in the availability of specialized extension packages. But most of all data analysis and econometrics tasks can be equally well performed in both packages. At the end, it's most important point is to get used to the workflow of *some* dedicated data analysis software package instead of not using any software or a spreadsheet program for data analysis.

All computer code used in this book can be downloaded to make it easier to replicate the results and tinker with the specifications. The companion website also provides the full text of this book for online viewing and additional material. It is located at:

`http://www.UPfIE.net`

1. Introduction

Learning to use *Python* is straightforward but not trivial. This chapter prepares us for implementing the actual econometric analyses discussed in the following chapters. First, we introduce the basics of the software system *Python* in Section 1.1. In order to build a solid foundation we can later rely on, Chapters 1.2 through 1.4 cover the most important concepts and approaches used in *Python* like working with objects, dealing with data, and generating graphs. Sections 1.5 through 1.7 quickly go over the most fundamental concepts in statistics and probability and show how they can be implemented in *Python*. More advanced *Python* topics like conditional execution, loops, functions and object orientation are presented in Section 1.8. They are not really necessary for most of the material in this book. An exception is Monte Carlo simulation which is introduced in Section 1.9.

1.1. Getting Started

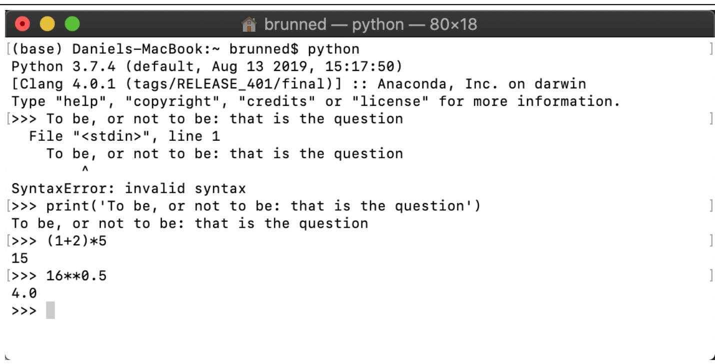
Before we can get going, we have to find and download the relevant software, figure out how the examples presented in this book can be easily replicated and tinkered with, and understand the most basic aspects of *Python*. That is what this section is all about.

1.1.1. Software

Python is a free and open source software. Its homepage is `https://www.python.org/`. There, a wealth of information is available as well as the software itself. We recommend installing the *Python* distribution Anaconda (also open source), which includes *Python* plus many tools needed for data analysis. For more information and installation files, see `https://www.anaconda.com`.

Distributions are available for Windows, Mac, and Linux systems and come in two versions. The examples in this book are based on the installation of the latest version, *Python* 3. It is not backwards compatible to *Python* 2.

Figure 1.1. *Python* in the Command Line



After downloading and installing, *Python* can be accessed by the command line interface. In Windows, run the program “Anaconda Prompt”. In Linux or macOS you can simply open a terminal window. You start *Python* by typing **`python`** and pressing the return key (↩). This will look similar to the screenshot in Figure 1.1. It provides some basic information on *Python* and the installed version. Right to the ">>>" sign is the prompt where the user can type commands for *Python* to evaluate.

We can type whatever we want here. After pressing ↩ , the line is terminated, *Python* tries to make sense out of what is written and gives an appropriate answer. In the example shown in Figure 1.1, this was done four times. The texts we typed are shown next to the ">>>" sign, *Python* answers under the respective line.

Our first attempt did not work out well: We have got an error message. Unfortunately, *Python* does not comprehend the language of Shakespeare. We will have to adjust and learn to speak *Python*'s less poetic language. The second command shows one way to do this. Here, we provide the input to the command **`print`** in the correct syntax, so *Python* understands that we entered text and knows what to do with it: print it out on the console. Next, we gave *Python* simple computational tasks and got the result under the respective command. The syntax should be easy to understand – apparently, *Python* can do simple addition and deals with the parentheses in the expected way. The meaning of the last command is less obvious, because it uses the pythonian way of calculating an exponential term: **`16**0.5`** = $16^{0.5} = \sqrt{16} = 4$.

Python is used by typing commands such as these. Not only Apple users may be less than impressed by the design of the user interface and the way the software is used. There are various approaches to make it more user friendly by providing a different user interface added on top of plain *Python*. Notable examples include IDLE, PyCharm, Visual Studio and Spyder. The latter was already set up during the installation of Anaconda and we use it for all what follows. The easiest way to start Spyder is by selecting it in the Anaconda Navigator that was also set up during the installation of Anaconda.

A screenshot of the user interface on a Mac computer is shown in Figure 1.2 (on other systems it will look very similar). There are several sub-windows. The one on the bottom right named "IPython console" looks very similar and behaves exactly the same as the command line. The usefulness of the other windows will become clear soon.

Here are a few quick tricks for working in the console of Spyder:

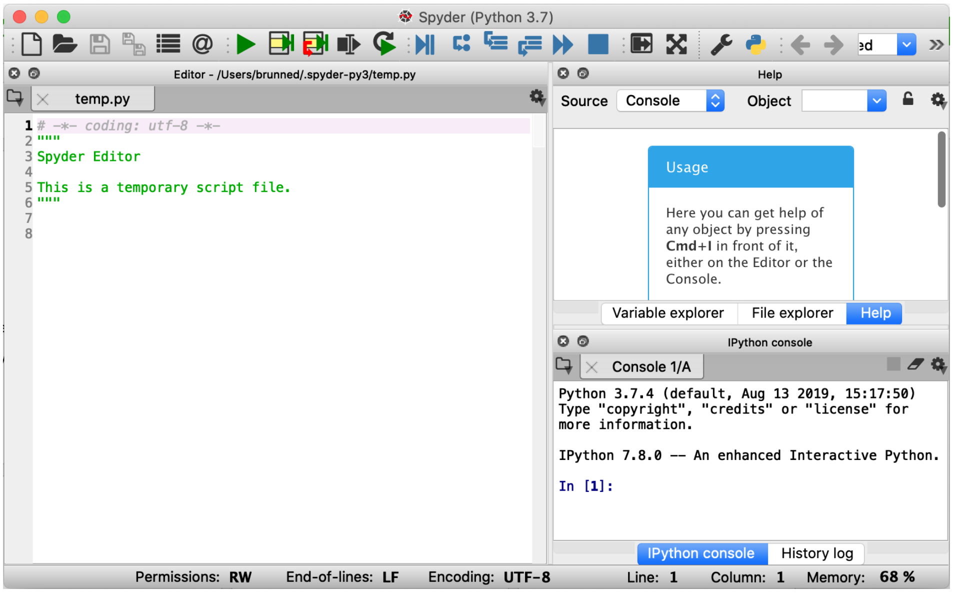
- When starting to type a command, press the tabulator key ⇆ to see a list of suggested commands. Typing **`pr`**, for example, followed by ⇆ gives a list of all *Python* commands starting with **`pr`**, like the **`print`** command.
- Use **`help(command)`** to print the help page for the provided command.
- With the ↑ and ↓ arrow keys, we can scroll through the previously entered commands to repeat or correct them.

1.1.2. *Python* Scripts

As already seen, we will have to get used to interacting with our software using written commands. While this may seem odd to readers who do not have any experience with similar software at this point, it is actually very common for econometrics software and there are good reasons for this. An important advantage is that we can easily collect all commands we need for a project in a text file called *Python* script.

A *Python* script contains all commands including those for reading the raw data, data manipulation, estimation, post-estimation analyses, and the creation of graphs and tables. In a complex project, these tasks can be divided into separate *Python* scripts. The point is that the script(s) together with the raw data generate the output used in the term paper, thesis, or research paper. We can then ask *Python* to evaluate all or some of the commands listed in the *Python* script at once.

Figure 1.2. Spyder User Interface



This is important since a key feature of the scientific method is reproducibility. Our thesis adviser as well as the referee in an academic peer review process or another researcher who wishes to build on our analyses must be able to fully understand where the results come from. This is easy if we can simply present our *Python* script which has all the answers.

Working with *Python* scripts is not only best practice from a scientific perspective, but also very convenient once we get used to it. In a nontrivial data analysis project, it is very hard to remember all the steps involved. If we manipulate the data for example by directly changing the numbers in a spreadsheet, we will never be able to keep track of everything we did. Each time we make a mistake (which is impossible to avoid), we can simply correct the command and let *Python* start from scratch by a simple mouse click if we are using scripts. And if there is a change in the raw data set, we can simply rerun everything and get the updated tables and figures instantly.

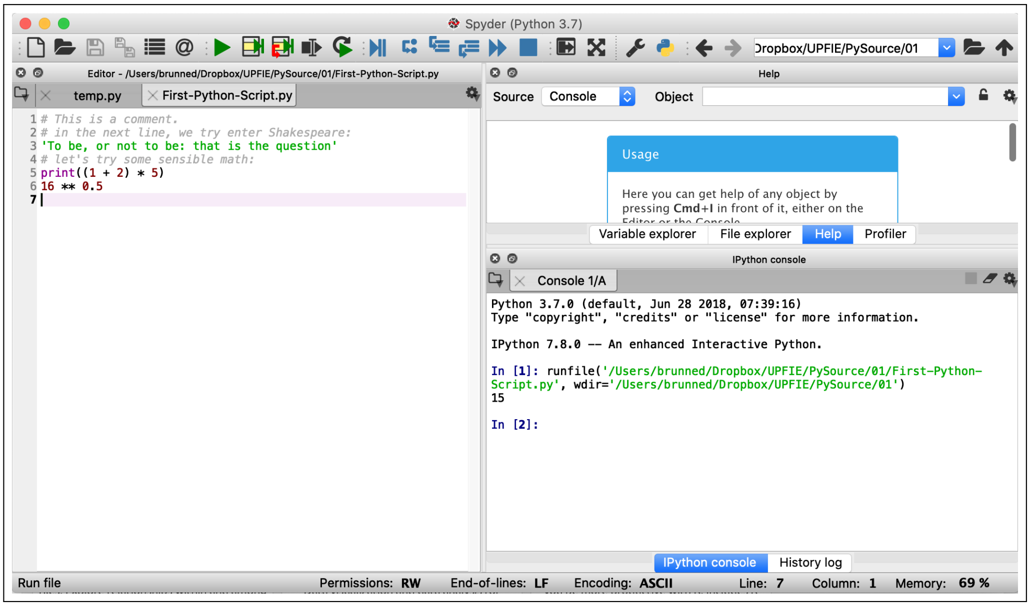
Using *Python* scripts is straightforward: We just write our commands into a text file and save it with a "`.py`" extension. When using a user interface like Spyder, working with scripts is especially convenient since it is equipped with a specialized editor for script files. To use the editor for working on a new *Python* script, use the menu `File`→`New file...`.

The window in the left part of Figure 1.2 is the script editor. We can type arbitrary text, begin a new line with the return key, and navigate using the mouse or the ↑ ↓ ← → arrow keys. Our goal is not to type arbitrary text but sensible *Python* commands. In the editor, we can also use tricks like code completion that work in the Console window as described above. A new command is generally started in a new line, but also a semicolon "`;`" can be used if we want to cram more than one command into one line – which is often not a good idea in terms of readability.

An extremely useful tool to make *Python* scripts more readable are comments. These are lines beginning with a "`#`". These lines are not evaluated by *Python* but can (and should) be used to structure the script and explain the steps. *Python* scripts can be saved and opened using the `File` menu.

Figures 1.3 and 1.4 show a screenshot of Spyder with a *Python* script saved as "First-Python-Script.py". It consists of six lines in total including three comments. We can send lines of code to

Figure 1.3. Executing a Script with ▶



Python to be evaluated in two different ways:

- Click ▶. The complete script is executed and only results that are explicitly printed out (by the command **print**) show up in the "IPython console" window. The example in Figure 1.3 therefore only returns **15**.
- Execute Python commands and scripts line by line or blockwise. The window "IPython console" shows the command you executed and the output. Press F9 to execute the line of the current cursor position or a highlighted block of code (with the mouse or by holding Shift ⇧ while navigating). Figure 1.4 demonstrates the execution line by line.

In what follows, we will do everything using *Python* scripts. All these scripts are available for download to make it easy and convenient to reproduce all contents in real time when reading this book. As already mentioned, the address is

`http://www.UPfIE.net`

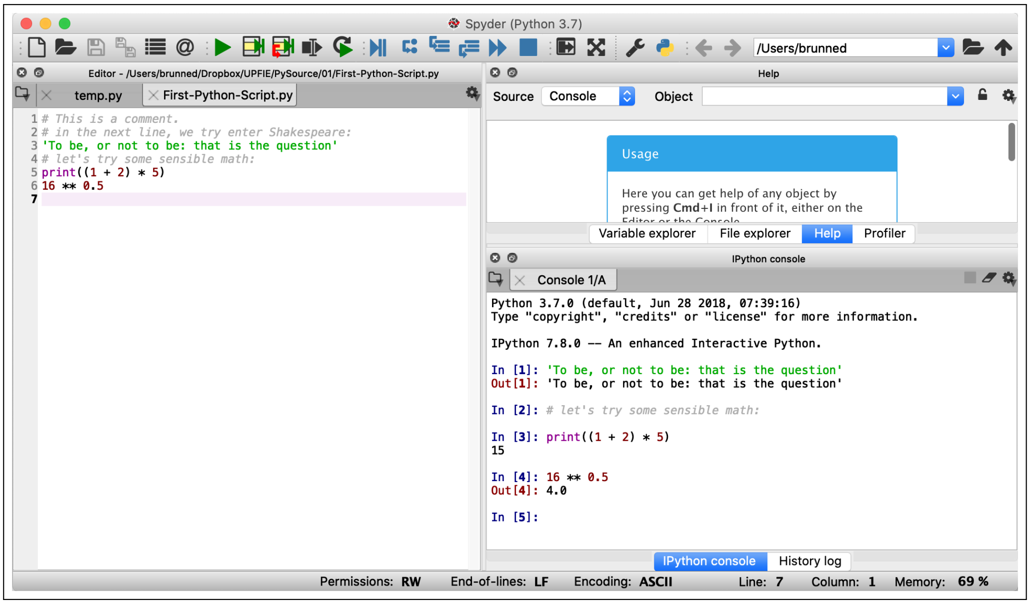
They are also printed in Appendix IV. In the text, we will not show screenshots, but the script files printed in **bold** and (if any) *Python*'s output in `standard font`. The latter only contains output that is explicitly printed out, just like the example in Figure 1.3. Script 1.1 (`First-Python-Script.py`) demonstrates the way we discuss *Python* code in this book:[1]

Script 1.1: First-Python-Script.py

```
# This is a comment.
# in the next line, we try to enter Shakespeare:
'To be, or not to be: that is the question'
# let's try some sensible math:
print((1 + 2) * 5)
16 ** 0.5
print('\n')
```

[1] To improve the readability of generated output, we will often use print commands including **\n** to start a new line.

Figure 1.4. Executing a Script Line by Line



Output of Script 1.1: `First-Python-Script.py`

```
15
```

Script 1.2 (`Python-as-a-Calculator.py`) is a second (and more representative) example in which *Python* is used for simple tasks any basic calculator can do. The *Python* script and output are:

Script 1.2: `Python-as-a-Calculator.py`

```python
result1 = 1 + 1
print(f'result1: {result1}\n')

result2 = 5 * (4 - 1) ** 2
print(f'result2: {result2}\n')

result3 = [result1, result2]
print(f'result3: \n{result3}\n')
```

Output of Script 1.2: `Python-as-a-Calculator.py`

```
result1: 2

result2: 45

result3:
[2, 45]
```

By using the function **`print(f'some text {variablename}')`** we can combine text we want to print out in combination with values of certain variables. This gives clear and readable output. We will discuss some additional hints for efficiently working with *Python* scripts in Section 19.

1.1.3. Modules

Modules are *Python* files that contain functions and variables. You can access these modules and make use of their code to solve your problem.

The standard distribution of *Python* already comes with a number of built-in modules. To make use of their commands you have to import these modules first. Script 1.3 (`Module-Math.py`) demonstrates this with the **`math`** module. All content of this module becomes available under the module name, or, as in this case, an alias object we labeled **`someAlias`**.[2] You can choose whatever name you want, but usually these aliases follow a naming convention. After the import, functions and variables are accessed by the dot (`.`) syntax, which is related to the concept of object orientation described in Section 1.8.4.

Script 1.3: `Module-Math.py`

```
import math as someAlias

result1 = someAlias.sqrt(16)
print(f'result1: {result1}\n')

result2 = someAlias.pi
print(f'Pi: {result2}\n')

result3 = someAlias.e
print(f'Eulers number: {result3}\n')
```

Output of Script 1.3: `Module-Math.py`

```
result1: 4.0

Pi: 3.141592653589793

Eulers number: 2.718281828459045
```

The functionality of *Python* can also be extended relatively easily by advanced users. This is not only useful to those who are able and willing to do this, but also for a novice user who can easily make use of a wealth of extensions generated by a big and active community. Since these extensions are mostly programmed in *Python*, everybody can check and improve the code submitted by a user, so the quality control works very well. The Anaconda distribution of *Python* already comes with a number of external modules, also called packages, that we need for data analyses.

On top of the packages that come with the standard installation or Anaconda, there are countless packages available for download. If they meet certain quality criteria, they can be published on the official "Python Package Index" (PyPI) servers at `https://pypi.org/`. Downloading and installing these packages is simple: Run your command line as explained in Section 1.1.1 and type

```
pip install modulename
```

There are thousands of packages provided at the PyPI. Here is a list of those we will use throughout this book with their official description:

- **`wooldridge`**: "Data sets from Introductory Econometrics: A Modern Approach (6th ed, J.M. Wooldridge)."
- **`numpy`**: "NumPy is the fundamental package for array computing with Python."

[2] You can also directly use objects from modules without referencing the modul name or its alias by using the command **`from`**. We will not use this way of importing, but sometimes it might be more convenient.

- **pandas**: "Powerful data structures for data analysis, time series, and statistics."
- **pandas_datareader**: "Data readers extracted from the pandas codebase, should be compatible with recent pandas versions."
- **statsmodels**: "Statistical computations and models for Python."
- **matplotlib**: "Python plotting package."
- **scipy**: "SciPy: Scientific Library for Python."
- **patsy**: "A Python package for describing statistical models and for building design matrices."
- **linearmodels**: "Instrumental Variable and Linear Panel models for Python."

Of course, the installation only has to be done once per computer/user and needs an active internet connection.

1.1.4. File Names and the Working Directory

There are several possibilities for *Python* to interact with files. The most important ones are to import or export a data file. We might also want to save a generated figure as a graphics file or store regression tables as text, spreadsheet, or LaTeX files.

Whenever we provide *Python* with a file name, it can include the full path on the computer. The full (i.e. "absolute") path to a script file might be something like

```
/Users/MyPyProject/MyScript.py
```

on a Mac or Linux system. The path is provided for Unix based operating systems using forward slashes. If you are a Windows user, you usually use back slashes instead of forward slashes, but the Unix-style will also work in *Python*. On a Windows system, a valid path would be

```
C:/Users/MyUserName/Documents/MyPyProject/MyScript.py
```

If we do not provide any path, *Python* will use the current "working directory" for reading or writing files. After importing the module **os**, it can be obtained by the command **os.getcwd()**. To change the working directory, use the command **os.chdir(path)**. Relative paths, are interpreted relative to the current working directory. For a neat file organization, best practice is to generate a directory for each project (say `MyPyProject`) with several sub-directories (say `PyScripts`, `data`, and `figures`). At the beginning of our script, we can use **os.chdir('/Users/MyPyProject')** and afterwards refer to a data set in the respective sub-directory as `data/MyData.csv` and to a graphics file as `figures/MyFigure.png`.[3]

1.1.5. Errors and Warnings

Something you will experience very soon when starting to work with *Python* (or any other similar software package) is that you will make mistakes. The main difference to learning to ride a bicycle is that when learning to use *Python*, mistakes will not hurt. Another difference is that even people who have been using *Python* for years make mistakes all the time.

Many mistakes will cause *Python* to complain in the form of error messages or warnings. An important part of learning *Python* is to roughly get an idea of what went wrong from these messages. Here is a list of frequent error messages and warnings you might get:

[3]For working with data sets, see Section 1.3.

- **`NameError: name 'x' is not defined`**: We have tried to use a variable **`x`** that isn't defined (yet). Could also be due to a typo in the variable name.
- **`FileNotFoundError: [Errno 2] No such file or directory: 'data.csv'`**: *Python* wasn't able to open the file. Check the working directory, path, file name.
- **`ModuleNotFoundError: No module named 'xyz'`**: We mistyped the module name. Or the required module is not installed on the computer. In this case, install it as described in Section 1.1.3.

There are countless other error messages and warnings you may encounter. Some of them are easy to interpret, but others might require more investigative prowess. Often, the search engine of your choice will be helpful.

1.1.6. Other Resources

There are many useful resources helping to learn and use *Python*. Useful books on *Python* in general include Downey (2015), Matthes (2015), Barry (2016) and many others. Oliphant (2007) introduces *Python* for scientific computing and Guido and Mueller (2016) narrow it down to data science.

Since *Python* has a very active user community, there is also a wealth of information available for free on the internet. Here are some suggestions:

- The official Python Tutorial
  `https://docs.python.org/3/tutorial/index.html`
- Additional links to external resources like tutorials and books
  `https://wiki.python.org/moin/BeginnersGuide`
- The links to module documentations available at the Python Package Index
  `https://pypi.org`
- Quantitative economic modeling with *Python*
  `https://python.quantecon.org/`
- Stack Overflow: A general discussion forum for programmers, including many *Python* users
  `https://stackoverflow.com`
- Cross Validated: Discussion forum on statistics and data analysis with an active *Python* community
  `https://stats.stackexchange.com`

1.2. Objects in *Python*

Python can work with numbers, lists, arrays, texts, data sets, graphs, functions, and many more objects of different types. This section covers the most important ones we will frequently encounter in the remainder of this book. We will first introduce built-in objects that are available with the standard distribution of *Python*. In the second part we cover objects included in the modules **numpy** and **pandas**.

1.2.1. Variables

We have already observed *Python* doing some basic arithmetic calculations. From Script 1.2 (`Python-as-a-Calculator.py`), the general approach of *Python* should be self-explanatory. Fundamental operators include **+**, **−**, *****, **/** for the respective arithmetic operations and parentheses **(** and **)** that work as expected.

We will often want to store results of calculations to reuse them later. For this, we can assign any result to a variable. A variable has a name and by this name you can access the assigned object. We can freely choose the variable name given certain rules – they have to start with a (small or capital) letter and include only letters, numbers, and the underscore character "_". *Python* is case sensitive, so **x** and **X** are different variables.

You already saw how variables are used to reference objects in Script 1.2 (`Python-as-a-Calculator.py`): The content of an object is assigned using **=**. In order to assign the result of **1 + 1** to the variable **result1**, type **result1 = 1 + 1** .
A new object is created, which includes the value 2. After assigning it to **result1**, we can use **result1** in our calculations. If there was a variable with this name before, its content is overwritten.

A list of all currently defined variable names is shown in the "Variable explorer" window in Spyder, see Figure 1.3 (top right by default). You can also use the command **dir** to do this. Removing a previously defined variable (for example **x**) from the workspace is done using **del x**.

Up to now, we assigned results of arithmetic operations to variables. In the next sections, we will introduce more complex types of objects like texts, arrays, lists, data sets, function definitions, and estimation results.

1.2.2. Objects in *Python*

You might wonder what kind of objects we have dealt with so far. Script 1.4 (`Objects-in-Python.py`) shows how to figure this out by using the command **type**:

Script 1.4: `Objects-in-Python.py`

```
result1 = 1 + 1
# determine the type:
type_result1 = type(result1)
# print the result:
print(f'type_result1: {type_result1}')

result2 = 2.5
type_result2 = type(result2)
print(f'type_result2: {type_result2}')

result3 = 'To be, or not to be: that is the question'
type_result3 = type(result3)
print(f'type_result3: {type_result3}\n')
```

Table 1.1. Logical Operators

x==y	**x** is equal to **y**	**x!=y**	**x** is NOT equal to **y**
x<y	**x** is less than **y**	**not b**	NOT **b** (i.e. **True**, if **b** is **False**)
x<=y	**x** is less than or equal to **y**	**a or b**	Either **a** or **b** is **True** (or both)
x>y	**x** is greater than **y**	**a and b**	Both **a** and **b** are **True**
x>=y	**x** is greater than or equal to **y**		

Output of Script 1.4: `Objects-in-Python.py`

```
type_result1: <class 'int'>
type_result2: <class 'float'>
type_result3: <class 'str'>
```

The command **type** tells us that we have created integers (**int**), floating point numbers (**float**) and text objects (**str**). The data type not only defines what values can be stored, but also the actions you can perform on these objects. For example, if you want to add an integer to **result3**, *Python* will return:

```
TypeError: can only concatenate str (not "int") to str
```

Scalar data types like **int**, **float** or **str** contain only one single value. A Boolean value, also called logical value, is another scalar data type that will become useful if you want to execute code only if one or more conditions are met. An object of type **bool** can only take one of two values: **True** or **False**. The easiest way to generate them is to state claims which are either true or false and let *Python* decide. Table 1.1 lists the main logical operators.

As we saw in previous examples, scalar types differ in what kind of data they can be used for:

- **int**: whole numbers, for example **2** or **5**
- **float**: numbers with a decimal point, for example **2.0** or **4.95**
- **str**: any sequence of characters delimited by either single or double quotes, for example **'ab'** or **"abc"**
- **bool**: either **True** or **False**

For statistical calculations, we obviously need to work with data sets including many numbers or texts instead of scalars. The simplest way we can collect components (even components of different types) is called a **list** in *Python* terminology. To define a **list**, we can collect different values using **[value1,value2,...]**. You can access a **list** entry by providing the position (starting at 0) within square brackets next to the variable name referencing the list (see Script 1.6 (`Lists.py`) for an example). You can also access a range of values by using their start position *i* and end position *j* with the syntax **listname[i:(j+1)]**.

There are two types of actions you can do with lists (or other objects): apply a function or a method. We will go into details in Section 1.8.4, and here just demonstrate the different syntax of function and method calls. The examples in Script 1.6 (`Lists.py`) should help to understand the concept and use of a **list**. Script 1.5 (`Lists-Copy.py`) in the appendix demonstrates how to work with a copy of a list. By default you will not work on a copy when assigning it to another variable, but the underlying object. For a **list**, use **[:]** to create a copy.

Script 1.6: `Lists.py`

```
# define a list:
example_list = [1, 5, 41.3, 2.0]
print(f'type(example_list): {type(example_list)}\n')

# access first entry by index:
first_entry = example_list[0]
print(f'first_entry: {first_entry}\n')

# access second to fourth entry by index:
range2to4 = example_list[1:4]
print(f'range2to4: {range2to4}\n')

# replace third entry by new value:
example_list[2] = 3
print(f'example_list: {example_list}\n')

# apply a function:
function_output = min(example_list)
print(f'function_output: {function_output}\n')

# apply a method:
example_list.sort()
print(f'example_list: {example_list}\n')

# delete third element of sorted list:
del example_list[2]
print(f'example_list: {example_list}\n')
```

Output of Script 1.6: `Lists.py`

```
type(example_list): <class 'list'>

first_entry: 1

range2to4: [5, 41.3, 2.0]

example_list: [1, 5, 3, 2.0]

function_output: 1

example_list: [1, 2.0, 3, 5]

example_list: [1, 2.0, 5]
```

A key characteristic of a **`list`** is the order of included components. This order allows you to access its components by a position. Dictionaries (**`dict`**) are unordered sets of components. You access components by their unique keys. Script 1.8 (`Dicts.py`) demonstrates their definition and some basic operations. Working on a copy is demonstrated in the appendix (Script 1.7 (`Dicts-Copy.py`)).

Script 1.8: Dicts.py

```
# define and print a dict:
var1 = ['Florian', 'Daniel']
var2 = [96, 49]
var3 = [True, False]
example_dict = dict(name=var1, points=var2, passed=var3)
print(f'example_dict: \n{example_dict}\n')

# another way to define the dict:
example_dict2 = {'name': var1, 'points': var2, 'passed': var3}
print(f'example_dict2: \n{example_dict2}\n')

# get data type:
print(f'type(example_dict): {type(example_dict)}\n')

# access 'points':
points_all = example_dict['points']
print(f'points_all: {points_all}\n')

# access 'points' of Daniel:
points_daniel = example_dict['points'][1]
print(f'points_daniel: {points_daniel}\n')

# add 4 to 'points' of Daniel and let him pass:
example_dict['points'][1] = example_dict['points'][1] + 4
example_dict['passed'][1] = True
print(f'example_dict: \n{example_dict}\n')

# add a new variable 'grade':
example_dict['grade'] = [1.3, 4.0]

# delete variable 'points':
del example_dict['points']
print(f'example_dict: \n{example_dict}\n')
```

Output of Script 1.8: Dicts.py

```
example_dict:
{'name': ['Florian', 'Daniel'], 'points': [96, 49], 'passed': [True, False]}

example_dict2:
{'name': ['Florian', 'Daniel'], 'points': [96, 49], 'passed': [True, False]}

type(example_dict): <class 'dict'>

points_all: [96, 49]

points_daniel: 49

example_dict:
{'name': ['Florian', 'Daniel'], 'points': [96, 53], 'passed': [True, True]}

example_dict:
{'name': ['Florian', 'Daniel'], 'passed': [True, True], 'grade': [1.3, 4.0]}
```

There are many more important data types and we covered only the ones relevant for this book. Table 1.2 summarizes these built-in data types plus a simple example in case you have to look them up later.

Table 1.2. *Python* Built-in Data Types

Python `type`	Data Type	Example
`int`	Integer	`a = 5`
`float`	Floating Point Number	`a = 5.3`
`str`	String	`a = 'abc'`
`bool`	Boolean	`a = True`
`list`	List	`a = [1, 3, 1.5]`
`dict`	Dict	`a = {'b':[1,2], 'c':[5,3]}`

1.2.3. Objects in `numpy`

Before you start working with **numpy**, make sure that you have the Anaconda distribution or install **numpy** as explained in Section 1.1.3. For more information about the module, see Walt, Colbert, and Varoquaux (2011). It is standard to import the module under the alias **np** when working with **numpy**, so the first line of code always is:

```
import numpy as np
```

The most important data type in **numpy** is the multidimensional array (**ndarry**). We will first introduce the definition of this data type as well as the basics of accessing and manipulating arrays. Second, we will demonstrate functions and methods that become useful when working on econometric problems.

To create a simple array, provide a **list** to the function **np.array**. You can also create a two-dimensional array by providing multiple lists within square brackets.[4] Instead of a two-dimensional array, we will often call this data type a matrix. Matrices are important tools for econometric analyses. Appendix D of Wooldridge (2019) introduces the basic concepts of matrix algebra.[5]

The syntax for defining a **numpy** array is:

```
testarray1D = np.array( list )
testarray2D = np.array( [ list1, list2, list3 ] )
```

Within a provided list, the **numpy** array requires a homogeneous data type. If you enter lists including elements of different type, **numpy** will convert them to a homogeneous data type (for example, **np.array(['a', 2])**, becomes an array of strings).

Indexing one-dimensional arrays is similar to the procedure with the data type **list**. Two-dimensional arrays are accessed by two comma separated values within the square brackets. The first number gives the row, the second number gives the column (starting at 0 for the first row or column). Just as with a **list**, accessing ranges of values with "**:**" excludes the upper limit. There are a lot more possibilities and Script 1.9 (`Numpy-Arrays.py`) demonstrates some of them.

[4]You can use higher dimensional arrays by typing more square brackets, but we will not need more than two dimensions in what follows.

[5]The strippped-down European and African textbook Wooldridge (2014) does not include the Appendix on matrix algebra.

Script 1.9: Numpy-Arrays.py

```
import numpy as np

# define arrays in numpy:
testarray1D = np.array([1, 5, 41.3, 2.0])
print(f'type(testarray1D): {type(testarray1D)}\n')

testarray2D = np.array([[4, 9, 8, 3],
                        [2, 6, 3, 2],
                        [1, 1, 7, 4]])

# get dimensions of testarray2D:
dim = testarray2D.shape
print(f'dim: {dim}\n')

# access elements by indices:
third_elem = testarray1D[2]
print(f'third_elem: {third_elem}\n')

second_third_elem = testarray2D[1, 2]  # element in 2nd row and 3rd column
print(f'second_third_elem: {second_third_elem}\n')

second_to_third_col = testarray2D[:, 1:3]  # each row in the 2nd and 3rd column
print(f'second_to_third_col: \n{second_to_third_col}\n')

# access elements by lists:
first_third_elem = testarray1D[[0, 2]]
print(f'first_third_elem: {first_third_elem}\n')

# same with Boolean lists:
first_third_elem2 = testarray1D[[True, False, True, False]]
print(f'first_third_elem2: {first_third_elem2}\n')

k = np.array([[True, False, False, False],
              [False, False, True, False],
              [True, False, True, False]])
elem_by_index = testarray2D[k]  # 1st elem in 1st row, 3rd elem in 2nd row...
print(f'elem_by_index: {elem_by_index}\n')
```

Output of Script 1.9: Numpy-Arrays.py

```
type(testarray1D): <class 'numpy.ndarray'>

dim: (3, 4)

third_elem: 41.3

second_third_elem: 3

second_to_third_col:
[[9 8]
 [6 3]
 [1 7]]

first_third_elem: [ 1.  41.3]

first_third_elem2: [ 1.  41.3]

elem_by_index: [4 3 1 7]
```

numpy has also some predefined and useful special cases of one and two-dimensional arrays. We show some of them in Script 1.10 (Numpy-SpecialCases.py).

Script 1.10: Numpy-SpecialCases.py

```
import numpy as np

# array of integers defined by the arguments start, end and sequence length:
sequence = np.linspace(0, 2, num=11)
print(f'sequence: \n{sequence}\n')

# sequence of integers starting at 0, ending at 5-1:
sequence_int = np.arange(5)
print(f'sequence_int: \n{sequence_int}\n')

# initialize array with each element set to zero:
zero_array = np.zeros((4, 3))
print(f'zero_array: \n{zero_array}\n')

# initialize array with each element set to one:
one_array = np.ones((2, 5))
print(f'one_array: \n{one_array}\n')

# uninitialized array (filled with arbitrary nonsense elements):
empty_array = np.empty((2, 3))
print(f'empty_array: \n{empty_array}\n')
```

Output of Script 1.10: Numpy-SpecialCases.py

```
sequence:
[0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2. ]

sequence_int:
[0 1 2 3 4]

zero_array:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]

one_array:
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]

empty_array:
[[-1.72723371e-077 -1.72723371e-077 -1.73060214e-077]
 [-1.49457718e-154  2.24183079e-314  4.17201348e-309]]
```

Table 1.3 lists important functions and methods in **numpy**. We can apply them to the data type **ndarray**, but they usually work for many built-in types too. Functions are often **vectorized** meaning that they are applied to each of the elements separately (in a very efficient way). Methods on an object referenced by **x** are invoked by using the **x.somemethod()** syntax discussed above. Script 1.11 (Numpy-Operations.py) provides examples to see them in action. We will see in Section 1.5 how to obtain descriptive statistics with **numpy**.

Table 1.3. Important **numpy** Functions and Methods

add(x, y) or **x+y**	Element-wise sum of all elements in **x** and **y**
subtract(x, y) or **x-y**	Element-wise subtraction of all elements in **x** and **y**
divide(x, y) or **x/y**	Element-wise division of all elements in **x** and **y**
multiply(x, y) or **x*y**	Element-wise multiplication of all elements in **x** and **y**
exp(x)	Element-wise exponential of all elements in **x**
sqrt(x)	Element-wise square root of all elements in **x**
log(x)	Element-wise natural logarithm of all elements in **x**
linalg.inv(x)	Inverse of **x**
x.sum()	Sum of all elements in **x**
x.min()	Minimum of all elements in **x**
x.max()	Maximum of all elements in **x**
x.dot(y) or **x@y**	Matrix multiplication of **x** and **y**
x.transpose() or **x.T**	Transpose of **x**

numpy has a powerful matrix algebra system. **Basic matrix algebra** includes:

- Matrix addition using the operator **+** as long as the matrices have the same dimensions.
- The operator ***** does <u>not</u> do matrix multiplication but rather element-wise multiplication.
- Matrix multiplication is done with the operator **@** (or the **dot** method) as long as the dimensions of the matrices match.
- Transpose of a matrix **X**: as **X.T**
- Inverse of a matrix **X**: as **linalg.inv(X)**

The examples in Script 1.11 (`Numpy-Operations.py`) should help to understand the workings of these basic operations. In order to see how the OLS estimator for the multiple regression model can be calculated using matrix algebra, see Section 3.2.

Script 1.11: Numpy-Operations.py

```
import numpy as np

# define an arrays in numpy:
mat1 = np.array([[4, 9, 8],
                 [2, 6, 3]])
mat2 = np.array([[1, 5, 2],
                 [6, 6, 0],
                 [4, 8, 3]])

# use a numpy function:
result1 = np.exp(mat1)
print(f'result1: \n{result1}\n')

result2 = mat1 + mat2[[0, 1]]  # same as np.add(mat1, mat2[[0, 1]])
print(f'result2: \n{result2}\n')

# use a method:
mat1_tr = mat1.transpose()
print(f'mat1_tr: \n{mat1_tr}\n')

# matrix algebra:
matprod = mat1.dot(mat2)  # same as  mat1 @ mat2
print(f'matprod: \n{matprod}\n')
```

Output of Script 1.11: `Numpy-Operations.py`

```
result1:
[[5.45981500e+01 8.10308393e+03 2.98095799e+03]
 [7.38905610e+00 4.03428793e+02 2.00855369e+01]]

result2:
[[ 5 14 10]
 [ 8 12  3]]

mat1_tr:
[[4 2]
 [9 6]
 [8 3]]

matprod:
[[ 90 138  32]
 [ 50  70  13]]
```

1.2.4. Objects in `pandas`

The module **`pandas`** builds on top of data types introduced in previous sections and allows us to work with something we will encounter almost every time we discuss an econometric application: a data frame.[6] A data frame is a structure that collects several variables and can be thought of as a rectangular shape with the rows representing the observational units and the columns representing the variables. A data frame can contain variables of different data types (for example a numerical **`list`**, a one-dimensional **`ndarray`**, **`str`** and so on). Before you start working with **`pandas`**, make sure that it is installed.[7] The standard alias of this module is **`pd`**, so when working with **`pandas`**, the first line of code always is:

```
import pandas as pd
```

The most important data type in **`pandas`** is **`DataFrame`**, which we will often simply refer to as "data frame". One strength of **`pandas`** is the existence of a whole set of operations that work on the index of a **`DataFrame`**. The index contains information on the observational unit, like the person answering a questionnaire or the date of a stock price you want to work with. Script 1.12 (`Pandas.py`) shows the definition of a variable with data type **`DataFrame`** by providing a **`dict`** to the function **`pd.DataFrame`**. The definition of an index, in this example a date with monthly frequency (**`freq='M'`**), is also demonstrated. Accessing elements of a variable **`df`** referencing an object of data type **`DataFrame`** can be done in multiple ways:

- Access columns/ variables by name:
  `df['varname1']` or **`df[['varname1', 'varname2',...]]`**
- Access rows/ observations by integer positions *i* to *j*: **`df[i:(j+1)]`** (also works with the index names of **`df`**)
- Access variables *and* observations by names:
  `df.loc['rowname', 'colname']`
- Access variables *and* observations by row and column integer positions *i* and *j*:
  `df.iloc[i, j]`

[6]For more information about the module, see McKinney (2011).
[7]The module **`pandas`** is part of the Anaconda distribution.

If you define a **DataFrame** by a combination of several **DataFrame**s, they are automatically matched by their indices.

Script 1.12: Pandas.py

```
import numpy as np
import pandas as pd

# define a pandas DataFrame:
icecream_sales = np.array([30, 40, 35, 130, 120, 60])
weather_coded = np.array([0, 1, 0, 1, 1, 0])
customers = np.array([2000, 2100, 1500, 8000, 7200, 2000])
df = pd.DataFrame({'icecream_sales': icecream_sales,
                   'weather_coded': weather_coded,
                   'customers': customers})

# define and assign an index (six ends of month starting in April, 2010)
# (details on generating indices are given in Chapter 10):
ourIndex = pd.date_range(start='04/2010', freq='M', periods=6)
df.set_index(ourIndex, inplace=True)

# print the DataFrame
print(f'df: \n{df}\n')

# access columns by variable names:
subset1 = df[['icecream_sales', 'customers']]
print(f'subset1: \n{subset1}\n')

# access second to fourth row:
subset2 = df[1:4]  # same as df['2010-05-31':'2010-07-31']
print(f'subset2: \n{subset2}\n')

# access rows and columns by index and variable names:
subset3 = df.loc['2010-05-31', 'customers']  # same as df.iloc[1,2]
print(f'subset3: \n{subset3}\n')

# access rows and columns by index and variable integer positions:
subset4 = df.iloc[1:4, 0:2]
# same as df.loc['2010-05-31':'2010-07-31', ['icecream_sales','weather']]
print(f'subset4: \n{subset4}\n')
```

Output of Script 1.12: Pandas.py

```
df: 
            icecream_sales  weather_coded  customers
2010-04-30              30              0       2000
2010-05-31              40              1       2100
2010-06-30              35              0       1500
2010-07-31             130              1       8000
2010-08-31             120              1       7200
2010-09-30              60              0       2000

subset1: 
            icecream_sales  customers
2010-04-30              30       2000
2010-05-31              40       2100
2010-06-30              35       1500
2010-07-31             130       8000
2010-08-31             120       7200
2010-09-30              60       2000
```

```
subset2:
            icecream_sales  weather_coded  customers
2010-05-31              40              1       2100
2010-06-30              35              0       1500
2010-07-31             130              1       8000

subset3:
2100

subset4:
            icecream_sales  weather_coded
2010-05-31              40              1
2010-06-30              35              0
2010-07-31             130              1
```

Table 1.4. Important **pandas** Methods

df.head()	First 5 observations in **df**
df.tail()	Last 5 observations in **df**
df.describe()	Print descriptive statistics
df.set_index(x)	Set the index of **df** as **x**
df['x'] or **df.x**	Access **x** in **df**
df.iloc(i, j)	Access variables and observations in **df** by integer position
df.loc(names_i, names_j)	Access variables and observations in **df** by names
df['x'].shift(i)	Creates a by i rows shifted variable of **x**
df['x'].diff(i)	Creates a variable that contains the ith difference of **x**
df.groupby('x').function()	Apply a function to subgroups of **df** according to **x**

Many economic variables of interest have a qualitative rather than quantitative interpretation. They only take a finite set of values and the outcomes don't necessarily have a numerical meaning. Instead, they represent **qualitative** information. Examples include gender, academic major, grade, marital status, state, product type or brand. In some of these examples, the order of the outcomes has a natural interpretation (such as the grades), in others, it does not (such as the state).

As a specific example, suppose we have asked our customers to rate our product on a scale between 0 (="bad"), 1 (="okay"), and 2 (="good"). We have stored the answers of our ten respondents in terms of the numbers 0,1, and 2 in a list. We could work directly with these numbers, but often, it is convenient to use so-called data type **Categorical**. One advantage is that we can attach labels to the outcomes. We extend a modified example in Script 1.13 (`Pandas-Operations.py`), where the variable **weather** is coded and demonstrate how to assign meaningful labels. The example also includes some methods from Table 1.4, i.e. lag variables and calling methods on subgroups of the data frame. The comments explain the effect of the respective action:

Script 1.13: **Pandas-Operations.py**

```
import numpy as np
import pandas as pd

# define a pandas DataFrame:
icecream_sales = np.array([30, 40, 35, 130, 120, 60])
weather_coded = np.array([0, 1, 0, 1, 1, 0])
customers = np.array([2000, 2100, 1500, 8000, 7200, 2000])
df = pd.DataFrame({'icecream_sales': icecream_sales,
                   'weather_coded': weather_coded,
                   'customers': customers})

# define and assign an index (six ends of month starting in April, 2010)
# (details on generating indices are given in Chapter 10):
ourIndex = pd.date_range(start='04/2010', freq='M', periods=6)
df.set_index(ourIndex, inplace=True)

# include sales two months ago:
df['icecream_sales_lag2'] = df['icecream_sales'].shift(2)
print(f'df: \n{df}\n')

# use a pandas.Categorical object to attach labels (0 = bad; 1 = good):
df['weather'] = pd.Categorical.from_codes(codes=df['weather_coded'],
                                          categories=['bad', 'good'])
print(f'df: \n{df}\n')

# mean sales for each weather category:
group_means = df.groupby('weather').mean()
print(f'group_means: \n{group_means}\n')
```

Output of Script 1.13: **Pandas-Operations.py**

```
df:
            icecream_sales  weather_coded  customers  icecream_sales_lag2
2010-04-30              30              0       2000                  NaN
2010-05-31              40              1       2100                  NaN
2010-06-30              35              0       1500                 30.0
2010-07-31             130              1       8000                 40.0
2010-08-31             120              1       7200                 35.0
2010-09-30              60              0       2000                130.0

df:
            icecream_sales  weather_coded  ...  icecream_sales_lag2  weather
2010-04-30              30              0  ...                  NaN      bad
2010-05-31              40              1  ...                  NaN     good
2010-06-30              35              0  ...                 30.0      bad
2010-07-31             130              1  ...                 40.0     good
2010-08-31             120              1  ...                 35.0     good
2010-09-30              60              0  ...                130.0      bad

[6 rows x 5 columns]

group_means:
         icecream_sales  weather_coded    customers  icecream_sales_lag2
weather
bad           41.666667            0.0  1833.333333                 80.0
good          96.666667            1.0  5766.666667                 37.5
```

1.3. External Data

In previous sections, we entered all of our data manually in the script files. This is a very untypical way of getting data into our computer and we will introduce more useful alternatives. These are based on the fact that many data sets are already stored somewhere else in data formats that *Python* can handle.

1.3.1. Data Sets in the Examples

We will reproduce many of the examples from Wooldridge (2019). The companion web site of the textbook provides the sample data sets in different formats. If you have an access code that came with the textbook, they can be downloaded free of charge. The Stata data sets are also made available online at the "Instructional Stata Datasets for econometrics" collection from Boston College, maintained by Christopher F. Baum.[8]

Fortunately, we do not have to download each data set manually and import them by the functions discussed in Section 1.3.2. Instead, we can use the external module **wooldridge**. It is not part of the Anaconda distribution and you have to install **wooldridge** as explained in Section 1.1.3. When working with **wooldridge**, the first line of code always is:

```
import wooldridge as woo
```

Script 1.14 (`Wooldridge.py`) demonstrates the first lines of a typical example in this book. As you see, we are dealing with a **pandas** data type, so all the methods from the previous section are applicable.

Script 1.14: `Wooldridge.py`

```
import wooldridge as woo

# load data:
wage1 = woo.dataWoo('wage1')

# get type:
print(f'type(wage1): \n{type(wage1)}\n')

# get an overview:
print(f'wage1.head(): \n{wage1.head()}\n')
```

Output of Script 1.14: `Wooldridge.py`

```
type(wage1):
<class 'pandas.core.frame.DataFrame'>

wage1.head():
   wage  educ  exper  tenure  ...  servocc     lwage  expersq  tenursq
0  3.10    11      2       0  ...        0  1.131402        4        0
1  3.24    12     22       2  ...        1  1.175573      484        4
2  3.00    11      2       0  ...        0  1.098612        4        0
3  6.00     8     44      28  ...        0  1.791759     1936      784
4  5.30    12      7       2  ...        0  1.667707       49        4

[5 rows x 24 columns]
```

[8] The address is `https://econpapers.repec.org/paper/bocbocins/`.

Figure 1.5. Examples of Text Data Files

(a) `sales.txt`

```
year product1 product2 product3
2008 0 1 2
2009 3 2 4
2010 6 3 4
2011 9 5 2
2012 7 9 3
2013 8 6 2
```

(b) `sales.csv`

```
2008,0,1,2
2009,3,2,4
2010,6,3,4
2011,9,5,2
2012,7,9,3
2013,8,6,2
```

1.3.2. Import and Export of Data Files

Probably all software packages that handle data are capable of working with data stored as text files. This makes them a natural way to exchange data between different programs and users. Common file name extensions for such data files are `RAW`, `CSV` or `TXT`. Most statistics and spreadsheet programs come with their own file format to save and load data. While it is basically always possible to exchange data via text files, it might be convenient to be able to directly read or write data in the native format of some other software.

Fortunately, the **pandas** toolbox provides the possibility for importing and exporting data from/to text files and many programs. This includes, for example,

- Text file (`TXT`) with **read_table** and **to_table**,
- CSV (`CSV`) with **read_csv** and **to_csv**,
- MS Excel (`XLS` and `XLSX`) with **read_excel** and **to_excel**,
- Stata (`DTA`) with **read_stata** and **to_stata**,
- SAS (`XPORT` and `SSD`) with **read_sas** and **to_sas**.

Figure 1.5 shows two flavors of a raw text file containing the same data. The file `sales.txt` contains a header with the variable names. In file `sales.csv`, the columns are separated by a comma.

Text files for storing data come in different flavors, mainly differing in how the columns of the table are separated. The **pandas** commands **read_table** and **read_csv** provides possibilities for reading many flavors of text files which are then stored as a **DataFrame**. Script 1.15 (`Import-Export.py`) demonstrates the import and export of the files shown in Figure 1.5. In this example, data files are stored in and exported to the folder `data`.

Script 1.15: Import-Export.py

```
import pandas as pd

# import csv with pandas:
df1 = pd.read_csv('data/sales.csv', delimiter=',', header=None,
                  names=['year', 'product1', 'product2', 'product3'])
print(f'df1: \n{df1}\n')

# import txt with pandas:
df2 = pd.read_table('data/sales.txt', delimiter=' ')
print(f'df2: \n{df2}\n')

# add a row to df1:
df3 = df1.append({'year': 2014, 'product1': 10, 'product2': 8, 'product3': 2},
                 ignore_index=True)
print(f'df3: \n{df3}\n')

# export with pandas:
df3.to_csv('data/sales2.csv')
```

Output of Script 1.15: Import-Export.py

```
df1:
   year  product1  product2  product3
0  2008         0         1         2
1  2009         3         2         4
2  2010         6         3         4
3  2011         9         5         2
4  2012         7         9         3
5  2013         8         6         2

df2:
   year  product1  product2  product3
0  2008         0         1         2
1  2009         3         2         4
2  2010         6         3         4
3  2011         9         5         2
4  2012         7         9         3
5  2013         8         6         2

df3:
   year  product1  product2  product3
0  2008         0         1         2
1  2009         3         2         4
2  2010         6         3         4
3  2011         9         5         2
4  2012         7         9         3
5  2013         8         6         2
6  2014        10         8         2
```

The command **read_csv** includes many optional arguments that can be added. Many of these arguments are detected automatically by **pandas**, but you can also specify them explicitly. The most important arguments are:

- **header**: Integer specifying the row that includes the variable names. Can also be **None**.
- **sep**: Often columns are separated by a comma, i.e. **sep=','** (default). Instead, an arbitrary other character can be given. **sep=';'** might be another relevant example of a separator.
- **names**: If no header is specified, you can provide a **list** of variable names.
- **index_col**: The values in column **index_col** are used as an index.

1.3.3. Data from other Sources

The last part of this section deals with importing data from other sources than local files on your computer. We will use an extension of **pandas** called **pandas_datareader**, which makes it straightforward to query online databases. It is not part of the Anaconda distribution and you have to install **pandas_datareader** as explained in Section 1.1.3. Script 1.16 (`Import-StockData.py`) demonstrates the workflow of importing stock data of Ford Motor Company. All you have to do is specify start and end date and the data source, which is Yahoo Finance in this case.

Script 1.16: Import-StockData.py

```
import pandas_datareader as pdr

# download data for 'F' (= Ford Motor Company) and define start and end:
tickers = ['F']
start_date = '2014-01-01'
end_date = '2015-12-31'

# use pandas_datareader for the import:
F_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# look at imported data:
print(f'F_data.head(): \n{F_data.head()}\n')
print(f'F_data.tail(): \n{F_data.tail()}\n')
```

Output of Script 1.16: Import-StockData.py

```
F_data.head():
Attributes  Adj Close  Close   High    Low   Open      Volume
Symbols             F      F      F      F      F           F
Date
2014-01-02  11.131250  15.44  15.45  15.28  15.42  31528500.0
2014-01-03  11.181718  15.51  15.64  15.30  15.52  46122300.0
2014-01-06  11.232182  15.58  15.76  15.52  15.72  42657600.0
2014-01-07  11.087993  15.38  15.74  15.35  15.73  54476300.0
2014-01-08  11.203343  15.54  15.71  15.51  15.60  48448300.0

F_data.tail():
Attributes  Adj Close  Close   High    Low   Open      Volume
Symbols             F      F      F      F      F           F
Date
2015-12-24  11.082371  14.31  14.37  14.25  14.35   9000100.0
2015-12-28  10.981693  14.18  14.34  14.16  14.28  13697500.0
2015-12-29  11.020413  14.23  14.30  14.15  14.28  18867800.0
2015-12-30  10.973947  14.17  14.26  14.12  14.23  13800300.0
2015-12-31  10.911990  14.09  14.16  14.04  14.14  19881000.0
```

1.4. Base Graphics with matplotlib

The module **matplotlib** is a popular and versatile tool for producing all kinds of graphs in *Python*. In this section, we discuss the overall base approach for producing graphs and the most important general types of graphs. We will only scratch the surface of **matplotlib**, but you will see most of the graph producing commands relevant for this book. For more information, see Hunter (2007). Some specific graphs used for descriptive statistics will be introduced in Section 1.5.

Before you start producing your own graphs, make sure that you use the Anaconda distribution or install **matplotlib** as explained in Section 1.1.3. When working with **matplotlib**, the first line of code always is:

```
import matplotlib.pyplot as plt
```

1.4.1. Basic Graphs

One very general type is a two-way graph with an abscissa and an ordinate that typically represent two variables like X and Y.

If we have data in two lists **x** and **y**, we can easily generate scatter plots, line plots or similar two-way graphs. The command **plot** is capable of these types of graphs and we will see some of the more specialized uses later on. Script 1.17 (Graphs-Basics.py) generates Figure 1.6(a) and demonstrates the minimum amount of code to produce a black line plot with all other options on default. Graphs are displayed in a separate *Python* window.[9] The last two lines export the created plot as a PDF file to the folder PyGraphs and reset the plot to create a completely new one. If the folder PyGraphs does not exist yet you must create one first to execute Script 1.17 (Graphs-Basics.py) without error.

Script 1.17: Graphs-Basics.py

```
import matplotlib.pyplot as plt

# create data:
x = [1, 3, 4, 7, 8, 9]
y = [0, 3, 6, 9, 7, 8]

# plot and save:
plt.plot(x, y, color='black')
plt.savefig('PyGraphs/Graphs-Basics-a.pdf')
plt.close()
```

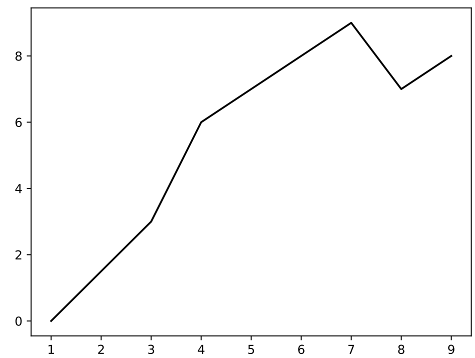
Two important arguments of the **plot** command are **linestyle** and **marker**. The argument **linestyle** takes the values **'-'** (the default), **'--'**, **':'**, and many more. The argument **marker** is empty by default, and can take **'o'**, **'v'**, and many more. Some resulting plots are shown in Figure 1.6. The code is shown in the appendix in Script 1.18 (Graphs-Basics2.py).

The **plot** command can be used to create a **function plot**, i.e. function values $y = f(x)$ are plotted against x. To plot a smooth function, the first step is to generate a fine grid of x values. In Script 1.19 (Graphs-Functions.py) we choose **linspace** from **numpy** and control the number of x values with **num**.[10] The following plotting of the function works exactly as in the previous example. We choose the quadratic function plotted in Figure 1.7(a) and the standard normal density (see Section 1.6) in Figure 1.7(b).

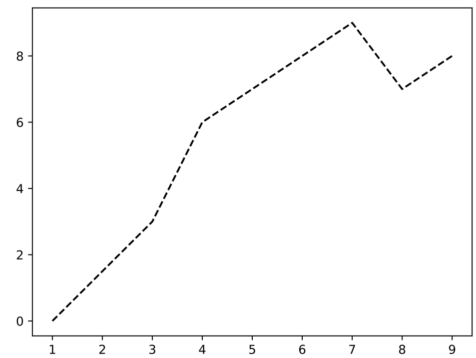
[9]If creating your graph requires the execution of multiple lines of code, make sure to execute them all at once and not line by line. Otherwise you might get multiple plots instead of one.
[10]The module **scipy** will be introduced in Section 1.6.

Figure 1.6. Examples of Point and Line Plots using **`plot(x, y)`**

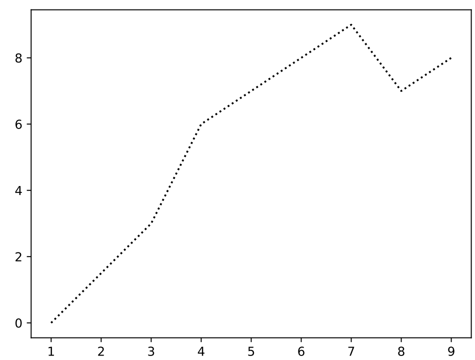
(a) see Script 1.17 (`Graphs-Basics.py`)



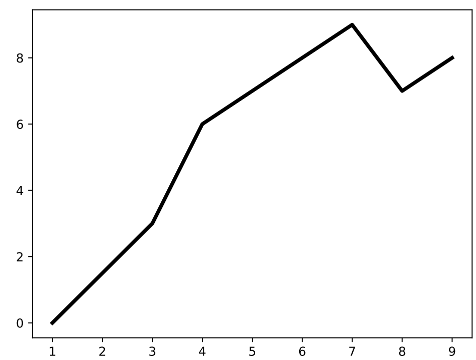
(b) **`linestyle='--'`**



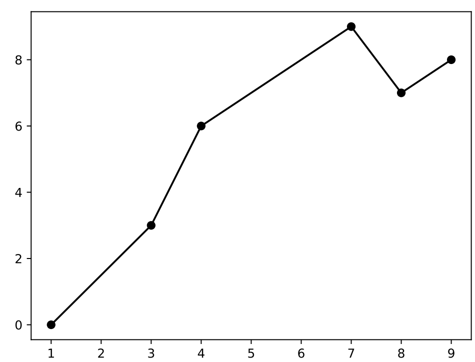
(c) **`linestyle=':'`**



(d) **`linewidth=3`**



(e) **`marker='o'`**



(f) **`marker='v', linestyle=''`**

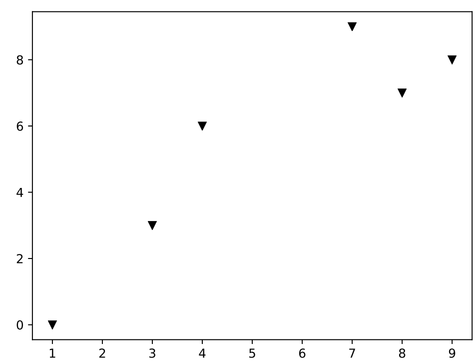
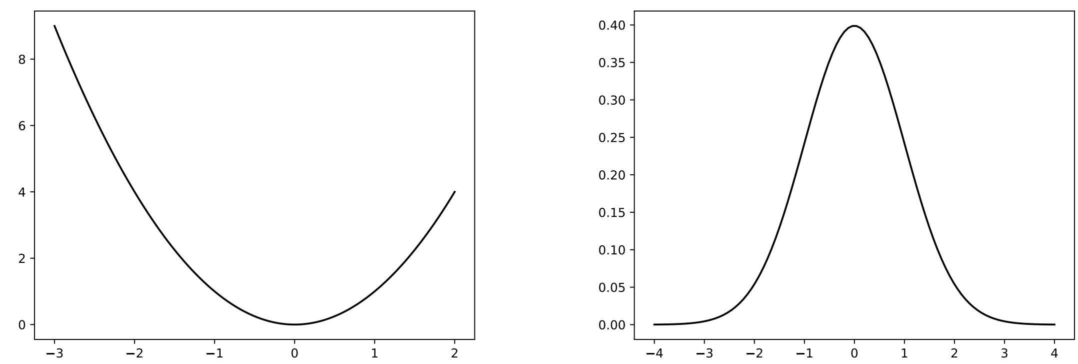


Figure 1.7. Examples of Function Plots using **`plot`**

(a) **`x ** 2`** (b) **`stats.norm.pdf(x)`**



Script 1.19: `Graphs-Functions.py`

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support of quadratic function
# (creates an array with 100 equispaced elements from -3 to 2):
x1 = np.linspace(-3, 2, num=100)
# function values for all these values:
y1 = x1 ** 2

# plot quadratic function:
plt.plot(x1, y1, linestyle='-', color='black')
plt.savefig('PyGraphs/Graphs-Functions-a.pdf')
plt.close()

# same for normal density:
x2 = np.linspace(-4, 4, num=100)
y2 = stats.norm.pdf(x2)

# plot normal density:
plt.plot(x2, y2, linestyle='-', color='black')
plt.savefig('PyGraphs/Graphs-Functions-b.pdf')
```

1.4.2. Customizing Graphs with Options

As already demonstrated in the examples, these plots can be adjusted very flexibly. A few examples:

- The width of the lines can be changed using the argument **`linewidth`** (default: **`linewidth=1`**).
- The size of the marker symbols can be changed using the argument **`markersize`** (default: **`markersize=1`**).
- The color of the lines and symbols can be changed using the argument **`color`**. It can be specified in several ways:

- By name: **'blue'**, **'green'**, **'red'**, **'cyan'**, **'magenta'**, **'yellow'**, **'black'**, **'white'**.
- Gray scale by a string encoding a number between 0 (black) and 1 (white), for example **plt.plot(x1, y1, linestyle='-', color='0.3')**.
- By RGBA values provided by (r, g, b, a) with each letter representing a number between 0 and 1, for example **plt.plot(x1, y1, linestyle='-', color=(0.9, 0.2, 0.1, 0.3))**.[11] This is useful for fine-tuning colors.

You can also add more elements to change the appearance of your plot:

- A title can be added using **title('My Title')**.
- The horizontal and vertical axis can be labeled using **xlabel('My x axis label')** and **ylabel('My y axis label')**.
- The limits of the horizontal and the vertical axis can be chosen using **xlim(min, max)** and **ylim(min, max)**, respectively.

For an example, see Script 1.20 (`Graphs-BuildingBlocks.py`) and Figure 1.8.

1.4.3. Overlaying Several Plots

Often, we want to plot more than one set of variables or multiple graphical elements. This is an easy task, because each plot is added to the previous one by default.[12]

Script 1.20 (`Graphs-BuildingBlocks.py`) shows an example that also demonstrates some of the options from the previous paragraph. Its result is shown in Figure 1.8.[13]

Script 1.20: Graphs-BuildingBlocks.py

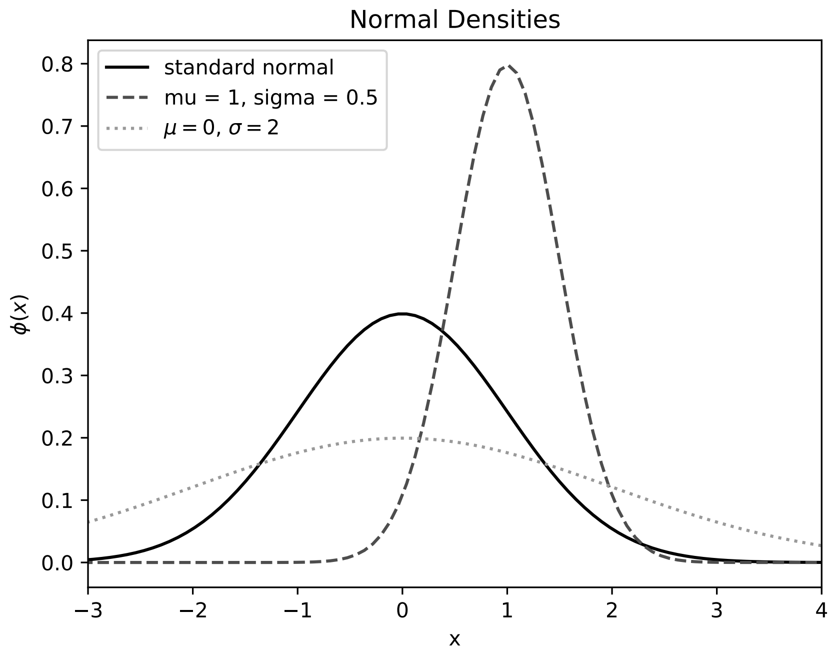
```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support for all normal densities:
x = np.linspace(-4, 4, num=100)
# get different density evaluations:
y1 = stats.norm.pdf(x, 0, 1)
y2 = stats.norm.pdf(x, 1, 0.5)
y3 = stats.norm.pdf(x, 0, 2)

# plot:
plt.plot(x, y1, linestyle='-', color='black', label='standard normal')
plt.plot(x, y2, linestyle='--', color='0.3', label='mu = 1, sigma = 0.5')
plt.plot(x, y3, linestyle=':', color='0.6', label='$\mu = 0$, $\sigma = 2$')
plt.xlim(-3, 4)
plt.title('Normal Densities')
plt.ylabel('$\phi(x)$')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/Graphs-BuildingBlocks.pdf')
```

[11] The RGB color model defines colors as a mix of the components red, green, and blue. **a** is optional and controls for transparency.

[12] To avoid this and reset your graph, use the command **close** after completing a graph.

[13] The module **scipy** will be introduced in Section 1.6.

Figure 1.8. Overlayed Plots



In this example, you can also see some useful commands for adding elements to an existing graph. Here are some (more) examples:

- **`axhline(y=value)`** adds a horizontal line at **`y`**.
- **`axvline(x=value)`** adds a vertical line at **`x`**.
- **`legend()`** adds a legend based on the string provided in each graphical element in **`label`**. **`matplotlib`** finds the best position.

In the legend, but also everywhere within a graph (title, axis labels, ...) we can also use Greek letters, equations, and similar features in a relatively straightforward way. This is done using respective TEX commands as demonstrated in Script 1.20 (`Graphs-BuildingBlocks.py`) and Figure 1.8.

1.4.4. Exporting to a File

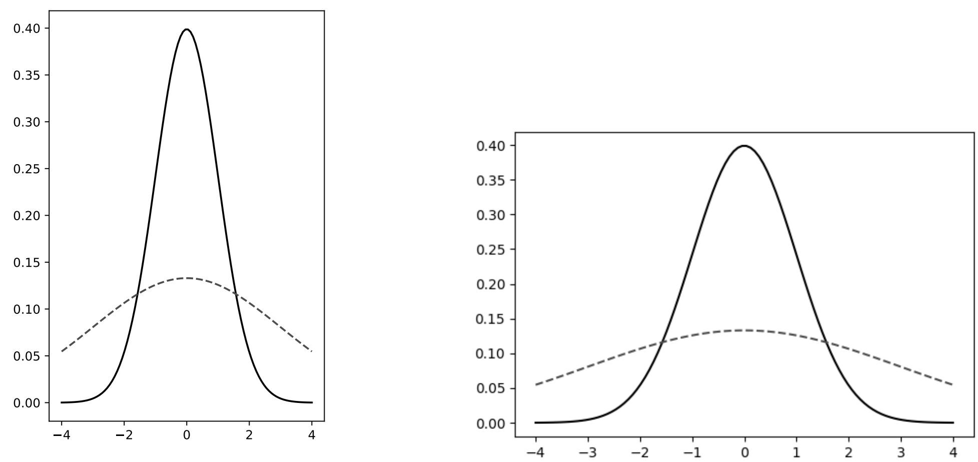
By default, a graph generated in one of the ways we discussed above will be displayed in its own window. *Python* offers the possibility to export the generated plots automatically using specific commands.

Among the different graphics formats, the `PNG` (Portable Network Graphics) format is very useful for saving plots to use them in a word processor and similar programs. For LATEX users, `PS`, `EPS` and `SVG` are available and `PDF` is very useful. You have already seen the export syntax in many examples:

```
plt.savefig('filepath/filename.format')
```

Figure 1.9. Examples of Exported Plots

(a) **plt.figure(figsize=(4, 6))** (b) **plt.figure(figsize=(6, 4))**



To set the width and height of your graph in inches, you start your code with **plt.figure(figsize=(width, height))**. Script 1.21 (Graphs-Export.py) and Figure 1.9 demonstrate the complete procedure.

Script 1.21: Graphs-Export.py

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support for all normal densities:
x = np.linspace(-4, 4, num=100)

# get different density evaluations:
y1 = stats.norm.pdf(x, 0, 1)
y2 = stats.norm.pdf(x, 0, 3)

# plot (a):
plt.figure(figsize=(4, 6))
plt.plot(x, y1, linestyle='-', color='black')
plt.plot(x, y2, linestyle='--', color='0.3')
plt.savefig('PyGraphs/Graphs-Export-a.pdf')
plt.close()

# plot (b):
plt.figure(figsize=(6, 4))
plt.plot(x, y1, linestyle='-', color='black')
plt.plot(x, y2, linestyle='--', color='0.3')
plt.savefig('PyGraphs/Graphs-Export-b.png')
```

1.5. Descriptive Statistics

The *Python* modules **pandas**, **numpy** and **matplotlib** offer many commands for descriptive statistics. In this section, we cover the most important ones for our purpose.

1.5.1. Discrete Distributions: Frequencies and Contingency Tables

Suppose we have a sample of the random variables X and Y stored in **numpy** or **pandas** data types **x** and **y**, respectively. For discrete variables, the most fundamental statistics are the frequencies of outcomes. The **numpy** command **unique(x, return_counts=True)** or **pandas** command **x.value_counts()** returns such a table of counts. If we are interested in the contingency table, i.e. the counts of each combination of outcomes for variables **x** and **y**, we provide it to the **crosstab** function in **pandas**. For getting the sample *shares* instead of the *counts*, we can change the functions argument **normalize**:

- The overall sample share: **crosstab(x, y, normalize='all')**
- The share within **x** values (row percentages): **crosstab(x, y, normalize='index')**
- The share within **y** values (column percentages): **crosstab(x, y, normalize='columns')**

As an example, we look at the data set `affairs` in Script 1.22 (`Descr-Tables.py`). We demonstrate the workings of the **numpy** and **pandas** commands with two variables:

- **kids** = 1 if the respondent has at least one child
- **ratemarr** = Rating of the own marriage (1=very unhappy, ... , 5=very happy)

Script 1.22: **Descr-Tables.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd

affairs = woo.dataWoo('affairs')

# adjust codings to [0-4] (Categoricals require a start from 0):
affairs['ratemarr'] = affairs['ratemarr'] - 1

# use a pandas.Categorical object to attach labels for "haskids":
affairs['haskids'] = pd.Categorical.from_codes(affairs['kids'],
                                               categories=['no', 'yes'])
# ... and "marriage" (for example: 0 = 'very unhappy', 1 = 'unhappy',...):
mlab = ['very unhappy', 'unhappy', 'average', 'happy', 'very happy']
affairs['marriage'] = pd.Categorical.from_codes(affairs['ratemarr'],
                                                categories=mlab)

# frequency table in numpy (alphabetical order of elements):
ft_np = np.unique(affairs['marriage'], return_counts=True)
unique_elem_np = ft_np[0]
counts_np = ft_np[1]
print(f'unique_elem_np: \n{unique_elem_np}\n')
print(f'counts_np: \n{counts_np}\n')

# frequency table in pandas:
ft_pd = affairs['marriage'].value_counts()
print(f'ft_pd: \n{ft_pd}\n')
```

```
# frequency table with groupby:
ft_pd2 = affairs['marriage'].groupby(affairs['haskids']).value_counts()
print(f'ft_pd2: \n{ft_pd2}\n')

# contingency table in pandas:
ct_all_abs = pd.crosstab(affairs['marriage'], affairs['haskids'], margins=3)
print(f'ct_all_abs: \n{ct_all_abs}\n')
ct_all_rel = pd.crosstab(affairs['marriage'], affairs['haskids'], normalize='all')
print(f'ct_all_rel: \n{ct_all_rel}\n')

# share within "marriage" (i.e. within a row):
ct_row = pd.crosstab(affairs['marriage'], affairs['haskids'], normalize='index')
print(f'ct_row: \n{ct_row}\n')

# share within "haskids"  (i.e. within a column):
ct_col = pd.crosstab(affairs['marriage'], affairs['haskids'], normalize='columns')
print(f'ct_col: \n{ct_col}\n')
```

Output of Script 1.22: `Descr-Tables.py`

```
unique_elem_np:
['average' 'happy' 'unhappy' 'very happy' 'very unhappy']

counts_np:
[ 93 194  66 232  16]

ft_pd:
very happy      232
happy           194
average          93
unhappy          66
very unhappy     16
Name: marriage, dtype: int64

ft_pd2:
haskids  marriage
no       very happy       96
         happy            40
         average          24
         unhappy           8
         very unhappy      3
yes      happy           154
         very happy      136
         average          69
         unhappy          58
         very unhappy     13
Name: marriage, dtype: int64

ct_all_abs:
haskids        no  yes  All
marriage
very unhappy    3   13   16
unhappy         8   58   66
average        24   69   93
happy          40  154  194
very happy     96  136  232
All           171  430  601
```

```
ct_all_rel:
haskids              no       yes
marriage
very unhappy  0.004992  0.021631
unhappy       0.013311  0.096506
average       0.039933  0.114809
happy         0.066556  0.256240
very happy    0.159734  0.226290

ct_row:
haskids              no       yes
marriage
very unhappy  0.187500  0.812500
unhappy       0.121212  0.878788
average       0.258065  0.741935
happy         0.206186  0.793814
very happy    0.413793  0.586207

ct_col:
haskids              no       yes
marriage
very unhappy  0.017544  0.030233
unhappy       0.046784  0.134884
average       0.140351  0.160465
happy         0.233918  0.358140
very happy    0.561404  0.316279
```

In the *Python* script, we first generate **`Categorical`** versions of the two variables of interest from the coded values provided by the data set `affairs`. In this way, we can generate tables with meaningful labels instead of numbers for the outcomes, see Section 1.2.4. Then different tables are produced. Of the 601 respondents, 430 (=71.5%) have children. Overall, 16 respondents report to be very unhappy with their marriage and 232 respondents are very happy. In the contingency table with counts, we see for example that 136 respondents are very happy and have kids.

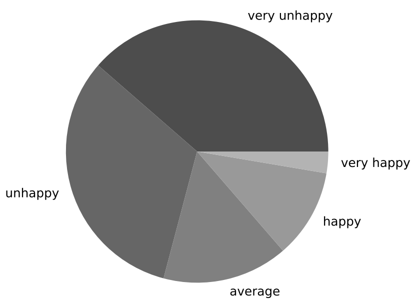
The table reporting shares within the rows (**`ct_row`**) tells us that for example 81.25% of very unhappy individuals have children and only 58.6% of very happy respondents have kids. The last table reports the distribution of marriage ratings separately for people with and without kids: 56.1% of the respondents without kids are very happy, whereas only 31.6% of those with kids report to be very happy with their marriage. Before drawing any conclusions for your own family planning, please keep on studying econometrics at least until you fully appreciate the difference between correlation and causation!

There are several ways to graphically depict the information in these tables. Script 1.23 (`Descr-Figures.py`) demonstrates the creation of basic pie and bar charts using the commands **`pie`** and **`bar`**, respectively. These figures can of course be tweaked in many ways, see the help pages and the general discussions of graphics in Section 1.4. The best way to explore the options is to tinker with the specification and observe the results.

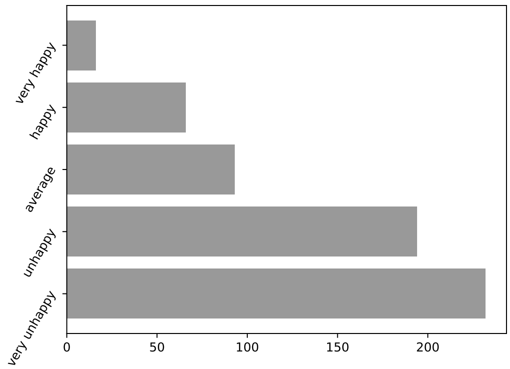
Script 1.23: `Descr-Figures.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

affairs = woo.dataWoo('affairs')

# attach labels (see previous script):
affairs['ratemarr'] = affairs['ratemarr'] - 1
affairs['haskids'] = pd.Categorical.from_codes(affairs['kids'],
                                               categories=['no', 'yes'])
mlab = ['very unhappy', 'unhappy', 'average', 'happy', 'very happy']
affairs['marriage'] = pd.Categorical.from_codes(affairs['ratemarr'],
                                                categories=mlab)

# counts for all graphs:
counts = affairs['marriage'].value_counts()
counts_bykids = affairs['marriage'].groupby(affairs['haskids']).value_counts()
counts_yes = counts_bykids['yes']
counts_no = counts_bykids['no']

# pie chart (a):
grey_colors = ['0.3', '0.4', '0.5', '0.6', '0.7']
plt.pie(counts, labels=mlab, colors=grey_colors)
plt.savefig('PyGraphs/Descr-Pie.pdf')
plt.close()

# horizontal bar chart (b):
y_pos = [0, 1, 2, 3, 4]  # the y locations for the bars
plt.barh(y_pos, counts, color='0.6')
plt.yticks(y_pos, mlab, rotation=60)  # add and adjust labeling
plt.savefig('PyGraphs/Descr-Bar1.pdf')
plt.close()

# stacked bar plot (c):
x_pos = [0, 1, 2, 3, 4]  # the x locations for the bars
plt.bar(x_pos, counts_yes, width=0.4, color='0.6', label='Yes')
# with 'bottom=counts_yes' bars are added on top of previous ones:
plt.bar(x_pos, counts_no, width=0.4, bottom=counts_yes, color='0.3', label='No')
plt.ylabel('Counts')
plt.xticks(x_pos, mlab)  # add labels on x axis
plt.legend()
plt.savefig('PyGraphs/Descr-Bar2.pdf')
plt.close()

# grouped bar plot (d)
# add left bars first and move bars to the left:
x_pos_leftbar = [-0.2, 0.8, 1.8, 2.8, 3.8]
plt.bar(x_pos_leftbar, counts_yes, width=0.4, color='0.6', label='Yes')
# add right bars first and move bars to the right:
x_pos_rightbar = [0.2, 1.2, 2.2, 3.2, 4.2]
plt.bar(x_pos_rightbar, counts_no, width=0.4, color='0.3', label='No')
plt.ylabel('Counts')
plt.xticks(x_pos, mlab)
plt.legend()
plt.savefig('PyGraphs/Descr-Bar3.pdf')
```

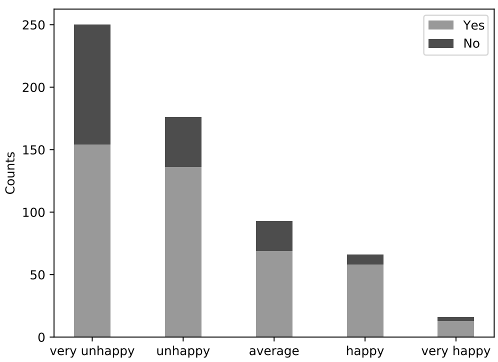
Figure 1.10. Pie and Bar Plots



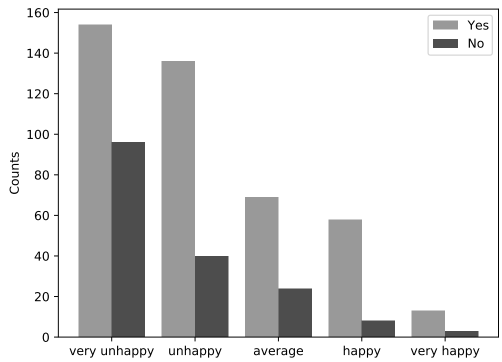
(a) Pie chart



(b) Bar plot



(c) Stacked bar plot



(d) Grouped bar plot

1.5.2. Continuous Distributions: Histogram and Density

For continuous variables, every observation has a distinct value. In practice, variables which have many (but not infinitely many) different values can be treated in the same way. Since each value appears only once (or a very few times) in the data, frequency tables or bar charts are not useful. Instead, the values can be grouped into intervals. The frequency of values within these intervals can then be tabulated or depicted in a histogram.

In the *Python* module **matplotlib**, the function **hist(x, options)** assigns observations to intervals which can be manually set or automatically chosen and creates a histogram which plots values of **x** against the count or density within the corresponding bin. The most relevant options are

- **bins=...**: Set the interval boundaries:
  - no **bins** specified: let *Python* choose number and position.
  - **bins=n** for a scalar **n**: select the *number* of bins, but let *Python* choose the position.
  - **bins=v** for a list **v**: explicitly set the boundaries.
- **density=True**: do not use the count but the density on the vertical axis.

Let's look at the data set CEOSAL1 which is described and used in Wooldridge (2019, Example 2.3). It contains information on the salary of CEOs and other information. We will try to depict the distribution of the return on equity (ROE), measured in percent. Script 1.24 (Histogram.py) generates the graphs of Figure 1.11. In Sub-figure (b), the **breaks** are manually chosen and not equally spaced. Setting **density=True** gives the densities on the vertical axis: The sample share of observations within a bin is therefore reflected by the *area* of the respective rectangle, not the height.

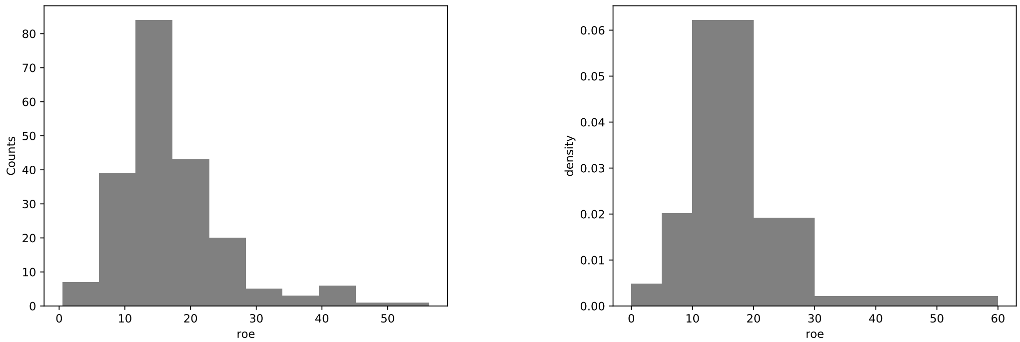
Script 1.24: Histogram.py

```
import wooldridge as woo
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe:
roe = ceosal1['roe']

# subfigure a (histogram with counts):
plt.hist(roe, color='grey')
plt.ylabel('Counts')
plt.xlabel('roe')
plt.savefig('PyGraphs/Histogram1.pdf')
plt.close()

# subfigure b (histogram with density and explicit breaks):
breaks = [0, 5, 10, 20, 30, 60]
plt.hist(roe, color='grey', bins=breaks, density=True)
plt.ylabel('density')
plt.xlabel('roe')
plt.savefig('PyGraphs/Histogram2.pdf')
```

A kernel density plot can be thought of as a more sophisticated version of a histogram. We cannot go into detail here, but an intuitive (and oversimplifying) way to think about it is this: We could create a histogram bin of a certain width, centered at an arbitrary point of x. We will do this for many points and plot these x values against the resulting densities. Here, we will not use this plot as an estimator of a population distribution but rather as a pretty alternative to a histogram for the descriptive characterization of the sample distribution. For details, see for example Silverman (1986).

In *Python*, generating a kernel density plot is straightforward with the module **statsmodels**: **nonparametric.KDEUnivariate(x).fit()** will automatically choose appropriate parameters

Figure 1.11. Histograms



(a) Histogram with counts (b) Histogram with density and explicit breaks

of the algorithm given the data and often produce a useful result.[14] Of course, these parameters (like the kernel and bandwidth for those who know what that is) can be set manually.

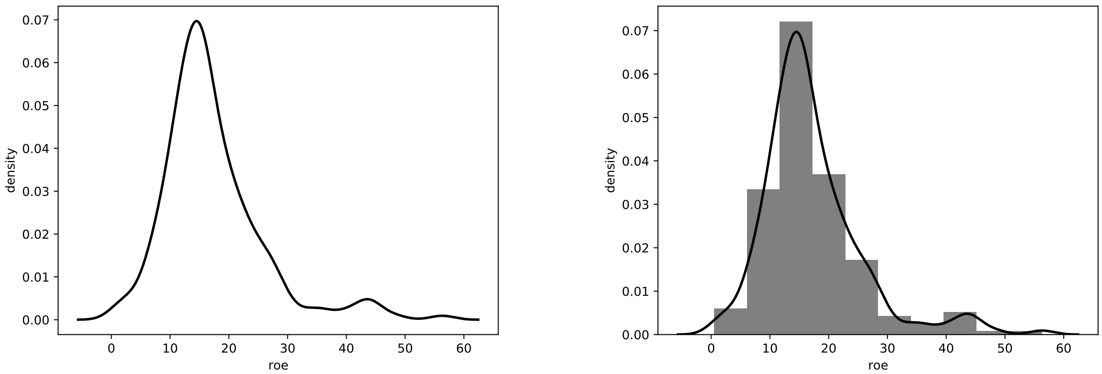
Script 1.25 (`KDensity.py`) demonstrates how the result of the density estimation can be plotted with **matplotlib** and generates the graphs of Figure 1.12. In Sub-figure (b), a histogram is overlayed with a kernel density plot.

Script 1.25: `KDensity.py`

```
import wooldridge as woo
import statsmodels.api as sm
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe:
roe = ceosal1['roe']

# estimate kernel density:
kde = sm.nonparametric.KDEUnivariate(roe)
kde.fit()

# subfigure a (kernel density):
plt.plot(kde.support, kde.density, color='black', linewidth=2)
plt.ylabel('density')
plt.xlabel('roe')
plt.savefig('PyGraphs/Density1.pdf')
plt.close()

# subfigure b (kernel density with overlayed histogram):
plt.hist(roe, color='grey', density=True)
plt.plot(kde.support, kde.density, color='black', linewidth=2)
plt.ylabel('density')
plt.xlabel('roe')
plt.savefig('PyGraphs/Density2.pdf')
```

[14] The module **statsmodels** will be introduced in Chapter 2.

Figure 1.12. Kernel Density Plots



(a) Kernel density

(b) Kernel density with histogram

1.5.3. Empirical Cumulative Distribution Function (ECDF)

The ECDF is a graph of all values x of a variable against the share of observations with a value less than or equal to x. A straightforward way to plot the ECDF for our ROE variable is shown in Script 1.26 (`Descr-ECDF.py`) and Figure 1.13. A more automated approach is the use of the **`statsmodels`** function **`distributions.empirical_distribution.ECDF(x)`**, which would give the same result.

For example, the value of the ECDF for point **`roe=15.5`** is 0.5. Half of the sample is less or equal to a ROE of 15.5%. In other words: the median ROE is 15.5%.

Script 1.26: `Descr-ECDF.py`

```
import wooldridge as woo
import numpy as np
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe:
roe = ceosal1['roe']

# calculate ECDF:
x = np.sort(roe)
n = x.size
y = np.arange(1, n + 1) / n  # generates cumulative shares of observations

# plot a step function:
plt.step(x, y, linestyle='-', color='black')
plt.xlabel('roe')
plt.savefig('PyGraphs/ecdf.pdf')
```

Figure 1.13. Empirical CDF

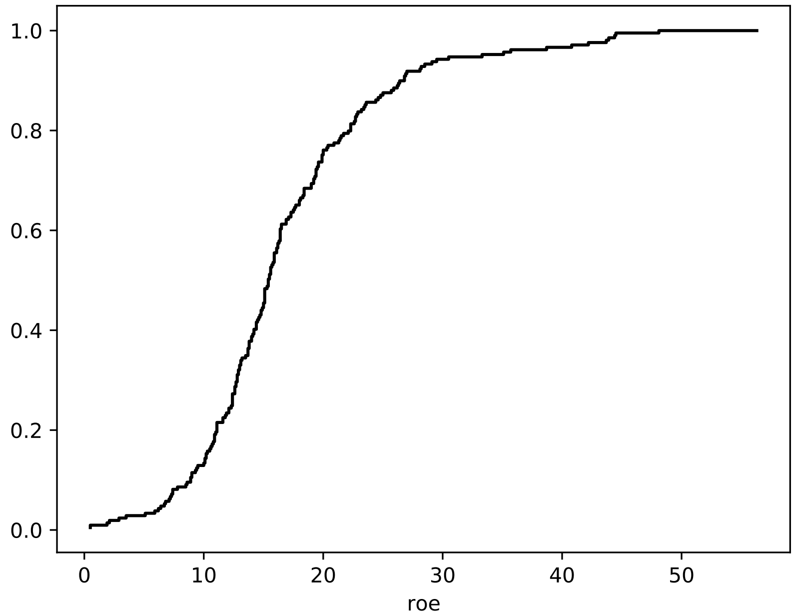


Table 1.5. `numpy` Functions for Descriptive Statistics

`mean(x)`	Sample average $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
`median(x)`	Sample median
`var(x, ddof=1)`	Sample variance $s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$
`std(x, ddof=1)`	Sample standard deviation $s_x = \sqrt{s_x^2}$
`cov(x, y)`	Sample covariance $c_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})$
`corrcoef(x, y)`	Sample correlation $r_{xy} = \frac{s_{xy}}{s_x \cdot s_y}$
`quantile(x, q)`	q quantile = $100 \cdot q$ percentile, e.g. `quantile(x, 0.5)` = sample median

1.5.4. Fundamental Statistics

The functions for calculating the most important descriptive statistics with **numpy** are listed in Table 1.5. Script 1.27 (Descr-Stats.py) demonstrates this using the CEOSAL1 data set we already introduced in Section 1.5.2.

Script 1.27: Descr-Stats.py

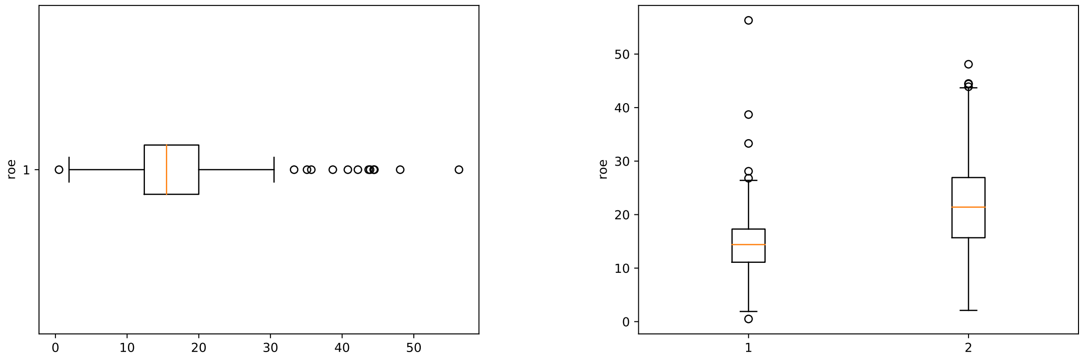
```
import wooldridge as woo
import numpy as np

ceosal1 = woo.dataWoo('ceosal1')

# extract roe and salary:
roe = ceosal1['roe']
salary = ceosal1['salary']

# sample average:
roe_mean = np.mean(salary)
print(f'roe_mean: {roe_mean}\n')

# sample median:
roe_med = np.median(salary)
print(f'roe_med: {roe_med}\n')

# standard deviation:
roe_s = np.std(salary, ddof=1)
print(f'roe_s: {roe_s}\n')

# correlation with ROE:
roe_corr = np.corrcoef(roe, salary)
print(f'roe_corr: \n{roe_corr}\n')
```

Output of Script 1.27: Descr-Stats.py

```
roe_mean: 1281.1196172248804

roe_med: 1039.0

roe_s: 1372.3453079588883

roe_corr:
[[1.         0.11484173]
 [0.11484173 1.        ]]
```

A box plot displays the median (the middle line), the upper and lower quartile (the box) and the extreme points graphically. Figure 1.14 shows two examples. 50% of the observations are within the interval covered by the box, 25% are above and 25% are below. The extreme points are marked by the "whiskers" and outliers are printed as separate dots. In **matplotlib**, box plots are generated using the **boxplot** command. We have to supply one or more data arrays and can alter the design flexibly with numerous options as demonstrated in Script 1.28 (Descr-Boxplot.py).

Figure 1.14. Box Plots



(a) Boxplot of one variable (b) Boxplot of two variables

Script 1.28: `Descr-Boxplot.py`

```
import wooldridge as woo
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe and salary:
roe = ceosal1['roe']
consprod = ceosal1['consprod']

# plotting descriptive statistics:
plt.boxplot(roe, vert=False)
plt.ylabel('roe')
plt.savefig('PyGraphs/Boxplot1.pdf')
plt.close()

# plotting descriptive statistics:
roe_cp0 = roe[consprod == 0]
roe_cp1 = roe[consprod == 1]

plt.boxplot([roe_cp0, roe_cp1])
plt.ylabel('roe')
plt.savefig('PyGraphs/Boxplot2.pdf')
```

Figure 1.14(a) shows how to get a horizontally aligned plot and Figure 1.14(b) demonstrates how to produce multiple boxplots for two sub groups. The variable `consprod` from the data set `ceosal1` is equal to 1 if the firm is in the consumer product business and 0 otherwise. Apparently, the ROE is much higher in this industry.

Table 1.6. `scipy` Functions for Statistical Distributions

Distribution	Param.	PMF/PDF	CDF	Quantile
Discrete distributions:				
Bernoulli	p	`bernoulli.pmf(`x,p`)`	`bernoulli.cdf(`x,p`)`	`bernoulli.ppf(`$q,\ p$`)`
Binomial	n,p	`binom.pmf(`x,n,p`)`	`binom.cdf(`x,n,p`)`	`binom.ppf(`q,n,p`)`
Hypergeom.	M,n,N	`hypergeom.pmf(`x,M,n,N`)`	`hypergeom.cdf(`x,M,n,N`)`	`hypergeom.ppf(`q,M,n,N`)`
Poisson	λ	`poisson.pmf(`x,λ`)`	`poisson.cdf(`x,λ`)`	`poisson.ppf(`q,λ`)`
Geometric	p	`geom.pmf(`x,p`)`	`geom.cdf(`x,p`)`	`geom.ppf(`q,p`)`
Continuous distributions:				
Uniform	a,b	`uniform.pdf(`$x,a,a+b$`)`	`uniform.cdf(`$x,a,a+b$`)`	`uniform.ppf(`$q,a,a+b$`)`
Logistic	—	`logistic.pdf(`x`)`	`logistic.cdf(`x`)`	`logistic.ppf(`q`)`
Exponential	λ	`expon.pdf(`x`,scale=`$1/\lambda$`)`	`expon.cdf(`x`,scale=`$1/\lambda$`)`	`expon.ppf(`q`,scale=`$1/\lambda$`)`
Std. normal	—	`norm.pdf(`x`)`	`norm.cdf(`x`)`	`norm.ppf(`q`)`
Normal	μ,σ	`norm.pdf(`x,μ,σ`)`	`norm.cdf(`x,μ,σ`)`	`norm.ppf(`q,μ,σ`)`
Lognormal	m,s	`lognorm.pdf(`q,s`,0,`m`)`	`lognorm.cdf(`x,s`,0,`m`)`	`lognorm.ppf(`q,s`,0,`m`)`
χ^2	n	`chi2.pdf(`x,n`)`	`chi2.cdf(`x,n`)`	`chi2.pdf(`q,n`)`
t	n	`t.pdf(`x,n`)`	`t.cdf(`x,n`)`	`t.pdf(`q,n`)`
F	m,n	`f.pdf(`x,m,n`)`	`f.cdf(`x,m,n`)`	`f.pdf(`q,m,n`)`

1.6. Probability Distributions

Appendix B of Wooldridge (2019) introduces the concepts of random variables and their probability distributions.[15] The module **scipy** has many functions for conveniently working with a large number of statistical distributions.[16] The commands for evaluating the probability density function (PDF) for continuous, the probability mass function (PMF) for discrete, and the cumulative distribution function (CDF) as well as the quantile function (inverse CDF) for the most relevant distributions are shown in Table 1.6. The functions are available after executing

```
import scipy.stats as stats
```

The module documentation defines the relation of a distribution's set of parameters and the function arguments in **scipy**. We will now briefly discuss each function type.

1.6.1. Discrete Distributions

Discrete random variables can only take a finite (or "countably infinite") set of values. The PMF $f(x) = P(X = x)$ gives the probability that a random variable X with this distribution takes the given value x. For the most important of those distributions (Bernoulli, Binomial, Hypergeometric[17], Poisson, and Geometric[18]), Table 1.6 lists the **scipy** functions that return the PMF for any value x given the parameters of the respective distribution. See the module documentation, if you are interested in the formal definitions of the PMFs.

For a specific example, let X denote the number of white balls we get when drawing with replacement 10 balls from an urn that includes 20% white balls. Then X has the Binomial distribution

[15] The stripped-down textbook for Europe and Africa Wooldridge (2014) does not include this appendix. But the material is pretty standard.

[16] **scipy** is part of the Anaconda distribution and more information about the module is given in Virtanen, Gommers, Oliphant, Haberland, Reddy, Cournapeau, Burovski, Peterson, Weckesser, Bright, van der Walt, Brett, Wilson, Jarrod Millman, Mayorov, Nelson, Jones, Kern, Larson, Carey, Polat, Feng, Moore, Vand erPlas, Laxalde, Perktold, Cimrman, Henriksen, Quintero, Harris, Archibald, Ribeiro, Pedregosa, van Mulbregt, and Contributors (2020).

[17] The parameters of the distribution are defined as follows: M is the total number of balls in an urn, n is the total number of marked balls in this urn, k is the number of drawn balls and x is number of drawn marked balls.

[18] x is the total number of trials, i.e. the number of failures in a sequence of Bernoulli trials before a success occurs plus the success trial.

with the parameters $n = 10$ and $p = 20\% = 0.2$. We know that the probability to get exactly $x \in \{0, 1, \ldots, 10\}$ white balls for this distribution is[19]

$$f(x) = \mathrm{P}(X = x) = \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x} = \binom{10}{x} \cdot 0.2^x \cdot 0.8^{10-x} \tag{1.1}$$

For example, the probability to get exactly $x = 2$ white balls is $f(2) = \binom{10}{2} \cdot 0.2^2 \cdot 0.8^8 = 0.302$. Of course, we can let *Python* do these calculations using basic *Python* commands we know from Section 1.1. More conveniently, we can also use the function **binom.pmf** for the Binomial distribution:

Script 1.29: PMF-binom.py

```
import scipy.stats as stats
import math

# pedestrian approach:
c = math.factorial(10) / (math.factorial(2) * math.factorial(10 - 2))
p1 = c * (0.2 ** 2) * (0.8 ** 8)
print(f'p1: {p1}\n')

# scipy function:
p2 = stats.binom.pmf(2, 10, 0.2)
print(f'p2: {p2}\n')
```

Output of Script 1.29: PMF-binom.py

```
p1: 0.30198988800000002

p2: 0.30198988799999993
```

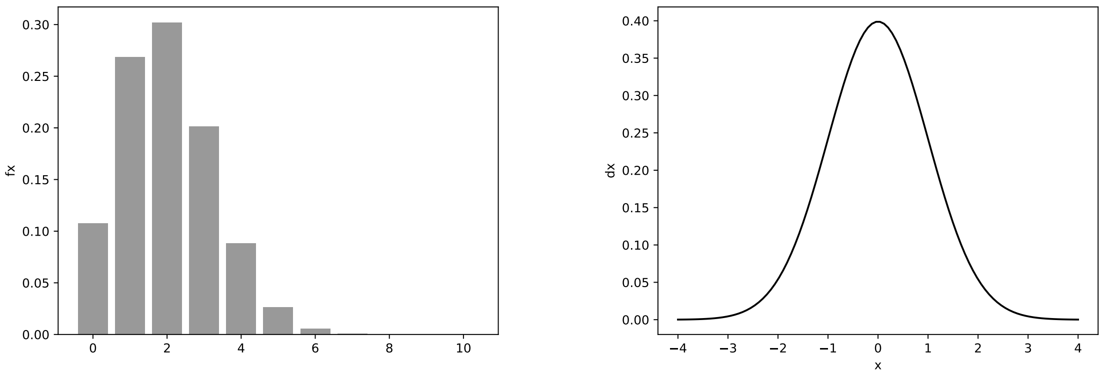
We can also give arrays as one or more arguments to **stats.binom.pmf(**x**,**n**,**p**)** and receive the results as an array. Script 1.30 (PMF-example.py) evaluates the PMF for our example at all possible values for x (0 through 10). It displays a table of the probabilities and creates a bar chart of these probabilities which is shown in Figure 1.15(a). As always: feel encouraged to experiment!

Script 1.30: PMF-example.py

```
import scipy.stats as stats
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# values for x (all between 0 and 10):
x = np.linspace(0, 10, num=11)

# PMF for all these values:
fx = stats.binom.pmf(x, 10, 0.2)

# collect values in DataFrame:
result = pd.DataFrame({'x': x, 'fx': fx})
print(f'result: \n{result}\n')

# plot:
plt.bar(x, fx, color='0.6')
plt.ylabel('x')
plt.ylabel('fx')
plt.savefig('PyGraphs/PMF-example.pdf')
```

[19] see Wooldridge (2019, Equation (B.14))

Figure 1.15. Plots of the PMF and PDF



(a) Binomial PMF (b) Standard normal PDF

Output of Script 1.30: `PMF-example.py`

```
result:
       x           fx
0    0.0  1.073742e-01
1    1.0  2.684355e-01
2    2.0  3.019899e-01
3    3.0  2.013266e-01
4    4.0  8.808038e-02
5    5.0  2.642412e-02
6    6.0  5.505024e-03
7    7.0  7.864320e-04
8    8.0  7.372800e-05
9    9.0  4.096000e-06
10  10.0  1.024000e-07
```

1.6.2. Continuous Distributions

For continuous distributions like the uniform, logistic, exponential, normal, t, χ^2, or F distribution, the probability density functions $f(x)$ are also implemented for direct use in **scipy**. These can for example be used to plot the density functions using the **plot** command (see Section 1.4). Figure 1.15(b) shows the famous bell-shaped PDF of the standard normal distribution and is created by Script 1.31 (`PDF-example.py`).

Script 1.31: **PDF-example.py**

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support of normal density:
x_range = np.linspace(-4, 4, num=100)

# PDF for all these values:
pdf = stats.norm.pdf(x_range)

# plot:
plt.plot(x_range, pdf, linestyle='-', color='black')
plt.xlabel('x')
plt.ylabel('dx')
plt.savefig('PyGraphs/PDF-example.pdf')
```

1.6.3. Cumulative Distribution Function (CDF)

For all distributions, the CDF $F(x) = \mathrm{P}(X \leq x)$ represents the probability that the random variable X takes a value of *at most* x. The probability that X is between two values a and b is $\mathrm{P}(a < X \leq b) = F(b) - F(a)$. We can directly use the **scipy** functions in the second column of Table 1.6 to do these calculations as demonstrated in Script 1.32 (`CDF-example.py`). In our example presented above, the probability that we get 3 or fewer white balls is $F(3)$ using the appropriate CDF of the Binomial distribution. It amounts to 87.9%. The probability that a standard normal random variable takes a value between -1.96 and 1.96 is 95%.

Script 1.32: **CDF-example.py**

```
import scipy.stats as stats

# binomial CDF:
p1 = stats.binom.cdf(3, 10, 0.2)
print(f'p1: {p1}\n')

# normal CDF:
p2 = stats.norm.cdf(1.96) - stats.norm.cdf(-1.96)
print(f'p2: {p2}\n')
```

Output of Script 1.32: **CDF-example.py**

```
p1: 0.8791261184000001

p2: 0.950004209703559
```

Wooldridge, Example B.6: Probabilities for a Normal Random Variable

We assume $X \sim$ Normal$(4, 9)$ and want to calculate $\mathrm{P}(2 < X \leq 6)$ as our first example. We can rewrite the problem so it is stated in terms of a standard normal distribution as shown by Wooldridge (2019): $\mathrm{P}(2 < X \leq 6) = \Phi(\frac{2}{3}) - \Phi(-\frac{2}{3})$. We can also spare ourselves the transformation and work with the non-standard normal distribution directly. Be careful that the third argument in the `scipy` commands for the normal distribution is not the variance $\sigma^2 = 9$ but the standard deviation $\sigma = 3$. The second example calculates $\mathrm{P}(|X| > 2) = \underbrace{1 - \mathrm{P}(X \leq 2)}_{\mathrm{P}(X>2)} + \mathrm{P}(X < -2)$.

Note that we get a slightly different answer in the first example than the one given in Wooldridge (2019) since we're working with the exact $\frac{2}{3}$ instead of the rounded .67.

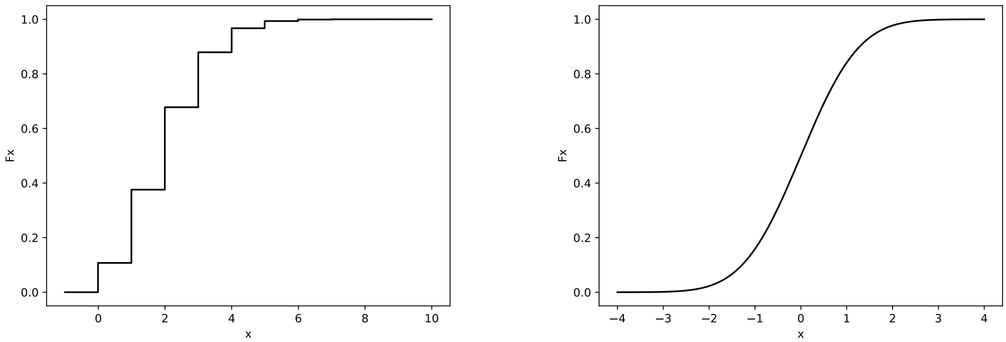
Script 1.33: Example-B-6.py

```
import scipy.stats as stats

# first example using the transformation:
p1_1 = stats.norm.cdf(2 / 3) - stats.norm.cdf(-2 / 3)
print(f'p1_1: {p1_1}\n')

# first example working directly with the distribution of X:
p1_2 = stats.norm.cdf(6, 4, 3) - stats.norm.cdf(2, 4, 3)
print(f'p1_2: {p1_2}\n')

# second example:
p2 = 1 - stats.norm.cdf(2, 4, 3) + stats.norm.cdf(-2, 4, 3)
print(f'p2: {p2}\n')
```

Output of Script 1.33: Example-B-6.py

```
p1_1: 0.4950149249061542

p1_2: 0.4950149249061542

p2: 0.7702575944012563
```

Figure 1.16. Plots of the CDF of Discrete and Continuous RV



(a) Binomial CDF

(b) Standard normal CDF

The graph of the CDF is a step function for discrete distributions. For the urn example, the CDF is shown in Figure 1.16(a). The CDF of a *continuous* distribution is illustrated by the S-shaped CDF of the normal distribution as shown in Figure 1.16(b). Both figures are created by the following code:

Script 1.34: `CDF-figure.py`

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# binomial:
# support of binomial PMF:
x_binom = np.linspace(-1, 10, num=1000)

# PMF for all these values:
cdf_binom = stats.binom.cdf(x_binom, 10, 0.2)

# plot:
plt.step(x_binom, cdf_binom, linestyle='-', color='black')
plt.xlabel('x')
plt.ylabel('Fx')
plt.savefig('PyGraphs/CDF-figure-discrete.pdf')
plt.close()

# normal:
# support of normal density:
x_norm = np.linspace(-4, 4, num=1000)

# PDF for all these values:
cdf_norm = stats.norm.cdf(x_norm)

# plot:
plt.plot(x_norm, cdf_norm, linestyle='-', color='black')
plt.xlabel('x')
plt.ylabel('Fx')
plt.savefig('PyGraphs/CDF-figure-cont.pdf')
```

Quantile function

The q-quantile $x[q]$ of a random variable is the value for which the probability to sample a value $x \leq x[q]$ is just q. These values are important for example for calculating critical values of test statistics.

To give a simple example: Given X is standard normal, the 0.975-quantile is $x[0.975] \approx 1.96$. So the probability to sample a value less or equal to 1.96 is 97.5%:

Script 1.35: `Quantile-example.py`

```
import scipy.stats as stats

q_975 = stats.norm.ppf(0.975)
print(f'q_975: {q_975}\n')
```

Output of Script 1.35: `Quantile-example.py`

```
q_975: 1.959963984540054
```

1.6.4. Random Draws from Probability Distributions

It is easy to simulate random outcomes by taking a sample from a random variable with a given distribution. Strictly speaking, a deterministic machine like a computer can never produce any truly random results and we should instead refer to the generated numbers as *pseudo-random* numbers. But for our purpose, it is enough that the generated samples look, feel and behave like true random numbers and so we are a little sloppy in our terminology here. For a review of sampling and related concepts see Wooldridge (2019, Appendix C.1).

Before we make heavy use of generating random samples in Section 1.9, we introduce the mechanics here. Commands in **`scipy`** to generate a (pseudo-) random sample are constructed by combining the command of the respective distribution (see Table 1.6) and the function name **`rvs`**. We could for example simulate the result of flipping a fair coin 10 times. We draw a sample of size $n = 10$ from a Bernoulli distribution with parameter $p = \frac{1}{2}$. Each of the 10 generated numbers will take the value 1 with probability $p = \frac{1}{2}$ and 0 with probability $1 - p = \frac{1}{2}$. The result behaves the same way as though we had actually flipped a coin and translated heads as 1 and tails as 0 (or vice versa). Here is the code and a sample generated by it:

Script 1.36: `smpl-bernoulli.py`

```
import scipy.stats as stats

sample = stats.bernoulli.rvs(0.5, size=10)
print(f'sample: {sample}\n')
```

Output of Script 1.36: `smpl-bernoulli.py`

```
sample: [1 0 0 1 0 1 1 0 0 1]
```

Translated into the coins, our sample is heads-tails-tails-heads-tails-heads-heads-tails-tails-heads. An obvious advantage of doing this in *Python* rather than with an actual coin is that we can painlessly increase the sample size to 1,000 or 10,000,000. Taking draws from the standard normal distribution is equally simple:

Script 1.37: `smpl-norm.py`

```
import scipy.stats as stats

sample = stats.norm.rvs(size=10)
print(f'sample: {sample}\n')
```

Output of Script 1.37: smpl-norm.py

```
sample: [ 2.1652536   0.63260132  0.20412996 -1.94355999 -0.95095503 -0.2650094
  0.46289967 -1.05426978  0.54156159 -0.95774292]
```

Working with computer-generated random samples creates problems for the reproducibility of the results. If you run the code above, you will get different samples. If we rerun the code, the sample will change again. We can solve this problem by making use of how the random numbers are actually generated which is, as already noted, not involving true randomness. Actually, we will always get the same sequence of numbers if we reset the random number generator to some specific state ("seed"). In *Python*, this is can be done with **numpy**'s function **random.seed(number)**, where **number** is some arbitrary integer that defines the state but has no other meaning. If we set the seed to some arbitrary integer, take a sample, reset the seed to the same state and take another sample, both samples will be the same. Also, if I draw a sample with that seed it will be equal to the sample you draw if we both start from the same seed.

Script 1.38 (Random-Numbers.py) demonstrates the workings of **random.seed**.

Script 1.38: Random-Numbers.py

```
import numpy as np
import scipy.stats as stats

# sample from a standard normal RV with sample size n=5:
sample1 = stats.norm.rvs(size=5)
print(f'sample1: {sample1}\n')

# a different sample from the same distribution:
sample2 = stats.norm.rvs(size=5)
print(f'sample2: {sample2}\n')

# set the seed of the random number generator and take two samples:
np.random.seed(6254137)
sample3 = stats.norm.rvs(size=5)
print(f'sample3: {sample3}\n')

sample4 = stats.norm.rvs(size=5)
print(f'sample4: {sample4}\n')

# reset the seed to the same value to get the same samples again:
np.random.seed(6254137)
sample5 = stats.norm.rvs(size=5)
print(f'sample5: {sample5}\n')

sample6 = stats.norm.rvs(size=5)
print(f'sample6: {sample6}\n')
```

Output of Script 1.38: Random-Numbers.py

```
sample1: [ 0.56038146 -1.41869121  1.74692595  0.4244097   0.67217059]

sample2: [-1.21348357  2.08717118 -0.4821461  -3.22837683  0.44109069]

sample3: [ 1.18545933 -0.261977    0.30894761 -2.23354318  0.17612456]

sample4: [-0.17500741 -1.30835159  0.5036692   0.14991385  0.99957472]

sample5: [ 1.18545933 -0.261977    0.30894761 -2.23354318  0.17612456]

sample6: [-0.17500741 -1.30835159  0.5036692   0.14991385  0.99957472]
```

1.7. Confidence Intervals and Statistical Inference

Wooldridge (2019) provides a concise overview over basic sampling, estimation, and testing. We will touch on some of these issues below.[20]

1.7.1. Confidence Intervals

Confidence intervals (CI) are introduced in Wooldridge (2019, Appendix C.5). They are constructed to cover the true population parameter of interest with a given high probability, e.g. 95%. More clearly: For 95% of all samples, the implied CI includes the population parameter.

CI are easy to compute. For a normal population with unknown mean μ and variance σ^2, the $100(1-\alpha)\%$ confidence interval for μ is given in Wooldridge (2019, Equations C.24 and C.25):

$$\left[\bar{y} - c_{\frac{\alpha}{2}} \cdot se(\bar{y}), \quad \bar{y} + c_{\frac{\alpha}{2}} \cdot se(\bar{y})\right] \tag{1.2}$$

where $\bar{y}$ is the sample average, $se(\bar{y}) = \frac{s}{\sqrt{n}}$ is the standard error of $\bar{y}$ (with s being the sample standard deviation of y), n is the sample size and $c_{\frac{\alpha}{2}}$ the $\left(1-\frac{\alpha}{2}\right)$ quantile of the t_{n-1} distribution. To get the 95% CI ($\alpha = 5\%$), we thus need $c_{0.025}$ which is the 0.975 quantile or 97.5$^{\text{th}}$ percentile.

We already know how to calculate all these ingredients. The way of calculating the CI is used in the solution to Example C.2. In Section 1.9.3, we will calculate confidence intervals in a simulation experiment to help us understand the meaning of confidence intervals.

[20] The stripped-down textbook for Europe and Africa Wooldridge (2014) does not include the discussion of this material.

Wooldridge, Example C.2: Effect of Job Training Grants on Worker Productivity

We are analyzing scrap rates for firms that receive a job training grant in 1988. The scrap rates for 1987 and 1988 are printed in Wooldridge (2019, Table C.3) and are entered manually in the beginning of Script 1.39 (`Example-C-2.py`). We are interested in the change between the years. The calculation of its average as well as the confidence interval are performed precisely as shown above. The resulting CI is the same as the one presented in Wooldridge (2019) except for rounding errors we avoid by working with the exact numbers.

Script 1.39: `Example-C-2.py`

```
import numpy as np
import scipy.stats as stats

# manually enter raw data from Wooldridge, Table C.3:
SR87 = np.array([10, 1, 6, .45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
                 .98, 1, .45, 5.03, 8, 9, 18, .28, 7, 3.97])
SR88 = np.array([3, 1, 5, .5, 1.54, 1.5, .8, 2, .67, 1.17, .51,
                 .5, .61, 6.7, 4, 7, 19, .2, 5, 3.83])

# calculate change:
Change = SR88 - SR87

# ingredients to CI formula:
avgCh = np.mean(Change)
print(f'avgCh: {avgCh}\n')

n = len(Change)
sdCh = np.std(Change, ddof=1)
se = sdCh / np.sqrt(n)
print(f'se: {se}\n')

c = stats.t.ppf(0.975, n - 1)
print(f'c: {c}\n')

# confidence interval:
lowerCI = avgCh - c * se
print(f'lowerCI: {lowerCI}\n')

upperCI = avgCh + c * se
print(f'upperCI: {upperCI}\n')
```

Output of Script 1.39: `Example-C-2.py`

```
avgCh: -1.1544999999999999

se: 0.5367992249386514

c: 2.093024054408263

lowerCI: -2.2780336901843095

upperCI: -0.030966309815690485
```

Wooldridge, Example C.3: Race Discrimination in Hiring

We are looking into race discrimination using the data set AUDIT. The variable **y** represents the difference in hiring rates between black and white applicants with the identical CV. After calculating the average, sample size, standard deviation and the standard error of the sample average, Script 1.40 (Example-C-3.py) calculates the value for the factor c as the 97.5 percentile of the standard normal distribution which is (very close to) 1.96. Finally, the 95% and 99% CI are reported.[21]

Script 1.40: Example-C-3.py

```
import wooldridge as woo
import numpy as np
import scipy.stats as stats

audit = woo.dataWoo('audit')
y = audit['y']

# ingredients to CI formula:
avgy = np.mean(y)
n = len(y)
sdy = np.std(y, ddof=1)
se = sdy / np.sqrt(n)
c95 = stats.norm.ppf(0.975)
c99 = stats.norm.ppf(0.995)

# 95% confidence interval:
lowerCI95 = avgy - c95 * se
print(f'lowerCI95: {lowerCI95}\n')

upperCI95 = avgy + c95 * se
print(f'upperCI95: {upperCI95}\n')

# 99% confidence interval:
lowerCI99 = avgy - c99 * se
print(f'lowerCI99: {lowerCI99}\n')

upperCI99 = avgy + c99 * se
print(f'upperCI99: {upperCI99}\n')
```

Output of Script 1.40: Example-C-3.py

```
lowerCI95: -0.19363006093502752

upperCI95: -0.07193010504007621

lowerCI99: -0.21275050976771243

upperCI99: -0.052809656207391295
```

[21] Note that Wooldridge (2019) has a typo in the discussion of this example, therefore the numbers don't quite match for the 95% CI.

1.7.2. t Tests

Hypothesis tests are covered in Wooldridge (2019, Appendix C.6). The t test statistic for testing a hypothesis about the mean μ of a normally distributed random variable Y is shown in Equation C.35. Given the null hypothesis $H_0 : \mu = \mu_0$,

$$t = \frac{\bar{y} - \mu_0}{se(\bar{y})}. \tag{1.3}$$

We already know how to calculate the ingredients from Section 1.7.1 and show to use them to perform a t test in Script 1.42 (`Example-C-5.py`). We also compare the result to the output of the **`scipy`** function **`ttest_1samp`**, which performs an automated t test.

The critical value for this test statistic depends on whether the test is one-sided or two-sided. The value needed for a two-sided test $c_{\frac{\alpha}{2}}$ was already calculated for the CI, the other values can be generated accordingly. The values for different degrees of freedom $n - 1$ and significance levels α are listed in Wooldridge (2019, Table G.2). Script 1.41 (`Critical-Values-t.py`) demonstrates how we can calculate our own table of critical values for the example of 19 degrees of freedom.

Script 1.41: `Critical-Values-t.py`

```
import numpy as np
import pandas as pd
import scipy.stats as stats

# degrees of freedom = n-1:
df = 19

# significance levels:
alpha_one_tailed = np.array([0.1, 0.05, 0.025, 0.01, 0.005, .001])
alpha_two_tailed = alpha_one_tailed * 2

# critical values & table:
CV = stats.t.ppf(1 - alpha_one_tailed, df)
table = pd.DataFrame({'alpha_one_tailed': alpha_one_tailed,
                      'alpha_two_tailed': alpha_two_tailed, 'CV': CV})
print(f'table: \n{table}\n')
```

Output of Script 1.41: `Critical-Values-t.py`

```
table:
   alpha_one_tailed  alpha_two_tailed        CV
0             0.100             0.200  1.327728
1             0.050             0.100  1.729133
2             0.025             0.050  2.093024
3             0.010             0.020  2.539483
4             0.005             0.010  2.860935
5             0.001             0.002  3.579400
```

Wooldridge, Example C.5: Race Discrimination in Hiring

We continue Example C.3 in Script 1.42 (`Example-C-5.py`) and perform a one-sided t test of the null hypothesis $H_0 : \mu = 0$ against $H_1 : \mu < 0$ for the same sample. As the output shows, the t test statistic is equal to -4.27. This is much smaller than the negative of the critical value for any sensible significance level. Therefore, we reject $H_0 : \mu = 0$ for this one-sided test, see Wooldridge (2019, Equation C.38).

Script 1.42: `Example-C-5.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import scipy.stats as stats

audit = woo.dataWoo('audit')
y = audit['y']

# automated calculation of t statistic for H0 (mu=0):
test_auto = stats.ttest_1samp(y, popmean=0)
t_auto = test_auto.statistic  # access test statistic
p_auto = test_auto.pvalue  # access two-sided p value
print(f't_auto: {t_auto}\n')
print(f'p_auto/2: {p_auto / 2}\n')

# manual calculation of t statistic for H0 (mu=0):
avgy = np.mean(y)
n = len(y)
sdy = np.std(y, ddof=1)
se = sdy / np.sqrt(n)
t_manual = avgy / se
print(f't_manual: {t_manual}\n')

# critical values for t distribution with n-1=240 d.f.:
alpha_one_tailed = np.array([0.1, 0.05, 0.025, 0.01, 0.005, .001])
CV = stats.t.ppf(1 - alpha_one_tailed, 240)
table = pd.DataFrame({'alpha_one_tailed': alpha_one_tailed, 'CV': CV})
print(f'table: \n{table}\n')
```

Output of Script 1.42: `Example-C-5.py`

```
t_auto: -4.276816348963646

p_auto/2: 1.369270781112999e-05

t_manual: -4.276816348963646

table:
   alpha_one_tailed        CV
0             0.100  1.285089
1             0.050  1.651227
2             0.025  1.969898
3             0.010  2.341985
4             0.005  2.596469
5             0.001  3.124536
```

1.7.3. p Values

The p value for a test is the probability that (under the assumptions needed to derive the distribution of the test statistic) a different random sample would produce the same or an even more extreme value of the test statistic.[22] The advantage of using p values for statistical testing is that they are convenient to use. Instead of having to compare the test statistic with critical values which are implied by the significance level α, we directly compare p with α. For two-sided t tests, the formula for the p value is given in Wooldridge (2019, Equation C.42):

$$p = 2 \cdot \mathrm{P}(T_{n-1} > |t|) = 2 \cdot \left(1 - F_{t_{n-1}}(|t|)\right) \ , \tag{1.4}$$

where $F_{t_{n-1}}(\cdot)$ is the CDF of the t_{n-1} distribution which we know how to calculate from Table 1.6. Similarly, a one-sided test rejects the null hypothesis only if the value of the estimate is "too high" or "too low" relative to the null hypothesis. The p values for these types of tests are:

$$p = \begin{cases} \mathrm{P}(T_{n-1} < t) = F_{t_{n-1}}(t) & \text{for } H_1 : \mu < \mu_0 \\ \mathrm{P}(T_{n-1} > t) = 1 - F_{t_{n-1}}(t) & \text{for } H_1 : \mu > \mu_0 \end{cases} \tag{1.5}$$

Since we are working on a computer program that knows the CDF of the t distribution, calculating p values is straightforward as demonstrated in Script 1.43 (`Example-C-6.py`). Maybe you noticed that the **`scipy`** function **`ttest_1samp`** in Script 1.42 (`Example-C-5.py`) also calculates the p value, but be aware that this function is always based on two-sided t tests.

[22]The p value is often misinterpreted. It is for example *not* the probability that the null hypothesis is true. For a discussion, see for example `https://www.nature.com/news/scientific-method-statistical-errors-1.14700`.

Wooldridge, Example C.6: Effect of Job Training Grants on Worker Productivity

We continue from Example C.2 in Script 1.43 (`Example-C-6.py`). We test $H_0 : \mu = 0$ against $H_1 : \mu < 0$. The t statistic is -2.15. The formula for the p value for this one-sided test is given in Wooldridge (2019, Equation C.41). As can be seen in the output of Script 1.43 (`Example-C-6.py`), its value (using exact values of t) is around 0.022. If you want to use the **scipy** function **ttest_1samp**, you have to divide the p value by 2, because we are dealing with a one-sided test.

Script 1.43: Example-C-6.py

```
import numpy as np
import scipy.stats as stats

# manually enter raw data from Wooldridge, Table C.3:
SR87 = np.array([10, 1, 6, .45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
                 .98, 1, .45, 5.03, 8, 9, 18, .28, 7, 3.97])
SR88 = np.array([3, 1, 5, .5, 1.54, 1.5, .8, 2, .67, 1.17, .51,
                 .5, .61, 6.7, 4, 7, 19, .2, 5, 3.83])
Change = SR88 - SR87

# automated calculation of t statistic for H0 (mu=0):
test_auto = stats.ttest_1samp(Change, popmean=0)
t_auto = test_auto.statistic
p_auto = test_auto.pvalue
print(f't_auto: {t_auto}\n')
print(f'p_auto/2: {p_auto / 2}\n')

# manual calculation of t statistic for H0 (mu=0):
avgCh = np.mean(Change)
n = len(Change)
sdCh = np.std(Change, ddof=1)
se = sdCh / np.sqrt(n)
t_manual = avgCh / se
print(f't_manual: {t_manual}\n')

# manual calculation of p value for H0 (mu=0):
p_manual = stats.t.cdf(t_manual, n - 1)
print(f'p_manual: {p_manual}\n')
```

Output of Script 1.43: Example-C-6.py

```
t_auto: -2.150711003973493

p_auto/2: 0.02229062646839212

t_manual: -2.150711003973493

p_manual: 0.02229062646839212
```

Wooldridge, Example C.7: Race Discrimination in Hiring

In Example C.5, we found the t statistic for $H_0 : \mu = 0$ against $H_1 : \mu < 0$ to be $t = -4.276816$. The corresponding p value is calculated in Script 1.44 (`Example-C-7.py`). The number **`1.369271e-05`** is the scientific notation for $1.369271 \cdot 10^{-5} = .00001369271$. So the p value is around 0.0014% which is much smaller than any reasonable significance level. By construction, we draw the same conclusion as when we compare the t statistic with the critical value in Example C.5. We reject the null hypothesis that there is no discrimination.

Script 1.44: `Example-C-7.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import scipy.stats as stats

audit = woo.dataWoo('audit')
y = audit['y']

# automated calculation of t statistic for H0 (mu=0):
test_auto = stats.ttest_1samp(y, popmean=0)
t_auto = test_auto.statistic
p_auto = test_auto.pvalue
print(f't_auto: {t_auto}\n')
print(f'p_auto/2: {p_auto/2}\n')

# manual calculation of t statistic for H0 (mu=0):
avgy = np.mean(y)
n = len(y)
sdy = np.std(y, ddof=1)
se = sdy / np.sqrt(n)
t_manual = avgy / se
print(f't_manual: {t_manual}\n')

# manual calculation of p value for H0 (mu=0):
p_manual = stats.t.cdf(t_manual, n - 1)
print(f'p_manual: {p_manual}\n')
```

Output of Script 1.44: `Example-C-7.py`

```
t_auto: -4.276816348963646

p_auto/2: 1.369270781112999e-05

t_manual: -4.276816348963646

p_manual: 1.369270781112999e-05
```

1.8. Advanced *Python*

The material covered in this section is not necessary for most of what we will do in the remainder of this book, so it can be skipped. However, it is important enough to justify an own section in this chapter. We will only scratch the surface, though. For more details, you will have to look somewhere else, for example Downey (2015).

1.8.1. Conditional Execution

We might want some parts of our code to be executed only under certain conditions. Like most other programming languages, this can be achieved with an **if else** statement. Note that in *Python*, the parts to be conditionally executed are identified by indenting them with the same amount of whitespaces. Editors like Spyder will assist us with this. This gives the following syntax:

```
if condition:
    expression1
else:
    expression2
```

The **condition** has to be a single logical value (**True** or **False**). If it is **True**, then **expression1** is executed, otherwise **expression2** which can also be omitted. A simple example would be

```
if p <= 0.05:
    print("reject H0!")
else:
    print("don't reject H0!")
```

Depending on the value of the numeric scalar **p**, the respective test decision is printed.

1.8.2. Loops

For repeatedly executing an expression, different kinds of loops are available. In this book, we will use them for Monte Carlo analyses introduced in Section 1.9. For our purposes, the **for** loop is well suited. The correct syntax (including the indenting) is:

```
for x in sequence:
    [some commands]
```

The loop variable **x** will take the value of each element of **sequence**, one after another. For each of these elements, **[some commands]** are executed. Often, **sequence** will be a list like **[1, 2, 3]**.

A nonsense example which combines **for** loops with an **if** statement is given in Script 1.45 (`Adv-Loops.py`). The reader is encouraged to first form expectations about the output this will generate and then compare them with the actual results.

Script 1.45: Adv-Loops.py

```
seq = [1, 2, 3, 4, 5, 6]
for i in seq:
    if i < 4:
        print(i ** 3)
    else:
        print(i ** 2)
```

Output of Script 1.45: `Adv-Loops.py`

```
1
8
27
16
25
```

Instead of iterating over a sequence you can also iterate over an index of a sequence and use the index to reference other objects. The "pythonian" way of generating such a sequence of indices uses the function **`range`**, which is demonstrated in Script 1.46 (`Adv-Loops2.py`) by doing the same as Script 1.45 (`Adv-Loops.py`).

Script 1.46: `Adv-Loops2.py`

```
seq = [1, 2, 3, 4, 5, 6]
for i in range(len(seq)):
    if seq[i] < 4:
        print(seq[i] ** 3)
    else:
        print(seq[i] ** 2)
```

Output of Script 1.46: `Adv-Loops2.py`

```
1
8
27
16
25
```

If you want to execute expressions as long as a given condition is **`True`**, *Python* offers the **`while`** loop, but we will not present it here.

1.8.3. Functions

A function is a block of code that is executed if the function is called. You can provide additional data to the function in form of arguments. There are many pre-defined functions and modules provide even more functions to expand the capabilities of *Python*. We're now ready to define our own little function.

The command **`def newfunc(arg1, arg2, ...)`** defines a new function **`newfunc`** which accepts the arguments **`arg1`**, **`arg2`**,.... The function definition follows in arbitrarily many lines of indented code. Within the function definition, the command **`return stuff`** means that **`stuff`** is to be returned as a result of the function call. For example, we can define the function **`mysqrt`** that expects one argument internally named **`x`**. Script 1.47 (`Adv-Functions.py`) shows how to define and call the function **`mysqrt`**.

Script 1.47: Adv-Functions.py

```
# define function:
def mysqrt(x):
    if x >= 0:
        result = x ** 0.5
    else:
        result = 'You fool!'
    return result


# call function and save result:
result1 = mysqrt(4)
print(f'result1: {result1}\n')

result2 = mysqrt(-1.5)
print(f'result2: {result2}\n')
```

Output of Script 1.47: Adv-Functions.py

```
result1: 2.0

result2: You fool!
```

Note that you can pass arguments by name, by position, or a combination of both. Passing arguments by position is used in the examples in Script 1.47 (`Adv-Functions.py`), because it is clear that any provided input to the function must be the argument **`x`**. In the case of multiple arguments the order of provided inputs matters: the first piece of input is related to the first argument in the function definition, the second piece of input is related to the second argument in the function definition, etc. .

As an alternative you could also execute **`mysqrt(x=4)`**, which is meant by providing arguments by name. In the case of multiple arguments the order of provided named inputs does not matter.

1.8.4. Object Orientation

You might have wondered where all the data types we have used so far (e.g. lists or **`numpy`** arrays) come from. In an object oriented language like *Python* almost everything is an object and you can easily define your own objects. You can think of an object as an elegant way of structuring your code: objects store a certain type of data and contain functions that can be applied to this data. In the context of objects, functions are called methods and data are saved in local variables of an object (also called attributes).

To work with objects that are suited for your purposes you have to define what kind of data they can store and what you want to do with them. The blueprint of such an object is called a "class". If you make use of this class to store data and work with them, you are dealing with an "instance" or "object" of this class. Of course, one class can be used to create multiple instances of this class. To use local variables or methods of an object, you follow the familiar syntax **`objectname.variablename`** or **`objectname.methodname(arg1, arg2, ...)`**.

Let's discuss an easy example: you want to build a database in *Python* for your local bike shop. The first thing you should do is to define a class **`bike`**, where you collect properties of a bike. This could be the price, size, color or anything else that might be important and you define them as local variables. Let's say the color of a bike must often be changed before it can be sold, so you add a method **`changeColor(newColor)`** to the class definition. The moment the first bike needs to be stored in the database, you create an instance of this class, say **`firstNewBike`**. Within this instance, all defined properties are set (also called "initializing"). If a bike with the exact same properties arrives a few hours later and needs to be stored in the database, you create a new instance, so every object has it's own identity. If you want to change the color of the first instance to green you call **firstNewBike.changeColor('green')**.

In this book, there are only very few cases where we cannot rely on predefined classes provided by *Python* or a given module. However, a basic understanding of object orientation helps you to understand how certain commands work. In Script 1.48 (`Adv-ObjOr.py`), for example, the class **list** is used to create an object named **a**. The author of this class also added a method **count** which is only applied on data stored within **a**. There are also methods like **sort**, which changes data stored in an object.

Script 1.48: Adv-ObjOr.py

```
# use the predefined class 'list' to create an object:
a = [2, 6, 3, 6]

# access a local variable (to find out what kind of object we are dealing with):
check = type(a).__name__
print(f'check: {check}\n')

# make use of a method (how many 6 are in a?):
count_six = a.count(6)
print(f'count_six: {count_six}\n')

# use another method (sort data in a):
a.sort()
print(f'a: {a}\n')
```

Output of Script 1.48: Adv-ObjOr.py

```
check: list

count_six: 2

a: [2, 3, 6, 6]
```

We are now ready to define our own class. Script 1.49 (`Adv-ObjOr2.py`) demonstrates how to write your version of the **dot** method in **numpy**. Local variables are always initiated by the **__init__** method in *Python*.

Note that the presented approach of nested loops is not the most computationally efficient way to implement matrix multiplication in *Python*. But it helps to demonstrate the definition of a class and gives another example for using **for** loops.

Script 1.49: Adv-ObjOr2.py

```
import numpy as np

# multiply these two matrices:
a = np.array([[3, 6, 1], [2, 7, 4]])
b = np.array([[1, 8, 6], [3, 5, 8], [1, 1, 2]])

# the numpy way:
result_np = a.dot(b)
print(f'result_np: \n{result_np}\n')

# or, do it yourself by defining a class:
class myMatrices:
    def __init__(self, A, B):
        self.A = A
        self.B = B

    def mult(self):
        N = self.A.shape[0]  # number of rows in A
        K = self.B.shape[1]  # number of cols in B
        out = np.empty((N, K))  # initialize output
        for i in range(N):
            for j in range(K):
                out[i, j] = sum(self.A[i, :] * self.B[:, j])
        return out


# create an object:
test = myMatrices(a, b)

# access local variables:
print(f'test.A: \n{test.A}\n')
print(f'test.B: \n{test.B}\n')

# use object method:
result_own = test.mult()
print(f'result_own: \n{result_own}\n')
```

Output of Script 1.49: Adv-ObjOr2.py

```
result_np:
[[22 55 68]
 [27 55 76]]

test.A:
[[3 6 1]
 [2 7 4]]

test.B:
[[1 8 6]
 [3 5 8]
 [1 1 2]]

result_own:
[[22. 55. 68.]
 [27. 55. 76.]]
```

You can easily build on other classes by using a concept called inheritance. Let's assume we want to extend our class **myMatrices** by a method that calculates the total amount of elements in the matrix product. Subclass **myMatNew** in Script 1.50 (`Adv-ObjOr3.py`) inherits the properties and methods from **myMatrices** and adds the method **getTotalElem**, so by using **myMatNew** you can do everything you can do with **myMatrices** and calculating the total amount of elements in the matrix product.

Script 1.50: Adv-ObjOr3.py

```
import numpy as np

# multiply these two matrices:
a = np.array([[3, 6, 1], [2, 7, 4]])
b = np.array([[1, 8, 6], [3, 5, 8], [1, 1, 2]])


# define your own class:
class myMatrices:
    def __init__(self, A, B):
        self.A = A
        self.B = B

    def mult(self):
        N = self.A.shape[0]  # number of rows in A
        K = self.B.shape[1]  # number of cols in B
        out = np.empty((N, K))  # initialize output
        for i in range(N):
            for j in range(K):
                out[i, j] = sum(self.A[i, :] * self.B[:, j])
        return out


# define a subclass:
class myMatNew(myMatrices):
    def getTotalElem(self):
        N = self.A.shape[0]  # number of rows in A
        K = self.B.shape[1]  # number of cols in B
        return N * K


# create an object of the subclass:
test = myMatNew(a, b)

# use a method of myMatrices:
result_own = test.mult()
print(f'result_own: \n{result_own}\n')

# use a method of myMatNew:
totalElem = test.getTotalElem()
print(f'totalElem: {totalElem}\n')
```

Output of Script 1.50: Adv-ObjOr3.py

```
result_own:
[[22. 55. 68.]
 [27. 55. 76.]]

totalElem: 6
```

Be aware that we only covered the most important concepts of object orientated programming that we will encounter in this book.

1.8.5. Outlook

While this section is called "Advanced *Python*", we have admittedly only scratched the surface of semi-advanced topics. One topic we defer to Chapter 19 is how *Python* can automatically create formatted reports and publication-ready documents.

Another advanced topic is the optimization of computational speed. So an example of seriously advanced topics for the real *Python* geek is to use parallel computing to speed up computations.

Since real *Python* geeks are not the target audience of this book, we will stop to even mention more intimidating possibilities and focus on implementing the most important econometric methods in the most straightforward and pragmatic way.

1.9. Monte Carlo Simulation

Appendix C.2 of Wooldridge (2019) contains a brief introduction to estimators and their properties.[23] In real-world applications, we typically have a data set corresponding to a random sample from a well-defined population. We don't know the population parameters and use the sample to estimate them.

When we generate a sample using a computer program as we have introduced in Section 1.6.4, we know the population parameters since we had to choose them when making the random draws. We could apply the same estimators to this artificial sample to estimate the population parameters. The tasks would be: (1) Select a population distribution and its parameters. (2) Generate a sample from this distribution. (3) Use the sample to estimate the population parameters.

If this sounds a little insane to you: Don't worry, that would be a healthy first reaction. We obtain a noisy estimate of something we know precisely. But this sort of analysis does in fact make sense. Because we estimate something we actually know, we are able to study the behavior of our estimator very well.

In this book, we mainly use this approach for illustrative and didactic reasons. In state-of-the-art research, it is widely used since it often provides the only way to learn about important features of estimators and statistical tests. A name frequently given to these sorts of analyses is Monte Carlo simulation in reference to the "gambling" involved in generating random samples.

1.9.1. Finite Sample Properties of Estimators

Let's look at a simple example and simulate a situation in which we want to estimate the mean μ of a normally distributed random variable

$$Y \sim \text{Normal}(\mu, \sigma^2) \tag{1.6}$$

using a sample of a given size n. The obvious estimator for the population mean would be the sample average $\bar{Y}$. But what properties does this estimator have? The informed reader immediately knows that the sampling distribution of $\bar{Y}$ is

$$\bar{Y} \sim \text{Normal}\left(\mu, \frac{\sigma^2}{n}\right) \tag{1.7}$$

Simulation provides a way to verify this claim.

[23]The stripped-down textbook for Europe and Africa Wooldridge (2014) does not include this either.

Script 1.51 (`Simulate-Estimate.py`) shows a simulation experiment in action: We set the seed to ensure reproducibility and draw a sample of size $n = 100$ from the population distribution (with the population parameters $\mu = 10$ and $\sigma = 2$).[24] Then, we calculate the sample average as an estimate of μ. We see results for three different samples.

Script 1.51: `Simulate-Estimate.py`

```
import numpy as np
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# set sample size:
n = 100

# draw a sample given the population parameters:
sample1 = stats.norm.rvs(10, 2, size=n)

# estimate the population mean with the sample average:
estimate1 = np.mean(sample1)
print(f'estimate1: {estimate1}\n')

# draw a different sample and estimate again:
sample2 = stats.norm.rvs(10, 2, size=n)
estimate2 = np.mean(sample2)
print(f'estimate2: {estimate2}\n')

# draw a third sample and estimate again:
sample3 = stats.norm.rvs(10, 2, size=n)
estimate3 = np.mean(sample3)
print(f'estimate3: {estimate3}\n')
```

Output of Script 1.51: `Simulate-Estimate.py`

```
estimate1: 9.573602656614304

estimate2: 10.24798129790092

estimate3: 9.96021755398913
```

All sample means $\bar{Y}$ are around the true mean $\mu = 10$ which is consistent with our presumption formulated in Equation 1.7. It is also not surprising that we don't get the exact population parameter – that's the nature of the sampling noise. According to Equation 1.7, the results are expected to have a variance of $\frac{\sigma^2}{n} = 0.04$. Three samples of this kind are insufficient to draw strong conclusions regarding the validity of Equation 1.7. Good Monte Carlo simulation studies should use as many samples as possible.

In Section 1.8.2, we introduced **`for`** loops. While they are not the most powerful technique available in *Python* to implement a Monte Carlo study, we will stick to them since they are quite transparent and straightforward. The code shown in Script 1.52 (`Simulation-Repeated.py`) uses a **`for`** loop to draw 10 000 samples of size $n = 100$ and calculates the sample average for all of them. After setting the random seed, the empty array **`ybar`** of size 10 000 is initialized using the **`np.empty`** command. We will replace these empty array values with the estimates one after another in the loop. In each of these replications $j = 0, 1, 2, \ldots, 9\,999$, a sample is drawn, its average calculated and stored

[24]See Section 1.6.4 for the basics of random number generation.

in position number **j** of **ybar**. In this way, we end up with a list of 10 000 estimates from different samples. The Script `Simulation-Repeated.py` does not generate any output.

Script 1.52: Simulation-Repeated.py

```
import numpy as np
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# set sample size:
n = 100

# initialize ybar to an array of length r=10000 to later store results:
r = 10000
ybar = np.empty(r)

# repeat r times:
for j in range(r):
    # draw a sample and store the sample mean in pos. j=0,1,... of ybar:
    sample = stats.norm.rvs(10, 2, size=n)
    ybar[j] = np.mean(sample)
```

Script 1.53 (`Simulation-Repeated-Results.py`) analyses these 10 000 estimates. Here, we just discuss the output, but you find the complete code in the appendix. The average of **ybar** is very close to the presumption $\mu = 10$ from Equation 1.7. Also the simulated sampling variance is close to the theoretical result $\frac{\sigma^2}{n} = 0.04$. Note that the degrees of freedom are adjusted with **ddof=1** in **np.var()** to compute the unbiased estimate of the variance. Finally, the estimated density (using a kernel density estimate from the module **statsmodels**) is compared to the theoretical normal distribution. The result is shown in Figure 1.17. The two lines are almost indistinguishable except for the area close to the mode (where the kernel density estimator is known to have problems).

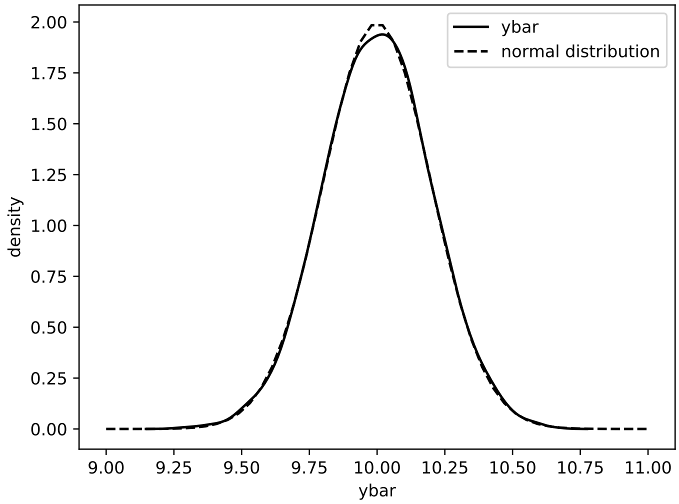
Output of Script 1.53: Simulation-Repeated-Results.py

```
ybar[0:19]:
[ 9.57360266 10.2479813   9.96021755  9.67635967  9.82261605  9.6270579
 10.02979223 10.15400282 10.28812728  9.69935763 10.41950951 10.07993562
  9.75764232 10.10504699  9.99813607  9.92113688  9.55713599 10.01404669
 10.25550724]

np.mean(ybar): 10.00082418067469

np.var(ybar, ddof=1): 0.03989666893894718
```

To summarize, the simulation results confirm the theoretical results in Equation 1.7. Mean, variance and density are very close and it seems likely that the remaining tiny differences are due to the fact that we "only" used 10 000 samples.

Remember: for most advanced estimators, such simulations are the only way to study some of their features since it is impossible to derive theoretical results of interest. For us, the simple example hopefully clarified the approach of Monte Carlo simulations and the meaning of the sampling distribution and prepared us for other interesting simulation exercises.

Figure 1.17. Simulated and Theoretical Density of $\bar{Y}$



1.9.2. Asymptotic Properties of Estimators

Asymptotic analyses are concerned with large samples and with the behavior of estimators and other statistics as the sample size n increases without bound. For a discussion of these topics, see Wooldridge (2019, Appendix C.3). According to the **law of large numbers**, the sample average $\bar{Y}$ in the above example converges in probability to the population mean μ as $n \to \infty$. In (infinitely) large samples, this implies that $\mathrm{E}(\bar{Y}) \to \mu$ and $\mathrm{Var}(\bar{Y}) \to 0$.

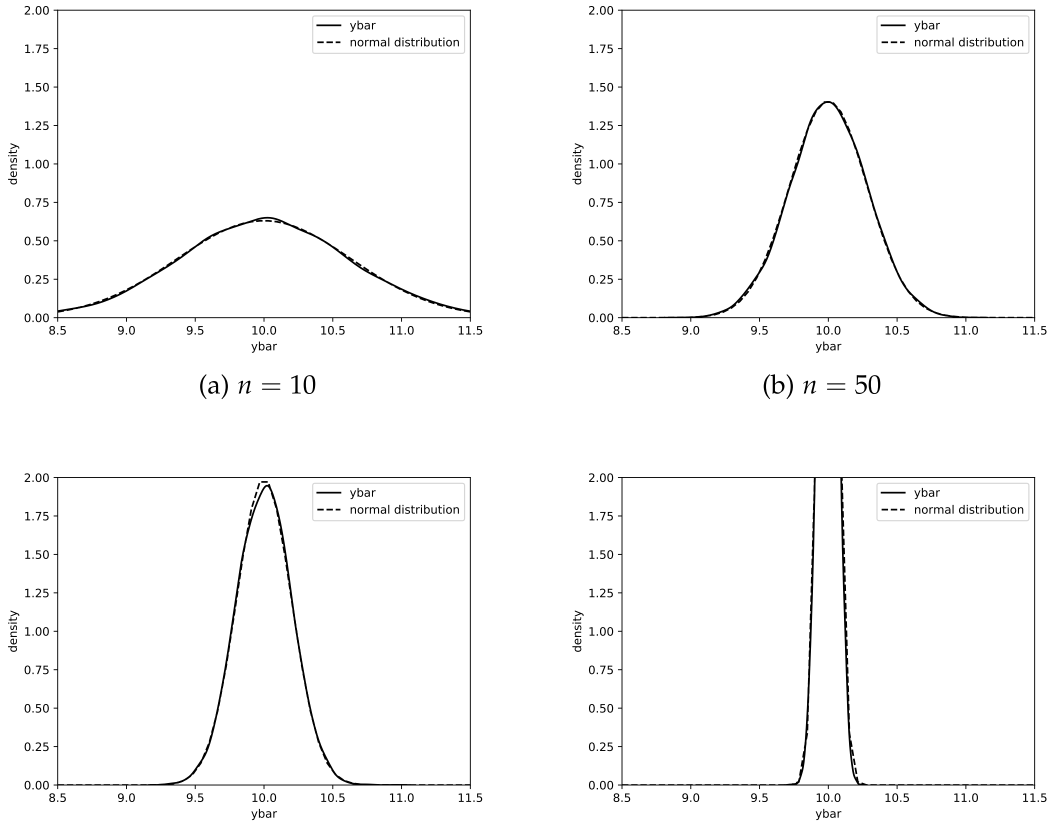
With Monte Carlo simulation, we have a tool to see how this works out in our example. We just have to change the sample size in the code line **`n = 100`** in Script 1.52 (`Simulation-Repeated.py`) to a different number and rerun the simulation code. Results for $n = 10, 50, 100$, and 1000 are presented in Figure 1.18. Apparently, the variance of $\bar{Y}$ does in fact decrease. The graph of the density for $n = 1000$ is already very narrow and high indicating a small variance. Of course, we cannot actually increase n to infinity without crashing our computer, but it appears plausible that the density will eventually collapse into one vertical line corresponding to $\mathrm{Var}(\bar{Y}) \to 0$ as $n \to \infty$.

In our example for the simulations, the random variable Y was normally distributed, therefore the sample average $\bar{Y}$ was also normal for any sample size. This can also be confirmed in Figure 1.18 where the respective normal densities were added to the graphs as dashed lines. The **central limit theorem** (CLT) claims that as $n \to \infty$, the sample mean $\bar{Y}$ of a random sample will eventually *always* be normally distributed, no matter what the distribution of Y is (unless it is very weird with an infinite variance). This is called convergence in distribution.

Let's check this with a very non-normal distribution, the χ^2 distribution with one degree of freedom. Its density is depicted in Figure 1.19.[25] It looks very different from our familiar bell-shaped normal density. The only line we have to change in the simulation code in Script 1.52 (`Simulation-Repeated.py`) is **`sample = stats.norm.rvs(10, 2, size=n)`** which we

[25] A motivated reader will already have figured out that this graph was generated by **`chi2.pdf(x, df)`** from the **`scipy`** module.

Figure 1.18. Density of $\bar{Y}$ with Different Sample Sizes



(a) $n = 10$ (b) $n = 50$ (c) $n = 100$ (d) $n = 1000$

have to replace with **`sample = stats.chi2.rvs(1, size=n)`** according to Table 1.6. Figure 1.20 shows the simulated densities for different sample sizes and compares them to the normal distribution with the same mean $\mu = 1$ and standard deviation $\frac{s}{\sqrt{n}} = \sqrt{\frac{2}{n}}$. Note that the scales of the axes now differ between the sub-figures in order to provide a better impression of the shape of the densities. The effect of a decreasing variance works here in exactly the same way as with the normal population.

Not surprisingly, the distribution of $\bar{Y}$ is very different from a normal one in small samples like $n = 2$. With increasing sample size, the CLT works its magic and the distribution gets closer to the normal bell-shape. For $n = 10000$, the densities hardly differ at all so it's easy to imagine that they will eventually be the same as $n \to \infty$.

1.9.3. Simulation of Confidence Intervals and t Tests

In addition to repeatedly estimating population parameters, we can also calculate confidence intervals and conduct tests on the simulated samples. Here, we present a somewhat advanced simulation

Figure 1.19. Density of the χ^2 Distribution with 1 d.f.

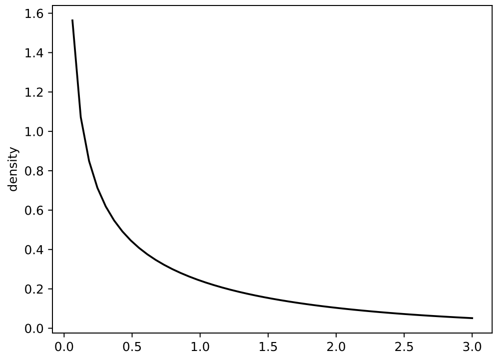
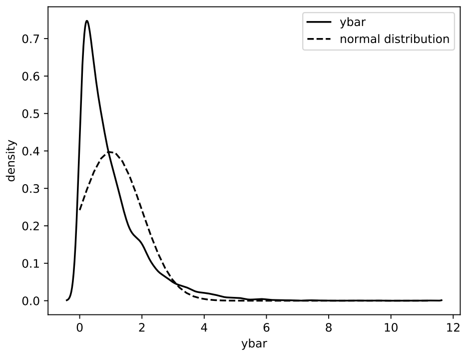
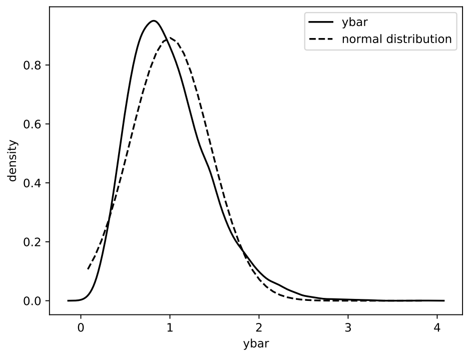


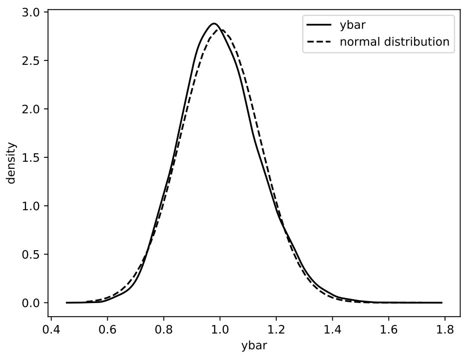
Figure 1.20. Density of $\bar{Y}$ with Different Sample Sizes: χ^2 Distribution



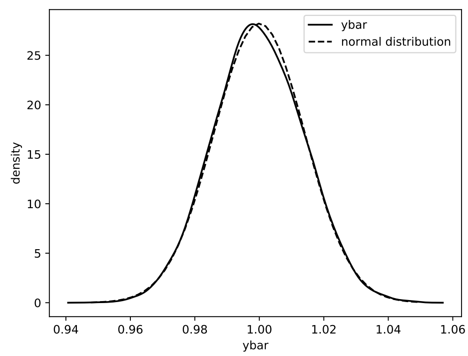
(a) $n = 2$



(b) $n = 10$



(c) $n = 100$



(d) $n = 10000$

routine. The payoff of going through this material is that it might substantially improve our understanding of the workings of statistical inference.

We start from the same example as in Section 1.9.1: In the population, $Y \sim \text{Normal}(10, 4)$. We draw 10 000 samples of size $n = 100$ from this population. For each of the samples we calculate

- The 95% confidence interval and store the limits in **`CIlower`** and **`CIupper`**.
- The p value for the two-sided test of the correct null hypothesis $H_0 : \mu = 10 \Rightarrow$ array **`pvalue1`**
- The p value for the two-sided test of the *incorrect* null hypothesis $H_0 : \mu = 9.5 \Rightarrow$ array **`pvalue2`**

Finally, we calculate the array **`reject1`** and **`reject2`** with logical items that are **`True`** if we reject the respective null hypothesis at $\alpha = 5\%$, i.e. if **`pvalue1`** or **`pvalue2`** are smaller than 0.05, respectively. Script 1.55 (`Simulation-Inference.py`) shows the *Python* code for these simulations and a frequency table for the results **`reject1`** and **`reject2`**.

If theory and the implementation in *Python* are accurate, the probability to reject a correct null hypothesis (i.e. to make a Type I error) should be equal to the chosen significance level α. In our simulation, we reject the correct hypothesis in 504 of the 10 000 samples, which amounts to 5.04%.

The probability to reject a false hypothesis is called the power of a test. It depends on many things like the sample size and "how bad" the error of H_0 is, i.e. how far away μ_0 is from the true μ. Theory just tells us that the power is larger than α. In our simulation, the wrong null $H_0 : \mu = 9.5$ is rejected in 69.9% of the samples. The reader is strongly encouraged to tinker with the simulation code to verify the theoretical results that this power increases if μ_0 moves away from 10 and if the sample size n increases.

Figure 1.21 graphically presents the 95% CI for the first 100 simulated samples.[26] Each horizontal line represents one CI. In these first 100 samples, the true null was rejected in 4 cases. This fact means that for those four samples the CI does not cover $\mu_0 = 10$, see Wooldridge (2019, Appendix C.6) on the relationship between CI and tests. These four cases are drawn in black in the left part of the figure, whereas the others are gray.

The t-test rejects the false null hypothesis $H_0 : \mu = 9.5$ in 72 of the first 100 samples. Their CIs do not cover 9.5 and are drawn in black in the right part of Figure 1.21.

[26]For the sake of completeness, the code for generating these graphs is shown in Appendix IV, Script 1.54 (`Simulation-Inference-Figure.py`), but most readers will probably not find it important to look at it at this point.

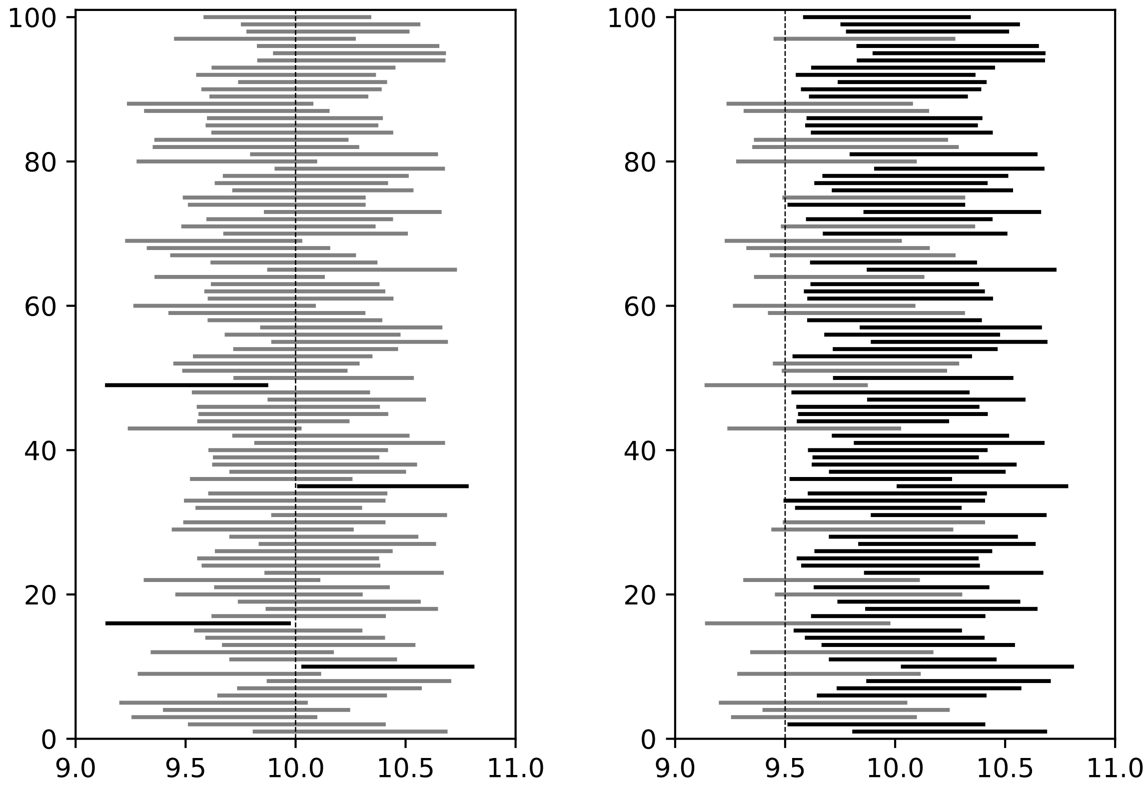
Script 1.55: Simulation-Inference.py

```
import numpy as np
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# set sample size and MC simulations:
r = 10000
n = 100

# initialize arrays to later store results:
CIlower = np.empty(r)
CIupper = np.empty(r)
pvalue1 = np.empty(r)
pvalue2 = np.empty(r)

# repeat r times:
for j in range(r):
    # draw a sample:
    sample = stats.norm.rvs(10, 2, size=n)
    sample_mean = np.mean(sample)
    sample_sd = np.std(sample, ddof=1)

    # test the (correct) null hypothesis mu=10:
    testres1 = stats.ttest_1samp(sample, popmean=10)
    pvalue1[j] = testres1.pvalue
    cv = stats.t.ppf(0.975, df=n - 1)
    CIlower[j] = sample_mean - cv * sample_sd / np.sqrt(n)
    CIupper[j] = sample_mean + cv * sample_sd / np.sqrt(n)

    # test the (incorrect) null hypothesis mu=9.5 & store the p value:
    testres2 = stats.ttest_1samp(sample, popmean=9.5)
    pvalue2[j] = testres2.pvalue

# test results as logical value:
reject1 = pvalue1 <= 0.05
count1_true = np.count_nonzero(reject1)  # counts true
count1_false = r - count1_true
print(f'count1_true: {count1_true}\n')
print(f'count1_false: {count1_false}\n')

reject2 = pvalue2 <= 0.05
count2_true = np.count_nonzero(reject2)
count2_false = r - count2_true
print(f'count2_true: {count2_true}\n')
print(f'count2_false: {count2_false}\n')
```

Output of Script 1.55: Simulation-Inference.py

```
count1_true: 504

count1_false: 9496

count2_true: 6990

count2_false: 3010
```

Figure 1.21. Simulation Results: First 100 Confidence Intervals



(a) correct H_0 (b) incorrect H_0

Part I.

Regression Analysis with Cross-Sectional Data

2. The Simple Regression Model

2.1. Simple OLS Regression

We are concerned with estimating the population parameters β_0 and β_1 of the simple linear regression model

$$y = \beta_0 + \beta_1 x + u \tag{2.1}$$

from a random sample of y and x. According to Wooldridge (2019, Section 2.2), the ordinary least squares (OLS) estimators are

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x} \tag{2.2}$$

$$\hat{\beta}_1 = \frac{\text{Cov}(x,y)}{\text{Var}(x)}. \tag{2.3}$$

Based on these estimated parameters, the OLS regression line is

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x. \tag{2.4}$$

For a given sample, we just need to calculate the four statistics $\bar{y}$, $\bar{x}$, $\text{Cov}(x,y)$, and $\text{Var}(x)$ and plug them into these equations. We already know how to make these calculations in *Python*, see Section 1.5. Let's do it!

Wooldridge, Example 2.3: CEO Salary and Return on Equity

We are using the data set `CEOSAL1` we already analyzed in Section 1.5. We consider the simple regression model

$$\texttt{salary} = \beta_0 + \beta_1 \texttt{roe} + u$$

where `salary` is the salary of a CEO in thousand dollars and `roe` is the return on investment in percent. In Script 2.1 (`Example-2-3.py`), we first load the modules and the data set. We also calculate the four statistics we need for Equations 2.2 and 2.3 so we can reproduce the OLS formulas by hand. Finally, the parameter estimates are calculated.
So the OLS regression line is

$$\widehat{\texttt{salary}} = 963.1913 + 18.50119 \cdot \texttt{roe}.$$

Script 2.1: Example-2-3.py

```
import wooldridge as woo
import numpy as np

ceosal1 = woo.dataWoo('ceosal1')
x = ceosal1['roe']
y = ceosal1['salary']

# ingredients to the OLS formulas:
cov_xy = np.cov(x, y)[1, 0]  # access 2. row and 1. column of covariance matrix
var_x = np.var(x, ddof=1)
x_bar = np.mean(x)
y_bar = np.mean(y)

# manual calculation of OLS coefficients:
b1 = cov_xy / var_x
b0 = y_bar - b1 * x_bar
print(f'b1: {b1}\n')
print(f'b0: {b0}\n')
```

Output of Script 2.1: Example-2-3.py

```
b1: 18.501186345214926

b0: 963.1913364725579
```

While calculating OLS coefficients using this pedestrian approach is straightforward, there is a more convenient way to do it. Given the importance of OLS regression, it is not surprising that many *Python* modules have a specialized command to do the calculations automatically. In the following chapters, we will often use the module **statsmodels** to apply linear regression and other econometric methods.[1] More information about the module is provided by Seabold and Perktold (2010). When working with **statsmodels**, the first line of code often is:

```
import statsmodels.formula.api as smf
```

If the data frame **sample** contains the values of the dependent variable in column **y** and those of the regressor in the column **x**, we can calculate the OLS coefficients as

```
reg = smf.ols(formula='y ~ x', data=sample)
results = reg.fit()
```

The first argument **y ~ x** is called a **formula**. Essentially, it means that we want to model a left-hand-side variable **y** to be explained by a right-hand-side variable **x** in a linear fashion. We will discuss more general model formulae in Section 6.1. In the second line of code, the actual calculation of OLS coefficients and many other results are performed by calling the method **fit**.

Finally, all kind of results are assigned to the variable **results**. The name could of course be anything, for example **yummy_chocolate_chip_cookies**, but choosing telling variable names makes our life easier. As already mentioned, the referenced object does not only include the OLS coefficients, but also information on the data source and much more we will get to know and use later on.

[1]The module **statsmodels** is part of the Anaconda distribution.

Wooldridge, Example 2.3: CEO Salary and Return on Equity (*cont'ed*)

In Script 2.2 (`Example-2-3-2.py`), we repeat the analysis we have already done manually. Besides the import of the data, there are only a few lines of code. The output shows how to access both estimated parameters with **`results.params`**: $\hat{\beta}_0$ is labeled **`Intercept`** and $\hat{\beta}_1$ is labeled with the name of the explanatory variable **`roe`**. The values are the same we already calculated except for different rounding in the output.

Script 2.2: `Example-2-3-2.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Output of Script 2.2: `Example-2-3-2.py`

```
b:
Intercept    963.191336
roe           18.501186
dtype: float64
```

From now on, we will rely on the built-in routine in **`statsmodels`** instead of doing the calculations manually. It is not only more convenient for calculating the coefficients, but also for further analyses as we will see soon.

Given the results from a regression, plotting the regression line is straightforward. As we have already seen in Section 1.4.3, the command **`plot`** can add points to a graph. In this case, we simply supply the regressor **`roe`** and the predicted values (available under **`results.fittedvalues`**) and connect them by a line.

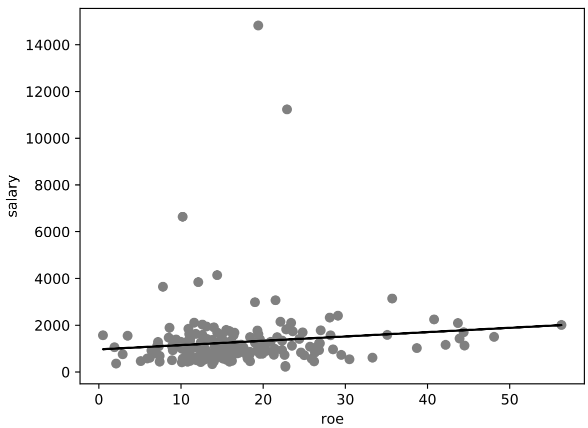
Wooldridge, Example 2.3: CEO Salary and Return on Equity (*cont'ed*)

Script 2.3 (`Example-2-3-3.py`) demonstrates how to store the regression results in a variable **`results`** and then use its included fitted values as an argument to **`plot`** to add the regression line to the scatter plot. It generates Figure 2.1.

Script 2.3: `Example-2-3-3.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# OLS regression:
reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()

# scatter plot and fitted values:
plt.plot('roe', 'salary', data=ceosal1, color='grey', marker='o', linestyle='')
plt.plot(ceosal1['roe'], results.fittedvalues, color='black', linestyle='-')
plt.ylabel('salary')
plt.xlabel('roe')
plt.savefig('PyGraphs/Example-2-3-3.pdf')
```

Figure 2.1. OLS Regression Line for Example 2-3



Wooldridge, Example 2.4: Wage and Education

We are using the data set `WAGE1`. We are interested in studying the relation between education and wage, and our regression model is

$$\texttt{wage} = \beta_0 + \beta_1 \texttt{education} + u.$$

In Script 2.4 (`Example-2-4.py`), we analyze the data and find that the OLS regression line is

$$\widehat{\texttt{wage}} = -0.90 + 0.54 \cdot \texttt{education}.$$

One additional year of education is associated with an increase of the typical wage by about 54 cents an hour.

Script 2.4: `Example-2-4.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='wage ~ educ', data=wage1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Output of Script 2.4: `Example-2-4.py`

```
b:
Intercept   -0.904852
educ         0.541359
dtype: float64
```

Wooldridge, Example 2.5: Voting Outcomes and Campaign Expenditures

The data set `VOTE1` contains information on campaign expenditures (`shareA` = share of campaign spending in %) and election outcomes (`voteA` = share of vote in %). The regression model

$$\texttt{voteA} = \beta_0 + \beta_1 \texttt{shareA} + u$$

is estimated in Script 2.5 (`Example-2-5.py`). The OLS regression line turns out to be

$$\widehat{\texttt{voteA}} = 26.81 + 0.464 \cdot \texttt{shareA}.$$

The scatter plot with the regression line generated in the code is shown in Figure 2.2.

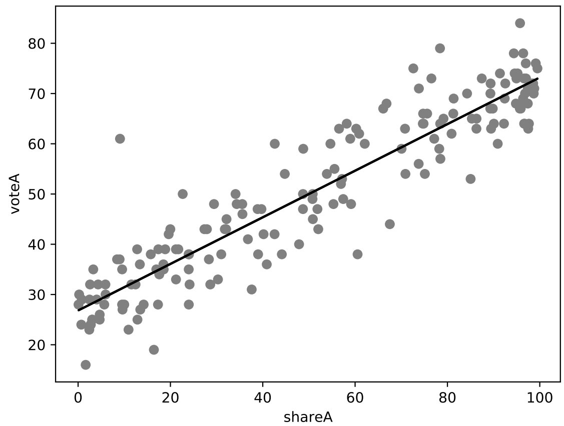
Script 2.5: `Example-2-5.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

vote1 = woo.dataWoo('vote1')

# OLS regression:
reg = smf.ols(formula='voteA ~ shareA', data=vote1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')

# scatter plot and fitted values:
plt.plot('shareA', 'voteA', data=vote1, color='grey', marker='o', linestyle='')
plt.plot(vote1['shareA'], results.fittedvalues, color='black', linestyle='-')
plt.ylabel('voteA')
plt.xlabel('shareA')
plt.savefig('PyGraphs/Example-2-5.pdf')
```

Output of Script 2.5: `Example-2-5.py`

```
b:
Intercept    26.812214
shareA        0.463827
dtype: float64
```

Figure 2.2. OLS Regression Line for Example 2-5



2.2. Coefficients, Fitted Values, and Residuals

The object returned by the method **fit** contains all relevant information on the regression. Since this information is distributed across multiple object local variables of the returned object, we can access them with the syntax **resultobject.local_var_name**. After defining the regression results object **results** in Script 2.2 (`Example-2-3-2.py`), we can access the OLS coefficients with

```
results.params
```

The coefficient object has names attached to its elements. The name of the intercept parameter $\hat{\beta}_0$ is **Intercept** and the name of the slope parameter $\hat{\beta}_1$ is the variable name of the regressor **x**. In this way, we can access the parameters separately by using either the position (0 or 1) or the name as an index to the coefficients object. For example, in Script 2.2 (`Example-2-3-2.py`) you can access intercept and slope parameter by

```
b[0] # intercept
b['roe'] # slope parameter
```

Given these parameter estimates, calculating the predicted values $\hat{y}_i$ and residuals $\hat{u}_i$ for each observation $i = 1, ..., n$ is easy:

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 \cdot x_i \tag{2.5}$$

$$\hat{u}_i = y_i - \hat{y}_i \tag{2.6}$$

If the values of the dependent and independent variables are stored in a data frame **sample** as **y** and **x**, respectively, we can estimate the model and do the calculations of these equations for all observations jointly using the code

```
reg = smf.ols(formula='y ~ x', data=sample)
results = reg.fit()
b = results.params
y_hat = b[0] + b[1] * sample['x']
u_hat = sample['y'] - y_hat
```

We can also use a more black-box approach which will give exactly the same results using the precalculated variables **fittedvalues** and **resid** on the regression results object:

```
reg = smf.ols(formula='y ~ x', data=sample)
results = reg.fit()
y_hat = results.fittedvalues
u_hat = results.resid
```

Wooldridge, Example 2.6: CEO Salary and Return on Equity

We extend the regression example on the return on equity of a firm and the salary of its CEO in Script 2.6 (`Example-2-6.py`). After the OLS regression, we calculate fitted values and residuals. A table similar to Wooldridge (2019, Table 2.2) is generated displaying the values for the first 15 observations.

Script 2.6: `Example-2-6.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

# OLS regression:
reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()

# obtain predicted values and residuals:
salary_hat = results.fittedvalues
u_hat = results.resid

# Wooldridge, Table 2.2:
table = pd.DataFrame({'roe': ceosal1['roe'],
                      'salary': ceosal1['salary'],
                      'salary_hat': salary_hat,
                      'u_hat': u_hat})
print(f'table.head(15): \n{table.head(15)}\n')
```

Output of Script 2.6: `Example-2-6.py`

```
table.head(15):
          roe  salary   salary_hat       u_hat
0   14.100000    1095  1224.058071 -129.058071
1   10.900000    1001  1164.854261 -163.854261
2   23.500000    1122  1397.969216 -275.969216
3    5.900000     578  1072.348338 -494.348338
4   13.800000    1368  1218.507712  149.492288
5   20.000000    1145  1333.215063 -188.215063
6   16.400000    1078  1266.610785 -188.610785
7   16.299999    1094  1264.760660 -170.760660
8   10.500000    1237  1157.453793   79.546207
9   26.299999     833  1449.772523 -616.772523
10  25.900000     567  1442.372056 -875.372056
11  26.799999     933  1459.023116 -526.023116
12  14.800000    1339  1237.008898  101.991102
13  22.299999     937  1375.767778 -438.767778
14  56.299999    2011  2004.808114    6.191886
```

Wooldridge (2019, Section 2.3) presents and discusses three properties of OLS statistics which we will confirm for an example.

$$\sum_{i=1}^{n} \hat{u}_i = 0 \quad \Rightarrow \quad \bar{\hat{u}}_i = 0 \tag{2.7}$$

$$\sum_{i=1}^{n} x_i \hat{u}_i = 0 \quad \Rightarrow \quad \text{Cov}(x_i, \hat{u}_i) = 0 \tag{2.8}$$

$$\bar{y} = \hat{\beta}_0 + \hat{\beta}_1 \cdot \bar{x} \tag{2.9}$$

Wooldridge, Example 2.7: Wage and Education

We already know the regression results when we regress wage on education from Example 2.4. In Script 2.7 (`Example-2-7.py`), we calculate fitted values and residuals to confirm the three properties from Equations 2.7 through 2.9. Note that *Python* does all calculations in "double precision" implying that it is accurate for at least 15 significant digits. The output that checks the first property shows that the average residual is `-7.564713e-15` which in scientific notation means $-7.564713 \cdot 10^{-15} = -0.000000000000007564713$. The reason it is not exactly equal to 0 is a rounding error in the 16$^{\text{th}}$ digit. The same holds for the second property: The covariance between the regressor and the residual is zero except for minimal rounding error. Note that running Script 2.7 (`Example-2-7.py`) will give you the same accurate digits, but the digits with rounding error will differ. The third property is also confirmed: If we plug the average value of the regressor into the regression line formula, we get the average value of the dependent variable.

Script 2.7: `Example-2-7.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')
reg = smf.ols(formula='wage ~ educ', data=wage1)
results = reg.fit()

# obtain coefficients, predicted values and residuals:
b = results.params
wage_hat = results.fittedvalues
u_hat = results.resid

# confirm property (1):
u_hat_mean = np.mean(u_hat)
print(f'u_hat_mean: {u_hat_mean}\n')

# confirm property (2):
educ_u_cov = np.cov(wage1['educ'], u_hat)[1, 0]
print(f'educ_u_cov: {educ_u_cov}\n')

# confirm property (3):
educ_mean = np.mean(wage1['educ'])
wage_pred = b[0] + b[1] * educ_mean
print(f'wage_pred: {wage_pred}\n')

wage_mean = np.mean(wage1['wage'])
print(f'wage_mean: {wage_mean}\n')
```

Output of Script 2.7: `Example-2-7.py`

```
u_hat_mean: -7.564713536609432e-15

educ_u_cov: -2.3211062701496606e-15

wage_pred: 5.896102674787043

wage_mean: 5.896102674787035
```

2.3. Goodness of Fit

The total sum of squares (SST), explained sum of squares (SSE) and residual sum of squares (SSR) can be written as

$$\text{SST} = \sum_{i=1}^{n}(y_i - \overline{y})^2 = (n-1)\cdot \text{Var}(y) \tag{2.10}$$

$$\text{SSE} = \sum_{i=1}^{n}(\hat{y}_i - \overline{y})^2 = (n-1)\cdot \text{Var}(\hat{y}) \tag{2.11}$$

$$\text{SSR} = \sum_{i=1}^{n}(\hat{u}_i - 0)^2 = (n-1)\cdot \text{Var}(\hat{u}) \tag{2.12}$$

where $\text{Var}(x)$ is the sample variance $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$.

Wooldridge (2019, Equation 2.38) defines the coefficient of determination in terms of these terms. Because $(n-1)$ cancels out, it can be equivalently written as

$$R^2 = \frac{\text{Var}(\hat{y})}{\text{Var}(y)} = 1 - \frac{\text{Var}(\hat{u})}{\text{Var}(y)}. \tag{2.13}$$

Wooldridge, Example 2.8: CEO Salary and Return on Equity

In the regression already studied in Example 2.6, the coefficient of determination is 0.0132. This is calculated in the two ways of Equation 2.13 in Script 2.8 (`Example-2-8.py`). In addition, it is calculated as the squared correlation coefficient of y and $\hat{y}$. Not surprisingly, all versions of these calculations produce the same result (they are not exactly equal to each other because of the rounding error in the 16$^{\text{th}}$ digit).

Script 2.8: **Example-2-8.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

# OLS regression:
reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()

# calculate predicted values & residuals:
sal_hat = results.fittedvalues
u_hat = results.resid

# calculate R^2 in three different ways:
sal = ceosal1['salary']
R2_a = np.var(sal_hat, ddof=1) / np.var(sal, ddof=1)
R2_b = 1 - np.var(u_hat, ddof=1) / np.var(sal, ddof=1)
R2_c = np.corrcoef(sal, sal_hat)[1, 0] ** 2

print(f'R2_a: {R2_a}\n')
print(f'R2_b: {R2_b}\n')
print(f'R2_c: {R2_c}\n')
```

Output of Script 2.8: **Example-2-8.py**

```
R2_a: 0.013188624081034115

R2_b: 0.01318862408103405

R2_c: 0.013188624081034089
```

Many interesting results for a regression can be displayed by calling the method **summary**. You call this method on the object returned by the method **fit** as demonstrated in Script 2.9 (`Example-2-9.py`). The output will display

- A block of general information about the regression model. It contains also other information about the estimation of which only R^2 is of interest to us so far. It is reported as **R-squared**.
- A coefficient table. So far, we only discussed the OLS coefficients shown in the first column. The next columns will be introduced below.
- A block of diagnostics regarding the residuals. We will discuss some of them later.

When we are only interested in the coefficients and their significance, we will often switch to a more compact presentation of results. This is demonstrated with the object **table** in Script 2.9 (`Example-2-9.py`).

Wooldridge, Example 2.9: Voting Outcomes and Campaign Expenditures

We already know the OLS coefficients to be $\hat{\beta}_0 = 26.8125$ and $\hat{\beta}_1 = 0.4638$ in the voting example (Script 2.5 (`Example-2-5.py`)). These values are again found in the output of Script 2.9 (`Example-2-9.py`). The coefficient of determination is reported as **R-squared** to be $R^2 = 0.856$. Reassuringly, we get the same numbers as with the pedestrian calculations.

Script 2.9: Example-2-9.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

vote1 = woo.dataWoo('vote1')

# OLS regression:
reg = smf.ols(formula='voteA ~ shareA', data=vote1)
results = reg.fit()

# print results using summary:
print(f'results.summary(): \n{results.summary()}\n')

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 2.9: Example-2-9.py

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                  voteA   R-squared:                       0.856
Model:                            OLS   Adj. R-squared:                  0.855
Method:                 Least Squares   F-statistic:                     1018.
Date:                Thu, 23 Apr 2020   Prob (F-statistic):           6.63e-74
Time:                        08:20:09   Log-Likelihood:                -565.20
No. Observations:                 173   AIC:                             1134.
Df Residuals:                     171   BIC:                             1141.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     26.8122      0.887     30.221      0.000      25.061      28.564
shareA         0.4638      0.015     31.901      0.000       0.435       0.493
==============================================================================
Omnibus:                       20.747   Durbin-Watson:                   1.826
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               44.613
Skew:                           0.525   Prob(JB):                     2.05e-10
Kurtosis:                       5.255   Cond. No.                         112.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly

table:
                 b      se        t  pval
Intercept  26.8122  0.8872  30.2207   0.0
shareA      0.4638  0.0145  31.9008   0.0
```

2.4. Nonlinearities

For the estimation of logarithmic or semi-logarithmic models, the respective formula can be directly entered into the specification of **`smf.ols(...)`** as demonstrated in Examples 2.10 and 2.11. For the interpretation as percentage effects and elasticities, see Wooldridge (2019, Section 2.4).

Wooldridge, Example 2.10: Wage and Education

Compared to Example 2.7, we simply change the command for the estimation to account for a logarithmic specification as shown in Script 2.10 (`Example-2-10.py`). The semi-logarithmic specification implies that wages are higher by about 8.3% for individuals with an additional year of education.

Script 2.10: `Example-2-10.py`

```
import numpy as np
import wooldridge as woo
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

# estimate log-level model:
reg = smf.ols(formula='np.log(wage) ~ educ', data=wage1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Output of Script 2.10: `Example-2-10.py`

```
b:
Intercept    0.583773
educ         0.082744
dtype: float64
```

Wooldridge, Example 2.11: CEO Salary and Firm Sales

We study the relationship between the sales of a firm and the salary of its CEO using a log-log specification. The results are shown in Script 2.11 (`Example-2-11.py`). If the sales increase by 1%, the salary of the CEO tends to increase by 0.257%.

Script 2.11: `Example-2-11.py`

```
import numpy as np
import wooldridge as woo
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

# estimate log-log model:
reg = smf.ols(formula='np.log(salary) ~ np.log(sales)', data=ceosal1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Output of Script 2.11: `Example-2-11.py`

```
b:
Intercept         4.821996
np.log(sales)     0.256672
dtype: float64
```

2.5. Regression through the Origin and Regression on a Constant

Wooldridge (2019, Section 2.6) discusses models without an intercept. This implies that the regression line is forced to go through the origin. In *Python*, we can suppress the constant which is otherwise implicitly added to a formula by specifying

```
smf.ols('y ~ 0 + x', data=sample)
```

instead of **`smf.ols('y ~ x', data=sample)`**. The result is a model which only has a slope parameter.

Another topic discussed in this section is a linear regression model without a slope parameter, i.e. with a constant only. In this case, the estimated constant will be the sample average of the dependent variable. This can be implemented in *Python* using the code

```
smf.ols('y ~ 1', data=sample)
```

Both special kinds of regressions are implemented in Script 2.12 (`SLR-Origin-Const.py`) for the example of the CEO salary and ROE we already analyzed in Example 2.8 and others. The resulting regression lines are plotted in Figure 2.3 which was generated using the last lines of code shown in the output.

Script 2.12: `SLR-Origin-Const.py`

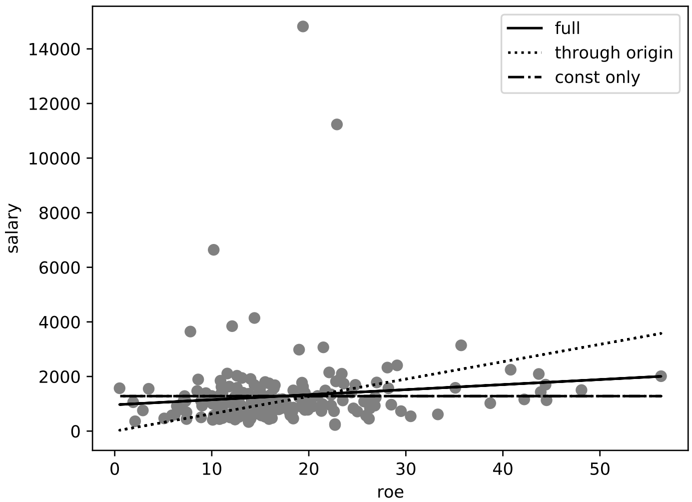
```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# usual OLS regression:
reg1 = smf.ols(formula='salary ~ roe', data=ceosal1)
results1 = reg1.fit()
b_1 = results1.params
print(f'b_1: \n{b_1}\n')

# regression without intercept (through origin):
reg2 = smf.ols(formula='salary ~ 0 + roe', data=ceosal1)
results2 = reg2.fit()
b_2 = results2.params
print(f'b_2: \n{b_2}\n')

# regression without slope (on a constant):
reg3 = smf.ols(formula='salary ~ 1', data=ceosal1)
results3 = reg3.fit()
```

```
b_3 = results3.params
print(f'b_3: \n{b_3}\n')

# average y:
sal_mean = np.mean(ceosal1['salary'])
print(f'sal_mean: {sal_mean}\n')

# scatter plot and fitted values:
plt.plot('roe', 'salary', data=ceosal1, color='grey', marker='o',
         linestyle='', label='')
plt.plot(ceosal1['roe'], results1.fittedvalues, color='black',
         linestyle='-', label='full')
plt.plot(ceosal1['roe'], results2.fittedvalues, color='black',
         linestyle=':', label='through origin')
plt.plot(ceosal1['roe'], results3.fittedvalues, color='black',
         linestyle='-.', label='const only')
plt.ylabel('salary')
plt.xlabel('roe')
plt.legend()
plt.savefig('PyGraphs/SLR-Origin-Const.pdf')
```

Output of Script 2.12: `SLR-Origin-Const.py`

```
b_1:
Intercept    963.191336
roe           18.501186
dtype: float64

b_2:
roe    63.537955
dtype: float64

b_3:
Intercept    1281.119617
dtype: float64

sal_mean: 1281.1196172248804
```

Figure 2.3. Regression through the Origin and on a Constant



2.6. Expected Values, Variances, and Standard Errors

Wooldridge (2019) discusses the role of five assumptions under which the OLS parameter estimators have desirable properties. In short form they are

- **SLR.1**: Linear population regression function: $y = \beta_0 + \beta_1 x + u$
- **SLR.2**: Random sampling of x and y from the population
- **SLR.3**: Variation in the sample values $x_1, ..., x_n$
- **SLR.4**: Zero conditional mean: $\text{E}(u|x) = 0$
- **SLR.5**: Homoscedasticity: $\text{Var}(u|x) = \sigma^2$

Based on those, Wooldridge (2019) shows in Section 2.5:

- **Theorem 2.1**: Under **SLR.1 – SLR.4**, OLS parameter estimators are unbiased.
- **Theorem 2.2**: Under **SLR.1 – SLR.5**, OLS parameter estimators have a specific sampling variance.

Because the formulas for the sampling variance involve the variance of the error term, we also have to estimate it using the unbiased estimator

$$\hat{\sigma}^2 = \frac{1}{n-2} \cdot \sum_{i=1}^{n} \hat{u}_i^2 = \frac{n-1}{n-2} \cdot \text{Var}(\hat{u}_i), \tag{2.14}$$

where $Var(\hat{u}_i) = \frac{1}{n-1} \cdot \sum_{i=1}^{n} \hat{u}_i^2$ is the usual sample variance. We have to use the degrees-of-freedom adjustment to account for the fact that we estimated the two parameters $\hat{\beta}_0$ and $\hat{\beta}_1$ for constructing the residuals. Its square root $\hat{\sigma} = \sqrt{\hat{\sigma}^2}$ is called **standard error of the regression (SER)** by Wooldridge (2019).

The **standard errors (SE) of the estimators** are

$$\text{se}(\hat{\beta}_0) = \sqrt{\frac{\hat{\sigma}^2 \overline{x^2}}{\sum_{i=1}^{n}(x-\overline{x})^2}} = \frac{1}{\sqrt{n-1}} \cdot \frac{\hat{\sigma}}{\text{sd}(x)} \cdot \sqrt{\overline{x^2}} \tag{2.15}$$

$$\text{se}(\hat{\beta}_1) = \sqrt{\frac{\hat{\sigma}^2}{\sum_{i=1}^{n}(x-\overline{x})^2}} = \frac{1}{\sqrt{n-1}} \cdot \frac{\hat{\sigma}}{\text{sd}(x)} \tag{2.16}$$

where $\text{sd}(x)$ is the sample standard deviation $\sqrt{\frac{1}{n-1} \cdot \sum_{i=1}^{n}(x_i - \overline{x})^2}$.

In *Python*, we can obviously do the calculations of Equations 2.15 through 2.16 explicitly. But the output of the **summary** command for linear regression results, which we discovered in Section 2.3, already contains the results. We use the following example to calculate the results in both ways to open the black box of the canned routine and convince ourselves that from now on we can rely on it.

Wooldridge, Example 2.12: Student Math Performance and the School Lunch Program

Using the data set MEAP93, we regress a math performance score of schools on the share of students eligible for a federally funded lunch program. Wooldridge (2019) uses this example to demonstrate the importance of assumption SLR.4 and warns us against interpreting the regression results in a causal way. Here, we merely use the example to demonstrate the calculation of standard errors.

Script 2.13 (Example-2-12.py) first calculates the SER manually using the fact that the residuals $\hat{u}$ are available as **results.resid**. Then, the SE of the parameters are calculated according to Equations 2.15 and 2.16, where the regressor is addressed as the variable in the data frame **meap93['lnchprg']**. Finally, we see the output of the **summary** method. The SE of the parameters are reported in the second column of the regression table, next to the parameter estimates. We will look at the other columns in Chapter 4. All values are exactly the same as the manual results.

Script 2.13: Example-2-12.py

```
import numpy as np
import wooldridge as woo
import statsmodels.formula.api as smf

meap93 = woo.dataWoo('meap93')

# estimate the model and save the results as "results":
reg = smf.ols(formula='math10 ~ lnchprg', data=meap93)
results = reg.fit()

# number of obs.:
n = results.nobs

# SER:
u_hat_var = np.var(results.resid, ddof=1)
SER = np.sqrt(u_hat_var) * np.sqrt((n - 1) / (n - 2))
print(f'SER: {SER}\n')

# SE of b0 & b1, respectively:
lnchprg_sq_mean = np.mean(meap93['lnchprg'] ** 2)
lnchprg_var = np.var(meap93['lnchprg'], ddof=1)
b1_se = SER / (np.sqrt(lnchprg_var)
               * np.sqrt(n - 1)) * np.sqrt(lnchprg_sq_mean)
b0_se = SER / (np.sqrt(lnchprg_var) * np.sqrt(n - 1))
```

```
print(f'b1_se: {b1_se}\n')
print(f'b0_se: {b0_se}\n')

# automatic calculations:
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 2.13: `Example-2-12.py`

```
SER: 9.565938459482759

b1_se: 0.9975823856755018

b0_se: 0.034839334258369624

results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                 math10   R-squared:                       0.171
Model:                            OLS   Adj. R-squared:                  0.169
Method:                 Least Squares   F-statistic:                     83.77
Date:                Thu, 23 Apr 2020   Prob (F-statistic):           2.75e-18
Time:                        08:20:16   Log-Likelihood:                -1499.3
No. Observations:                 408   AIC:                             3003.
Df Residuals:                     406   BIC:                             3011.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     32.1427      0.998     32.221      0.000      30.182      34.104
lnchprg       -0.3189      0.035     -9.152      0.000      -0.387      -0.250
==============================================================================
Omnibus:                       61.162   Durbin-Watson:                   1.908
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              105.062
Skew:                           0.886   Prob(JB):                     1.53e-23
Kurtosis:                       4.743   Cond. No.                         60.4
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

2.7. Monte Carlo Simulations

In this section, we use Monte Carlo simulation experiments to revisit many of the topics covered in this chapter. It can be skipped but can help quite a bit to grasp the concepts of estimators, estimates, unbiasedness, the sampling variance of the estimators, and the consequences of violated assumptions. Remember that the concept of Monte Carlo simulations was introduced in Section 1.9.

2.7.1. One Sample

In Section 1.9, we used simulation experiments to analyze the features of a simple mean estimator. We also discussed the sampling from a given distribution, the random seed and simple examples. We can use exactly the same strategy to analyze OLS parameter estimators.

Script 2.14 (SLR-Sim-Sample.py) shows how to draw a sample which is consistent with Assumptions SLR.1 through SLR.5. We simulate a sample of size $n = 1000$ with population parameters $\beta_0 = 1$ and $\beta_1 = 0.5$. We set the standard deviation of the error term u to $\sigma = 2$. Obviously, these parameters can be freely chosen and every reader is strongly encouraged to play around.

Script 2.14: SLR-Sim-Sample.py

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# set sample size:
n = 1000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# draw a sample of size n:
x = stats.norm.rvs(4, 1, size=n)
u = stats.norm.rvs(0, su, size=n)
y = beta0 + beta1 * x + u
df = pd.DataFrame({'y': y, 'x': x})

# estimate parameters by OLS:
reg = smf.ols(formula='y ~ x', data=df)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')

# features of the sample for the variance formula:
x_sq_mean = np.mean(x ** 2)
print(f'x_sq_mean: {x_sq_mean}\n')
x_var = np.sum((x - np.mean(x)) ** 2)
print(f'x_var: {x_var}\n')

# graph:
x_range = np.linspace(0, 8, num=100)
plt.ylim([-2, 10])
```

```
plt.plot(x, y, color='lightgrey', marker='o', linestyle='')
plt.plot(x_range, beta0 + beta1 * x_range, color='black',
         linestyle='-', linewidth=2, label='pop. regr. fct.')
plt.plot(x_range, b[0] + b[1] * x_range, color='grey',
         linestyle='-', linewidth=2, label='OLS regr. fct.')
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/SLR-Sim-Sample.pdf')
```

Output of Script 2.14: `SLR-Sim-Sample.py`

```
b:
Intercept    1.190238
x            0.444255
dtype: float64

x_sq_mean: 17.27675304867723

x_var: 953.7353266586754
```

Then a random sample of x and y is drawn in three steps:

- A sample of regressors x is drawn from an arbitrary distribution. The only thing we have to make sure to stay consistent with Assumption SLR.3 is that its variance is strictly positive. We choose a normal distribution with mean 4 and a standard deviation of 1.
- A sample of error terms u is drawn according to Assumptions SLR.4 and SLR.5: It has a mean of zero, and both the mean and the variance are unrelated to x. We simply choose a normal distribution with mean 0 and standard deviation $\sigma = 2$ for all 1000 observations independent of x. In Sections 2.7.3 and 2.7.4 we will adjust this to simulate the effects of a violation of these assumptions.
- Finally, we generate the dependent variable y according to the population regression function specified in Assumption SLR.1.

In an empirical project, we only observe x and y and not the realizations of the error term u. In the simulation, we "forget" them and the fact that we know the population parameters and estimate them from our sample using OLS. As motivated in Section 1.9, this will help us to study the behavior of the estimator in a sample like ours.

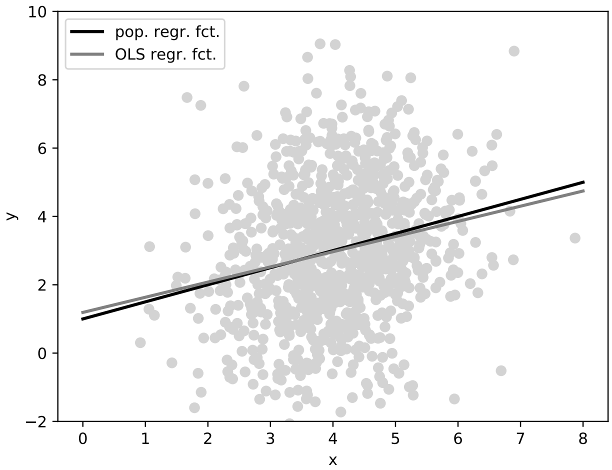
For our particular sample, the OLS parameter estimates are $\hat{\beta}_0 = 1.190238$ and $\hat{\beta}_1 = 0.444255$. The result of the graph generated in the last lines of Script 2.14 (`SLR-Sim-Sample.py`) is shown in Figure 2.4. It shows the population regression function with intercept $\beta_0 = 1$ and slope $\beta_1 = 0.5$. It also shows the scatter plot of the sample drawn from this population. This sample led to our OLS regression line with intercept $\hat{\beta}_0 = 1.190238$ and slope $\hat{\beta}_1 = 0.444255$ shown in gray.

Since the SLR assumptions hold in our exercise, Theorems 2.1 and 2.2 of Wooldridge (2019) should apply. Theorem 2.1 implies for our model that the estimators are unbiased, i.e.

$$\mathrm{E}(\hat{\beta}_0) = \beta_0 = 1 \qquad\qquad \mathrm{E}(\hat{\beta}_1) = \beta_1 = 0.5$$

The estimates obtained from our sample are relatively close to their population values. Obviously, we can never expect to hit the population parameter exactly. If we change the random seed by specifying a different number in Script 2.14 (`SLR-Sim-Sample.py`), we get a different sample and different parameter estimates.

Figure 2.4. Simulated Sample and OLS Regression Line



Theorem 2.2 of Wooldridge (2019) states the sampling variance of the estimators conditional on the sample values $\{x_1, \ldots, x_n\}$. It involves the average squared value $\overline{x^2} = 17.277$ and the sum of squares $\sum_{i-1}^{n}(x - \overline{x})^2 = 953.735$ which we also know from the *Python* output:

$$\text{Var}(\hat{\beta}_0) = \frac{\sigma^2 \overline{x^2}}{\sum_{i=1}^{n}(x - \overline{x})^2} = \frac{4 \cdot 17.277}{953.735} = 0.0725$$

$$\text{Var}(\hat{\beta}_1) = \frac{\sigma^2}{\sum_{i=1}^{n}(x - \overline{x})^2} = \frac{4}{953.735} = 0.0042$$

If Wooldridge (2019) is right, the standard error of $\hat{\beta}_1$ is $\sqrt{0.0042} = 0.0648$. So getting an estimate of $\hat{\beta}_1 = 0.444$ for one sample doesn't seem unreasonable given $\beta_1 = 0.5$.

2.7.2. Many Samples

Since the expected values and variances of our estimators are defined over separate random samples from the same population, it makes sense for us to repeat our simulation exercise over many simulated samples. Just as motivated in Section 1.9, the distribution of OLS parameter estimates across these samples will correspond to the sampling distribution of the estimators.

Script 2.16 (`SLR-Sim-Model-Condx.py`) implements this with the same **`for`** loop we introduced in Section 1.8.2 and already used for basic Monte Carlo simulations in Section 1.9.1. Remember that *Python* enthusiasts might choose a different technique but for us, this implementation has the big advantage that it is very transparent. We analyze $r = 10\,000$ samples.

Note that we use the same values for x in all samples since we draw them outside of the loop. We do this to simulate the exact setup of Theorem 2.2 which reports the sampling variances *conditional* on x. In a more realistic setup, we would sample x along with y. The conceptual difference is subtle and the results hardly differ in reasonably large samples. We will come back to these issues in

Chapter 5.[2] For each sample, we estimate our parameters and store them in the respective position $j = 0, \ldots, r-1$ of the arrays **b0** and **b1**.

Script 2.16: SLR-Sim-Model-Condx.py

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
b0 = np.empty(r)
b1 = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of y:
    u = stats.norm.rvs(0, su, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate and store parameters by OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b0[i] = results.params['Intercept']
    b1[i] = results.params['x']

# MC estimate of the expected values:
b0_mean = np.mean(b0)
b1_mean = np.mean(b1)

print(f'b0_mean: {b0_mean}\n')
print(f'b1_mean: {b1_mean}\n')

# MC estimate of the variances:
b0_var = np.var(b0, ddof=1)
b1_var = np.var(b1, ddof=1)

print(f'b0_var: {b0_var}\n')
print(f'b1_var: {b1_var}\n')
```

[2] In Script 2.15 (`SLR-Sim-Model.py`) shown on page 340, we implement the joint sampling from x and y. The results are essentially the same.

```
# graph:
x_range = np.linspace(0, 8, num=100)
plt.ylim([0, 6])

# add population regression line:
plt.plot(x_range, beta0 + beta1 * x_range, color='black',
         linestyle='-', linewidth=2, label='Population')

# add first OLS regression line (to attach a label):
plt.plot(x_range, b0[0] + b1[0] * x_range, color='grey',
         linestyle='-', linewidth=0.5, label='OLS regressions')

# add OLS regression lines no. 2 to 10:
for i in range(1, 10):
    plt.plot(x_range, b0[i] + b1[i] * x_range, color='grey',
             linestyle='-', linewidth=0.5)
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/SLR-Sim-Model-Condx.pdf')
```

Output of Script 2.16: `SLR-Sim-Model-Condx.py`

```
b0_mean: 1.00329460319241

b1_mean: 0.49936958775965984

b0_var: 0.07158103946245628

b1_var: 0.004157652196227234
```

Script 2.16 (`SLR-Sim-Model-Condx.py`) gives descriptive statistics of the $r = 10,000$ estimates we got from our simulation exercise. Wooldridge (2019, Theorem 2.1) claims that the OLS estimators are unbiased, so we should expect to get estimates which are very close to the respective population parameters. This is clearly confirmed. The average value of $\hat{\beta}_0$ is very close to $\beta_0 = 1$ and the average value of $\hat{\beta}_1$ is very close to $\beta_1 = 0.5$.

The simulated sampling variances are $\widetilde{\text{Var}}(\hat{\beta}_0) = 0.0716$ and $\widetilde{\text{Var}}(\hat{\beta}_1) = 0.0042$. Also these values are very close to the ones we expected from Theorem 2.2. The last lines of the code produce Figure 2.5. It shows the OLS regression lines for the first 10 simulated samples together with the population regression function.

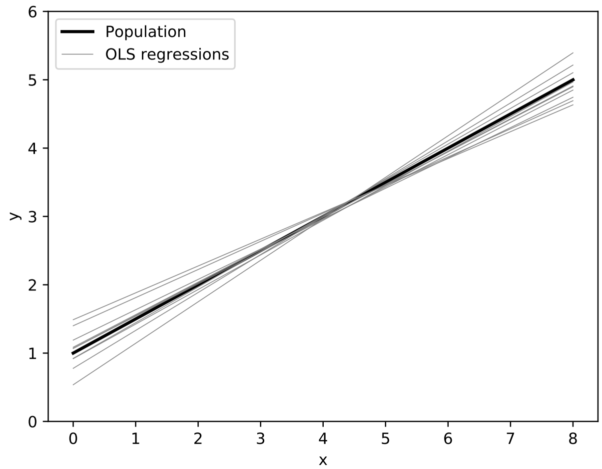
2.7.3. Violation of SLR.4

We will come back to a more systematic discussion of the consequences of violating the SLR assumptions below. At this point, we can already simulate the effects. In order to implement a violation of SLR.4 (zero conditional mean), consider a case where in the population u is not mean independent of x. A simple example is

$$\text{E}(u|x) = \frac{x-4}{5}.$$

What happens to our OLS estimator? Script 2.17 (`SLR-Sim-Model-ViolSLR4.py`) implements a simulation of this model and is listed in the appendix (p. 342).

Figure 2.5. Population and Simulated OLS Regression Lines



The only line of code we changed compared to Script 2.16 (`SLR-Sim-Model-Condx.py`) is the sampling of **u** which now reads

```
u_mean = np.array((x - 4) / 5)
u = stats.norm.rvs(u_mean, su, size=n)
```

The simulation results are presented in the output of Script 2.17 (`SLR-Sim-Model-ViolSLR4.py`). Obviously, the OLS coefficients are now biased: The average estimates are far from the population parameters $\beta_0 = 1$ and $\beta_1 = 0.5$. This confirms that Assumption SLR.4 is required to hold for the unbiasedness shown in Theorem 2.1.

Output of Script 2.17: `SLR-Sim-Model-ViolSLR4.py`

```
b0_mean: 0.2032946031924096

b1_mean: 0.6993695877596598

b0_var: 0.07158103946245628

b1_var: 0.004157652196227234
```

2.7.4. Violation of SLR.5

Theorem 2.1 (unbiasedness) does not require Assumption SLR.5 (homoscedasticity), but Theorem 2.2 (sampling variance) does. As an example for a violation consider the population specification

$$\mathrm{Var}(u|x) = \frac{4}{e^{4.5}} \cdot e^x,$$

so SLR.5 is clearly violated since the variance depends on x. We assume exogeneity, so assumption SLR.4 holds. The factor in front ensures that the unconditional variance $\mathrm{Var}(u) = 4$.[3] Based on this

[3]Since $x \sim \mathrm{Normal}(4, 1)$, e^x is log-normally distributed and has a mean of $e^{4.5}$.

unconditional variance only, the sampling variance should not change compared to the results above and we would still expect $\text{Var}(\hat{\beta}_0) = 0.0716$ and $\text{Var}(\hat{\beta}_1) = 0.0042$. But since Assumption SLR.5 is violated, Theorem 2.2 is not applicable.

Script 2.18 (`SLR-Sim-Model-ViolSLR5.py`) implements a simulation of this model and is listed in the appendix (p. 342). Here, we only had to change the line of code for the sampling of **u** to

```
u_var = np.array(4 / np.exp(4.5) * np.exp(x))
u = stats.norm.rvs(0, np.sqrt(u_var), size=n)
```

The output of Script 2.18 (`SLR-Sim-Model-ViolSLR5.py`) demonstrates two effects: The unbiasedness provided by Theorem 2.1 is unaffected, but the formula for sampling variance provided by Theorem 2.2 is incorrect.

Output of Script 2.18: `SLR-Sim-Model-ViolSLR5.py`

```
b0_mean: 1.001414297039418

b1_mean: 0.4997594115253497

b0_var: 0.13175544492656727

b1_var: 0.010016166348092534
```

3. Multiple Regression Analysis: Estimation

Running a multiple regression in *Python* is as straightforward as running a simple regression using the **ols** command in **statsmodels**. Section 3.1 shows how it is done. Section 3.2 opens the black box and replicates the main calculations using matrix algebra. This is not required for the remaining chapters, so it can be skipped by readers who prefer to keep black boxes closed.

Section 3.3 should not be skipped since it discusses the interpretation of regression results and the prevalent omitted variables problems. Finally, Section 3.4 covers standard errors and multicollinearity for multiple regression.

3.1. Multiple Regression in Practice

Consider the population regression model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots + \beta_k x_k + u \tag{3.1}$$

and suppose the data set **sample** contains variables **y**, **x1**, **x2**, **x3**, with the respective data of our sample. We estimate the model parameters by OLS using the commands

```
reg = smf.ols(formula='y ~ x1 + x2 + x3', data=sample)
results = reg.fit()
```

The tilde "**~**" again separates the dependent variable from the regressors which are now separated using a "**+**" sign. We can add options as before. The constant is again automatically added unless it is explicitly suppressed using **'y ~ 0 + x1 + x2 + x3 + ...'**.

We are already familiar with the workings of **smf.ols** and **fit**: The first command creates an object which contains all relevant information and the estimation is performed in a second step. The estimation results are stored in a variable **results** using the code **results = reg.fit()**. We can use this variable for further analyses. For a typical regression output including a coefficient table, call **results.summary()**. Of course if this is all we want, we can leave these steps and simply call **smf.ols(...).fit().summary()** in one step. Further analyses involving residuals, fitted values and the like can be used exactly as presented in Chapter 2.

The output of **summary** includes parameter estimates, standard errors according to Theorem 3.2 of Wooldridge (2019), the coefficient of determination R^2, and many more useful results we cannot interpret yet before we have worked through Chapter 4.

Wooldridge, Example 3.1: Determinants of College GPA

This example from Wooldridge (2019) relates the college GPA (`colGPA`) to the high school GPA (`hsGPA`) and achievement test score (`ACT`) for a sample of 141 students. The commands and results can be found in Script 3.1 (`Example-3-1.py`). The OLS regression function is

$$\widehat{\texttt{colGPA}} = 1.286 + 0.453 \cdot \texttt{hsGPA} + 0.0094 \cdot \texttt{ACT}.$$

Script 3.1: `Example-3-1.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

gpa1 = woo.dataWoo('gpa1')

reg = smf.ols(formula='colGPA ~ hsGPA + ACT', data=gpa1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 3.1: `Example-3-1.py`

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                 colGPA   R-squared:                       0.176
Model:                            OLS   Adj. R-squared:                  0.164
Method:                 Least Squares   F-statistic:                     14.78
Date:                Tue, 12 May 2020   Prob (F-statistic):           1.53e-06
Time:                        10:34:11   Log-Likelihood:                -46.573
No. Observations:                 141   AIC:                             99.15
Df Residuals:                     138   BIC:                             108.0
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.2863      0.341      3.774      0.000       0.612       1.960
hsGPA          0.4535      0.096      4.733      0.000       0.264       0.643
ACT            0.0094      0.011      0.875      0.383      -0.012       0.031
==============================================================================
Omnibus:                        3.056   Durbin-Watson:                   1.885
Prob(Omnibus):                  0.217   Jarque-Bera (JB):                2.469
Skew:                           0.199   Prob(JB):                        0.291
Kurtosis:                       2.488   Cond. No.                         298.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

Wooldridge, Example 3.4: Determinants of College GPA

For the regression run in Example 3.1, the output of Script 3.1 (`Example-3-1.py`) reports $R^2 = 0.176$, so about 17.6% of the variance in college GPA is explained by the two regressors.

Examples 3.2, 3.3, 3.5, 3.6: Further Multiple Regression Examples

In order to get a feeling of the methods and results, we present the analyses including the full regression tables of the mentioned Examples from Wooldridge (2019) in Scripts 3.2 (`Example-3-2.py`) through 3.6 (`Example-3-6.py`). See Wooldridge (2019) for descriptions of the data sets and variables and for comments on the results.

Script 3.2: Example-3-2.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ educ + exper + tenure', data=wage1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 3.2: Example-3-2.py

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:           np.log(wage)   R-squared:                       0.316
Model:                            OLS   Adj. R-squared:                  0.312
Method:                 Least Squares   F-statistic:                     80.39
Date:                Tue, 12 May 2020   Prob (F-statistic):           9.13e-43
Time:                        10:34:13   Log-Likelihood:                -313.55
No. Observations:                 526   AIC:                             635.1
Df Residuals:                     522   BIC:                             652.2
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.2844      0.104      2.729      0.007       0.080       0.489
educ           0.0920      0.007     12.555      0.000       0.078       0.106
exper          0.0041      0.002      2.391      0.017       0.001       0.008
tenure         0.0221      0.003      7.133      0.000       0.016       0.028
==============================================================================
Omnibus:                       11.534   Durbin-Watson:                   1.769
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               20.941
Skew:                           0.021   Prob(JB):                     2.84e-05
Kurtosis:                       3.977   Cond. No.                         135.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

Script 3.3: **Example-3-3.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

k401k = woo.dataWoo('401k')

reg = smf.ols(formula='prate ~ mrate + age', data=k401k)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 3.3: **Example-3-3.py**

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                  prate   R-squared:                       0.092
Model:                            OLS   Adj. R-squared:                  0.091
Method:                 Least Squares   F-statistic:                     77.79
Date:                Tue, 12 May 2020   Prob (F-statistic):           6.67e-33
Time:                        10:34:15   Log-Likelihood:                -6422.3
No. Observations:                1534   AIC:                         1.285e+04
Df Residuals:                    1531   BIC:                         1.287e+04
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     80.1190      0.779    102.846      0.000      78.591      81.647
mrate          5.5213      0.526     10.499      0.000       4.490       6.553
age            0.2431      0.045      5.440      0.000       0.155       0.331
==============================================================================
Omnibus:                      375.579   Durbin-Watson:                   1.910
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              805.992
Skew:                          -1.387   Prob(JB):                    9.57e-176
Kurtosis:                       5.217   Cond. No.                         32.5
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

Script 3.4: `Example-3-5a.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

crime1 = woo.dataWoo('crime1')

# model without avgsen:
reg = smf.ols(formula='narr86 ~ pcnv + ptime86 + qemp86', data=crime1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 3.4: `Example-3-5a.py`

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                 narr86   R-squared:                       0.041
Model:                            OLS   Adj. R-squared:                  0.040
Method:                 Least Squares   F-statistic:                     39.10
Date:                Tue, 12 May 2020   Prob (F-statistic):           9.91e-25
Time:                        10:34:16   Log-Likelihood:                -3394.7
No. Observations:                2725   AIC:                             6797.
Df Residuals:                    2721   BIC:                             6821.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.7118      0.033     21.565      0.000       0.647       0.776
pcnv          -0.1499      0.041     -3.669      0.000      -0.230      -0.070
ptime86       -0.0344      0.009     -4.007      0.000      -0.051      -0.018
qemp86        -0.1041      0.010    -10.023      0.000      -0.124      -0.084
==============================================================================
Omnibus:                     2394.860   Durbin-Watson:                   1.836
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           106169.153
Skew:                           4.002   Prob(JB):                         0.00
Kurtosis:                      32.513   Cond. No.                         8.27
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

Script 3.5: `Example-3-5b.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

crime1 = woo.dataWoo('crime1')

# model with avgsen:
reg = smf.ols(formula='narr86 ~ pcnv + avgsen + ptime86 + qemp86', data=crime1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 3.5: `Example-3-5b.py`

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                 narr86   R-squared:                       0.042
Model:                            OLS   Adj. R-squared:                  0.041
Method:                 Least Squares   F-statistic:                     29.96
Date:                Tue, 12 May 2020   Prob (F-statistic):           2.01e-24
Time:                        10:34:17   Log-Likelihood:                -3393.5
No. Observations:                2725   AIC:                             6797.
Df Residuals:                    2720   BIC:                             6826.
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.7068      0.033     21.319      0.000       0.642       0.772
pcnv          -0.1508      0.041     -3.692      0.000      -0.231      -0.071
avgsen         0.0074      0.005      1.572      0.116      -0.002       0.017
ptime86       -0.0374      0.009     -4.252      0.000      -0.055      -0.020
qemp86        -0.1033      0.010     -9.940      0.000      -0.124      -0.083
==============================================================================
Omnibus:                     2396.990   Durbin-Watson:                   1.837
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           106841.658
Skew:                           4.006   Prob(JB):                         0.00
Kurtosis:                      32.611   Cond. No.                         10.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

Script 3.6: Example-3-6.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ educ', data=wage1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 3.6: Example-3-6.py

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:           np.log(wage)   R-squared:                       0.186
Model:                            OLS   Adj. R-squared:                  0.184
Method:                 Least Squares   F-statistic:                     119.6
Date:                Tue, 12 May 2020   Prob (F-statistic):           3.27e-25
Time:                        10:34:19   Log-Likelihood:                -359.38
No. Observations:                 526   AIC:                             722.8
Df Residuals:                     524   BIC:                             731.3
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.5838      0.097      5.998      0.000       0.393       0.775
educ           0.0827      0.008     10.935      0.000       0.068       0.098
==============================================================================
Omnibus:                       11.804   Durbin-Watson:                   1.801
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               13.811
Skew:                           0.268   Prob(JB):                      0.00100
Kurtosis:                       3.586   Cond. No.                         60.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

3.2. OLS in Matrix Form

For applying regression methods to empirical problems, we do not actually need to know the formulas our software uses. In multiple regression, we need to resort to matrix algebra in order to find an explicit expression for the OLS parameter estimates. Wooldridge (2019) defers this discussion to Appendix E and we follow the notation used there. Going through this material is not required for applying multiple regression to real-world problems but is useful for a deeper understanding of the methods and their black-box implementations in software packages. In the following chapters, we will rely on the comfort of the canned routine `fit`, so this section may be skipped.

In matrix form, we store the regressors in a $n \times (k+1)$ matrix $\mathbf{X}$ which has a column for each regressor plus a column of ones for the constant. The sample values of the dependent variable are stored in a $n \times 1$ column vector $\mathbf{y}$. Wooldridge (2019) derives the OLS estimator $\hat{\boldsymbol{\beta}} = (\hat{\beta}_0, \hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_k)'$ to be

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}. \tag{3.2}$$

This equation involves three matrix operations which we know how to implement in *Python* from Section 1.2.3:

- Transpose: The expression $\mathbf{X}'$ is **X.T** in **numpy**
- Matrix multiplication: The expression $\mathbf{X}'\mathbf{X}$ is translated as **X.T @ X**
- Inverse: $(\mathbf{X}'\mathbf{X})^{-1}$ is written as **np.linalg.inv(X.T @ X)**

So we can collect everything and translate Equation 3.2 into the somewhat unsightly expression

```
b = np.linalg.inv(X.T @ X) @ X.T @ y
```

The vector of residuals can be manually calculated as

$$\hat{\mathbf{u}} = \mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}} \tag{3.3}$$

or translated into the **numpy** matrix language

```
u_hat = y - X @ b
```

The formula for the estimated variance of the error term is

$$\hat{\sigma}^2 = \tfrac{1}{n-k-1}\hat{\mathbf{u}}'\hat{\mathbf{u}} \tag{3.4}$$

which is equivalent to

```
sigsq_hat = (u_hat.T @ u_hat) / (n - k - 1)
```

The standard error of the regression (SER) is its square root $\hat{\sigma} = \sqrt{\hat{\sigma}^2}$. The estimated OLS variance-covariance matrix according to Wooldridge (2019, Theorem E.2) is then

$$\widehat{\text{Var}(\hat{\boldsymbol{\beta}})} = \hat{\sigma}^2(\mathbf{X}'\mathbf{X})^{-1} \tag{3.5}$$

```
Vb_hat = sigsq_hat * np.linalg.inv(X.T @ X)
```

Finally, the standard errors of the parameter estimates are the square roots of the main diagonal of $\text{Var}(\hat{\boldsymbol{\beta}})$ which can be expressed in **numpy** as

```
se = np.sqrt(np.diagonal(Vb_hat))
```

Script 3.7 (`OLS-Matrices.py`) implements this for the GPA regression from Example 3.1. Comparing the results to the built-in function (see Script 3.1 (`Example-3-1.py`)), it is reassuring that we get exactly the same numbers for the parameter estimates and standard errors of the coefficients. Script 3.7 (`OLS-Matrices.py`) also demonstrates another way of generating $\mathbf{y}$ and $\mathbf{X}$ by using the module **patsy**. It includes the command **dmatrices**, which allows to conveniently create the matrices by formula syntax.

Script 3.7: OLS-Matrices.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import patsy as pt

gpa1 = woo.dataWoo('gpa1')

# determine sample size & no. of regressors:
n = len(gpa1)
k = 2

# extract y:
y = gpa1['colGPA']

# extract X & add a column of ones:
X = pd.DataFrame({'const': 1, 'hsGPA': gpa1['hsGPA'], 'ACT': gpa1['ACT']})

# alternative with patsy:
y2, X2 = pt.dmatrices('colGPA ~ hsGPA + ACT', data=gpa1, return_type='dataframe')

# display first rows of X:
print(f'X.head(): \n{X.head()}\n')

# parameter estimates:
X = np.array(X)
y = np.array(y).reshape(n, 1)  # creates a row vector
b = np.linalg.inv(X.T @ X) @ X.T @ y
print(f'b: \n{b}\n')

# residuals, estimated variance of u and SER:
u_hat = y - X @ b
sigsq_hat = (u_hat.T @ u_hat) / (n - k - 1)
SER = np.sqrt(sigsq_hat)
print(f'SER: {SER}\n')

# estimated variance of the parameter estimators and SE:
Vbeta_hat = sigsq_hat * np.linalg.inv(X.T @ X)
se = np.sqrt(np.diagonal(Vbeta_hat))
print(f'se: {se}\n')
```

Output of Script 3.7: OLS-Matrices.py

```
X.head():
   const  hsGPA  ACT
0      1    3.0   21
1      1    3.2   24
2      1    3.6   26
3      1    3.5   27
4      1    3.9   28

b:
[[1.28632777]
 [0.45345589]
 [0.00942601]]

SER: [[0.34031576]]

se: [0.34082212 0.09581292 0.01077719]
```

3.3. Ceteris Paribus Interpretation and Omitted Variable Bias

The parameters in a multiple regression can be interpreted as partial effects. In a general model with k regressors, the estimated slope parameter β_j associated with variable x_j is the change of $\hat{y}$ as x_j increases by one unit *and the other variables are held fixed*.

Wooldridge (2019) discusses this interpretation in Section 3.2 and offers a useful formula for interpreting the difference between simple regression results and this *ceteris paribus* interpretation of multiple regression: Consider a regression with two explanatory variables:

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2. \tag{3.6}$$

The parameter $\hat{\beta}_1$ is the estimated effect of increasing x_1 by one unit while keeping x_2 fixed. In contrast, consider the simple regression including only x_1 as a regressor:

$$\tilde{y} = \tilde{\beta}_0 + \tilde{\beta}_1 x_1. \tag{3.7}$$

The parameter $\tilde{\beta}_1$ is the estimated effect of increasing x_1 by one unit (and NOT keeping x_2 fixed). It can be related to $\hat{\beta}_1$ using the formula

$$\tilde{\beta}_1 = \hat{\beta}_1 + \hat{\beta}_2 \tilde{\delta}_1 \tag{3.8}$$

where $\tilde{\delta}_1$ is the slope parameter of the linear regression of x_2 on x_1

$$x_2 = \tilde{\delta}_0 + \tilde{\delta}_1 x_1. \tag{3.9}$$

This equation is actually quite intuitive: As x_1 increases by one unit,

- Predicted y directly increases by $\hat{\beta}_1$ units (*ceteris paribus* effect, Equ. 3.6).
- Predicted x_2 increases by $\tilde{\delta}_1$ units (see Equ. 3.9).
- Each of these $\tilde{\delta}_1$ units leads to an increase of predicted y by $\hat{\beta}_2$ units, giving a total indirect effect of $\tilde{\delta}_1 \hat{\beta}_2$ (see again Equ. 3.6)
- The overall effect $\tilde{\beta}_1$ is the sum of the direct and indirect effects (see Equ. 3.8).

We revisit Example 3.1 to see whether we can demonstrate Equation 3.8 in *Python*. Script 3.8 (`Omitted-Vars.py`) repeats the regression of the college GPA (`colGPA`) on the achievement test score (`ACT`) and the high school GPA (`hsGPA`). We study the *ceteris paribus* effect of `ACT` on `colGPA` which has an estimated value of $\hat{\beta}_1 = 0.0094$. The estimated effect of `hsGPA` is $\hat{\beta}_2 = 0.453$. The slope parameter of the regression corresponding to Equation 3.9 is $\tilde{\delta}_1 = 0.0389$. Plugging these values into Equation 3.8 gives a total effect of $\tilde{\beta}_1 = 0.0271$ which is exactly what the simple regression at the end of the output delivers.

In this example, the indirect effect is actually stronger than the direct effect. `ACT` predicts `colGPA` mainly because it is related to `hsGPA` which in turn is strongly related to `colGPA`.

These relations hold for the estimates from a given sample. In Section 3.3, Wooldridge (2019) discusses how to apply the same sort of arguments to the OLS estimators which are random variables varying over different samples. Omitting relevant regressors causes bias if we are interested in estimating partial effects. In practice, it is difficult to include *all* relevant regressors making of omitted variables a prevalent problem. It is important enough to have motivated a vast amount of methodological and applied research. More advanced techniques like instrumental variables or panel data methods try to solve the problem in cases where we cannot add all relevant regressors, for example because they are unobservable. We will come back to this in Part 3.

Script 3.8: `Omitted-Vars.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

gpa1 = woo.dataWoo('gpa1')

# parameter estimates for full and simple model:
reg = smf.ols(formula='colGPA ~ ACT + hsGPA', data=gpa1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')

# relation between regressors:
reg_delta = smf.ols(formula='hsGPA ~ ACT', data=gpa1)
results_delta = reg_delta.fit()
delta_tilde = results_delta.params
print(f'delta_tilde: \n{delta_tilde}\n')

# omitted variables formula for b1_tilde:
b1_tilde = b['ACT'] + b['hsGPA'] * delta_tilde['ACT']
print(f'b1_tilde:  \n{b1_tilde}\n')

# actual regression with hsGPA omitted:
reg_om = smf.ols(formula='colGPA ~ ACT', data=gpa1)
results_om = reg_om.fit()
b_om = results_om.params
print(f'b_om: \n{b_om}\n')
```

Output of Script 3.8: `Omitted-Vars.py`

```
b:
Intercept    1.286328
ACT          0.009426
hsGPA        0.453456
dtype: float64

delta_tilde:
Intercept    2.462537
ACT          0.038897
dtype: float64

b1_tilde:
0.027063973943178612

b_om:
Intercept    2.402979
ACT          0.027064
dtype: float64
```

3.4. Standard Errors, Multicollinearity, and VIF

We have already seen the matrix formula for the conditional variance-covariance matrix under the usual assumptions including homoscedasticity (MLR.5) in Equation 3.5. Theorem 3.2 provides another useful formula for the variance of a single parameter β_j, i.e. for a single element on the main diagonal of the variance-covariance matrix:

$$\text{Var}(\hat{\beta}_j) = \frac{\sigma^2}{SST_j(1-R_j^2)} = \frac{1}{n} \cdot \frac{\sigma^2}{\text{Var}(x_j)} \cdot \frac{1}{1-R_j^2}, \tag{3.10}$$

where $SST_j = \sum_{i=1}^n (x_{ji} - \overline{x}_j)^2 = (n-1) \cdot Var(x_j)$ is the total sum of squares and R_j^2 is the usual coefficient of determination from a regression of x_j on all of the other regressors.[1]

The variance of $\hat{\beta}_j$ consists of four parts:

- $\frac{1}{n}$: The variance is smaller for larger samples.
- σ^2: The variance is larger if the error term varies a lot, since it introduces randomness into the relationship between the variables of interest.
- $\frac{1}{\text{Var}(x_j)}$: The variance is smaller if the regressor x_j varies a lot since this provides relevant information about the relationship.
- $\frac{1}{1-R_j^2}$: This variance inflation factor (VIF) accounts for (imperfect) multicollinearity. If x_j is highly related to the other regressors, R_j^2 and therefore also VIF_j and the variance of $\hat{\beta}_j$ are large.

Since the error variance σ^2 is unknown, we replace it with an estimate to come up with an estimated variance of the parameter estimate. Its square root is the standard error

$$\text{se}(\hat{\beta}_j) = \frac{1}{\sqrt{n}} \cdot \frac{\hat{\sigma}}{\text{sd}(x_j)} \cdot \frac{1}{\sqrt{1-R_j^2}}. \tag{3.11}$$

It is not directly obvious that this formula leads to the same results as the matrix formula in Equation 3.5. We will validate this formula by replicating Example 3.1 which we also used for manually calculating the SE using the matrix formula above. The calculations are shown in Script 3.9 (`MLR-SE.py`).

We also use this example to demonstrate how to extract results which are included in the object returned by the **`fit`** method. Given its results are stored in variable **`sures`** using the results of **`sures = smf.ols(...).fit()`**, we can easily access the information using **`sures.resultname`** where the **`resultname`** can be any of the following:

- **`params`** for the regression coefficients
- **`resid`** for the residuals
- **`mse_resid`** for the (squared) SER
- **`rsquared`** for R^2
- and more.

[1]Note that here, we use the population variance formula $\text{Var}(x_j) = \frac{1}{n}\sum_{i=1}^n (x_{ji} - \overline{x}_j)^2$.

Script 3.9: MLR-SE.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

gpa1 = woo.dataWoo('gpa1')

# full estimation results including automatic SE:
reg = smf.ols(formula='colGPA ~ hsGPA + ACT', data=gpa1)
results = reg.fit()

# extract SER (instead of calculation via residuals):
SER = np.sqrt(results.mse_resid)

# regressing hsGPA on ACT for calculation of R2 & VIF:
reg_hsGPA = smf.ols(formula='hsGPA ~ ACT', data=gpa1)
results_hsGPA = reg_hsGPA.fit()
R2_hsGPA = results_hsGPA.rsquared
VIF_hsGPA = 1 / (1 - R2_hsGPA)
print(f'VIF_hsGPA: {VIF_hsGPA}\n')

# manual calculation of SE of hsGPA coefficient:
n = results.nobs
sdx = np.std(gpa1['hsGPA'], ddof=1) * np.sqrt((n - 1) / n)
SE_hsGPA = 1 / np.sqrt(n) * SER / sdx * np.sqrt(VIF_hsGPA)
print(f'SE_hsGPA: {SE_hsGPA}\n')
```

Output of Script 3.9: MLR-SE.py

```
VIF_hsGPA: 1.1358234481972789

SE_hsGPA: 0.09581291608057597
```

This is used in Script 3.9 (MLR-SE.py) to extract the SER of the main regression and the R_j^2 from the regression of hsGPA on ACT which is needed for calculating the VIF for the coefficient of hsGPA.[2] The other ingredients of Equation 3.11 are straightforward. The standard error calculated this way is exactly the same as the one of the built-in command and the matrix formula used in Script 3.7 (OLS-Matrices.py).

A convenient way to automatically calculate variance inflation factors (VIF) is provided by the module **statsmodels** in **stats.outliers_influence**. The command **variance_inflation_factor(X, regressornumber)** delivers the VIF for a matrix **X** and the number of a given regressor (starting with the constant as the regressor with number **0**). The calculation for each of the regressors is performed in a loop as demonstrated in Script 3.10 (MLR-VIF.py).

We extend Example 3.6. and regress individual log wage on education (educ), potential overall work experience (exper), and the number of years with current employer (tenure). We could imagine that these three variables are correlated with each other, but the results show no big VIF. The largest one is for the coefficient of exper. Its variance is higher by a factor of (only) 1.478 than in a world in which it were uncorrelated with the other regressors. So we don't have to worry about multicollinearity here.

[2]We could have calculated these values manually like in Scripts 2.8 (Example-2-8.py), 2.13 (Example-2-12.py) or 3.7 (OLS-Matrices.py).

Script 3.10: `MLR-VIF.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.stats.outliers_influence as smo
import patsy as pt

wage1 = woo.dataWoo('wage1')

# extract matrices using patsy:
y, X = pt.dmatrices('np.log(wage) ~ educ + exper + tenure',
                    data=wage1, return_type='dataframe')

# get VIF:
K = X.shape[1]
VIF = np.empty(K)
for i in range(K):
    VIF[i] = smo.variance_inflation_factor(X.values, i)
print(f'VIF: \n{VIF}\n')
```

Output of Script 3.10: `MLR-VIF.py`

```
VIF:
[29.37890286  1.11277075  1.47761777  1.34929556]
```

4. Multiple Regression Analysis: Inference

Section 4.1 of Wooldridge (2019) adds assumption MLR.6 (normal distribution of the error term) to the previous assumptions MLR.1 through MLR.5. Together, these assumptions constitute the classical linear model (CLM).

The main additional result we get from this assumption is stated in Theorem 4.1: The OLS parameter estimators are normally distributed (conditional on the regressors $x_1, ..., x_k$). The benefit of this result is that it allows us to do statistical inference similar to the approaches discussed in Section 1.7 for the simple estimator of the mean of a normally distributed random variable.

4.1. The t Test

After the sign and magnitude of the estimated parameters, empirical research typically pays most attention to the results of t tests discussed in this section.

4.1.1. General Setup

An important type of hypotheses we are often interested in is of the form

$$H_0 : \beta_j = a_j, \tag{4.1}$$

where a_j is some given number, very often $a_j = 0$. For the most common case of two-tailed tests, the alternative hypothesis is

$$H_1 : \beta_j \neq a_j, \tag{4.2}$$

and for one-tailed tests it is either one of

$$H_1 : \beta_j < a_j \qquad \text{or} \qquad H_1 : \beta_j > a_j. \tag{4.3}$$

These hypotheses can be conveniently tested using a t test which is based on the test statistic

$$t = \frac{\hat{\beta}_j - a_j}{\text{se}(\hat{\beta}_j)}. \tag{4.4}$$

If H_0 is in fact true and the CLM assumptions hold, then this statistic has a t distribution with $n - k - 1$ degrees of freedom.

4.1.2. Standard Case

Very often, we want to test whether there is any relation at all between the dependent variable y and a regressor x_j and do not want to impose a sign on the partial effect *a priori*. This is a mission for the standard two-sided t test with the hypothetical value $a_j = 0$, so

$$H_0 : \beta_j = 0, \qquad H_1 : \beta_j \neq 0, \tag{4.5}$$

$$t_{\hat{\beta}_j} = \frac{\hat{\beta}_j}{\text{se}(\hat{\beta}_j)}. \tag{4.6}$$

The subscript on the t statistic indicates that this is "the" t value for $\hat{\beta}_j$ for this frequent version of the test. Under H_0, it has the t distribution with $n-k-1$ degrees of freedom implying that the probability that $|t_{\hat{\beta}_j}| > c$ is equal to α if c is the $1 - \frac{\alpha}{2}$ quantile of this distribution. If α is our significance level (e.g. $\alpha = 5\%$), then we

$$\text{reject } H_0 \text{ if } |t_{\hat{\beta}_j}| > c$$

in our sample. For the typical significance level $\alpha = 5\%$, the critical value c will be around 2 for reasonably large degrees of freedom and approach the counterpart of 1.96 from the standard normal distribution in very large samples.

The p value indicates the smallest value of the significance level α for which we would still reject H_0 using our sample. So it is the probability for a random variable T with the respective t distribution that $|T| > |t_{\hat{\beta}_j}|$ where $t_{\hat{\beta}_j}$ is the value of the t statistic in our particular sample. In our two-tailed test, it can be calculated as

$$p_{\hat{\beta}_j} = 2 \cdot F_{t_{n-k-1}}(-|t_{\hat{\beta}_j}|), \tag{4.7}$$

where $F_{t_{n-k-1}}(\cdot)$ is the CDF of the t distribution with $n-k-1$ degrees of freedom. If our software provides us with the relevant p values, they are easy to use: We

$$\text{reject } H_0 \text{ if } p_{\hat{\beta}_j} \leq \alpha.$$

Since this standard case of a t test is so common, **`statsmodels`** provides us with the relevant t and p values directly in the **`summary`** of the estimation results we already saw in the previous chapter. The regression table includes for all regressors and the intercept:

- parameter estimates and standard errors, see Section 3.1.
- test statistics $t_{\hat{\beta}_j}$ from Equation 4.6 in the column **`t`**
- respective p values $p_{\hat{\beta}_j}$ from Equation 4.7 in the column **`P>|t|`**
- respective 95% confidence interval from Equation 4.8 in columns **`[0.025`** and **`0.975]`** (see Section 4.2)

Wooldridge, Example 4.3: Determinants of College GPA

We have repeatedly used the data set `GPA1` in Chapter 3. This example uses three regressors and estimates a regression model of the form

$$\texttt{colGPA} = \beta_0 + \beta_1 \cdot \texttt{hsGPA} + \beta_2 \cdot \texttt{ACT} + \beta_3 \cdot \texttt{skipped} + u.$$

For the critical values of the t tests, using the normal approximation instead of the exact t distribution with $n-k-1 = 137$ d.f. doesn't make much of a difference:

Script 4.1: `Example-4-3-cv.py`

```
import scipy.stats as stats
import numpy as np

# CV for alpha=5% and 1% using the t distribution with 137 d.f.:
alpha = np.array([0.05, 0.01])
cv_t = stats.t.ppf(1 - alpha / 2, 137)
print(f'cv_t: {cv_t}\n')

# CV for alpha=5% and 1% using the normal approximation:
cv_n = stats.norm.ppf(1 - alpha / 2)
print(f'cv_n: {cv_n}\n')
```

Output of Script 4.1: `Example-4-3-cv.py`

```
cv_t: [1.97743121 2.61219198]

cv_n: [1.95996398 2.5758293 ]
```

Script 4.2 (`Example-4-3.py`) presents the standard **`summary`** which directly contains all the information to test the hypotheses in Equation 4.5 for all parameters. The t statistics for all coefficients except β_2 are larger in absolute value than the critical value $c = 2.61$ (or $c = 2.58$ using the normal approximation) for $\alpha = 1\%$. So we would reject H_0 for all usual significance levels. By construction, we draw the same conclusions from the p values.
In order to confirm that **`statsmodels`** is exactly using the formulas of Wooldridge (2019), we next reconstruct the t and p values manually. We extract the coefficients (**`params`**) and standard errors (**`bse`**) from the regression results, and simply apply Equations 4.6 and 4.7.

Script 4.2: `Example-4-3.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

gpa1 = woo.dataWoo('gpa1')

# store and display results:
reg = smf.ols(formula='colGPA ~ hsGPA + ACT + skipped', data=gpa1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')

# manually confirm the formulas, i.e. extract coefficients and SE:
b = results.params
se = results.bse

# reproduce t statistic:
tstat = b / se
print(f'tstat: \n{tstat}\n')

# reproduce p value:
pval = 2 * stats.t.cdf(-abs(tstat), 137)
print(f'pval: \n{pval}\n')
```

Output of Script 4.2: `Example-4-3.py`

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                 colGPA   R-squared:                       0.234
Model:                            OLS   Adj. R-squared:                  0.217
Method:                 Least Squares   F-statistic:                     13.92
Date:                Tue, 12 May 2020   Prob (F-statistic):           5.65e-08
Time:                        10:37:33   Log-Likelihood:                -41.501
No. Observations:                 141   AIC:                             91.00
Df Residuals:                     137   BIC:                             102.8
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      1.3896      0.332      4.191      0.000       0.734       2.045
hsGPA          0.4118      0.094      4.396      0.000       0.227       0.597
ACT            0.0147      0.011      1.393      0.166      -0.006       0.036
skipped       -0.0831      0.026     -3.197      0.002      -0.135      -0.032
==============================================================================
Omnibus:                        1.917   Durbin-Watson:                   1.881
Prob(Omnibus):                  0.383   Jarque-Bera (JB):                1.636
Skew:                           0.125   Prob(JB):                        0.441
Kurtosis:                       2.535   Cond. No.                         300.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly

tstat:
Intercept    4.191039
hsGPA        4.396260
ACT          1.393319
skipped     -3.196840
dtype: float64

pval:
[4.95026897e-05 2.19205015e-05 1.65779902e-01 1.72543113e-03]
```

4.1.3. Other Hypotheses

For a one-tailed test, the critical value c of the t test and the p values have to be adjusted appropriately. Wooldridge (2019) provides a general discussion in Section 4.2. For testing the null hypothesis $H_0 : \beta_j = a_j$, the tests for the three common alternative hypotheses are summarized in Table 4.1:

Table 4.1. One- and Two-tailed t Tests for $H_0 : \beta_j = a_j$

H_1 :	$\beta_j \neq a_j$	$\beta_j > a_j$	$\beta_j < a_j$
c=quantile	$1 - \frac{\alpha}{2}$	$1 - \alpha$	$1 - \alpha$
reject H_0 if	$\lvert t_{\hat{\beta}_j} \rvert > c$	$t_{\hat{\beta}_j} > c$	$t_{\hat{\beta}_j} < -c$
p value	$2 \cdot F_{t_{n-k-1}}(-\lvert t_{\hat{\beta}_j} \rvert)$	$F_{t_{n-k-1}}(-t_{\hat{\beta}_j})$	$F_{t_{n-k-1}}(t_{\hat{\beta}_j})$

Given the standard regression output like the one in Script 4.2 (`Example-4-3.py`) including the p value for two-sided tests $p_{\hat{\beta}_j}$, we can easily do one-sided t tests for the null hypothesis $H_0 : \beta_j = 0$ in two steps:

- Is $\hat{\beta}_j$ positive (if $H_1 : \beta_j > 0$) or negative (if $H_1 : \beta_j < 0$)?
  - No $\rightarrow$ Do not reject H_0 since this cannot be evidence against H_0.
  - Yes $\rightarrow$ The relevant p value is half of the reported $p_{\hat{\beta}_j}$.

    $\Rightarrow$ Reject H_0 if $p = \frac{1}{2} p_{\hat{\beta}_j} < \alpha$.

Wooldridge, Example 4.1: Hourly Wage Equation

We have already estimated the wage equation

$$\log(\texttt{wage}) = \beta_0 + \beta_1 \cdot \texttt{educ} + \beta_2 \cdot \texttt{exper} + \beta_3 \cdot \texttt{tenure} + u$$

in Example 3.2. Now we are ready to test $H_0 : \beta_2 = 0$ against $H_1 : \beta_2 > 0$. For the critical values of the t tests, using the normal approximation instead of the exact t distribution with $n - k - 1 = 522$ d.f. doesn't make any relevant difference:

Script 4.3: `Example-4-1-cv.py`

```
import scipy.stats as stats
import numpy as np

# CV for alpha=5% and 1% using the t distribution with 522 d.f.:
alpha = np.array([0.05, 0.01])
cv_t = stats.t.ppf(1 - alpha, 522)
print(f'cv_t: {cv_t}\n')

# CV for alpha=5% and 1% using the normal approximation:
cv_n = stats.norm.ppf(1 - alpha)
print(f'cv_n: {cv_n}\n')
```

Output of Script 4.3: `Example-4-1-cv.py`

```
cv_t: [1.64777794 2.33351273]

cv_n: [1.64485363 2.32634787]
```

Script 4.4 (`Example-4-1.py`) shows the standard regression output. The reported t statistic for the parameter of `exper` is $t_{\hat{\beta}_2} = 2.391$ which is larger than the critical value $c = 2.33$ for the significance level $\alpha = 1\%$, so we reject H_0. By construction, we get the same answer from looking at the p value. Like always, the reported $p_{\hat{\beta}_j}$ value is for a two-sided test, so we have to divide it by 2. The resulting value $p = \frac{0.017}{2} = 0.0085 < 0.01$, so we reject H_0 using an $\alpha = 1\%$ significance level.

Script 4.4: Example-4-1.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ educ + exper + tenure', data=wage1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Output of Script 4.4: Example-4-1.py

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:           np.log(wage)   R-squared:                       0.316
Model:                            OLS   Adj. R-squared:                  0.312
Method:                 Least Squares   F-statistic:                     80.39
Date:                Tue, 12 May 2020   Prob (F-statistic):           9.13e-43
Time:                        10:37:35   Log-Likelihood:                -313.55
No. Observations:                 526   AIC:                             635.1
Df Residuals:                     522   BIC:                             652.2
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.2844      0.104      2.729      0.007       0.080       0.489
educ           0.0920      0.007     12.555      0.000       0.078       0.106
exper          0.0041      0.002      2.391      0.017       0.001       0.008
tenure         0.0221      0.003      7.133      0.000       0.016       0.028
==============================================================================
Omnibus:                       11.534   Durbin-Watson:                   1.769
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               20.941
Skew:                           0.021   Prob(JB):                     2.84e-05
Kurtosis:                       3.977   Cond. No.                         135.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
```

4.2. Confidence Intervals

We have already looked at confidence intervals (CI) for the mean of a normally distributed random variable in Sections 1.7 and 1.9.3. CI for the regression parameters are equally easy to construct and closely related to t tests. Wooldridge (2019, Section 4.3) provides a succinct discussion. The 95% confidence interval for parameter β_j is simply

$$\hat{\beta}_j \pm c \cdot \mathrm{se}(\hat{\beta}_j), \tag{4.8}$$

where c is the same critical value for the two-sided t test using a significance level $\alpha = 5\%$. Wooldridge (2019) shows examples of how to manually construct these CI.

statsmodels provides the 95% confidence intervals for all parameters in the regression table. If you use the method **conf_int** on the object with the regression results, you can compute other significance levels. Script 4.5 (`Example-4-8.py`) demonstrates the procedure.

Wooldridge, Example 4.8: Model of R&D Expenditures

We study the relationship between the R&D expenditures of a firm, its size, and the profit margin for a sample of 32 firms in the chemical industry. The regression equation is

$$\log(\texttt{rd}) = \beta_0 + \beta_1 \cdot \log(\texttt{sales}) + \beta_2 \cdot \texttt{profmarg} + u.$$

Script 4.5 (`Example-4-8.py`) presents the regression results as well as the 95% and 99% CI. See Wooldridge (2019) for the manual calculation of the CI and comments on the results.

Script 4.5: `Example-4-8.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

rdchem = woo.dataWoo('rdchem')

# OLS regression:
reg = smf.ols(formula='np.log(rd) ~ np.log(sales) + profmarg', data=rdchem)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')

# 95% CI:
CI95 = results.conf_int(0.05)
print(f'CI95: \n{CI95}\n')

# 99% CI:
CI99 = results.conf_int(0.01)
print(f'CI99: \n{CI99}\n')
```

Output of Script 4.5: `Example-4-8.py`

```
results.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:             np.log(rd)   R-squared:                       0.918
Model:                            OLS   Adj. R-squared:                  0.912
Method:                 Least Squares   F-statistic:                     162.2
Date:                Tue, 12 May 2020   Prob (F-statistic):           1.79e-16
Time:                        10:37:37   Log-Likelihood:                -22.511
No. Observations:                  32   AIC:                             51.02
Df Residuals:                      29   BIC:                             55.42
Df Model:                           2
Covariance Type:            nonrobust
=================================================================================
                    coef    std err          t      P>|t|      [0.025      0.975]
---------------------------------------------------------------------------------
Intercept        -4.3783      0.468     -9.355      0.000      -5.335      -3.421
np.log(sales)     1.0842      0.060     18.012      0.000       0.961       1.207
profmarg          0.0217      0.013      1.694      0.101      -0.004       0.048
==============================================================================
Omnibus:                        0.670   Durbin-Watson:                   1.859
Prob(Omnibus):                  0.715   Jarque-Bera (JB):                0.671
Skew:                           0.308   Prob(JB):                        0.715
Kurtosis:                       2.649   Cond. No.                         70.6
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly

CI95:
                      0         1
Intercept     -5.335478 -3.421068
np.log(sales)  0.961107  1.207332
profmarg      -0.004488  0.047799

CI99:
                      0         1
Intercept     -5.668313 -3.088234
np.log(sales)  0.918299  1.250141
profmarg      -0.013578  0.056890
```

4.3. Linear Restrictions: F Tests

Wooldridge (2019, Sections 4.4 and 4.5) discusses more general tests than those for the null hypotheses in Equation 4.1. They can involve one or more hypotheses involving one or more population parameters in a linear fashion.

We follow the illustrative example of Wooldridge (2019, Section 4.5) and analyze major league baseball players' salaries using the data set `MLB1` and the regression model

$$\log(\texttt{salary}) = \beta_0 + \beta_1 \cdot \texttt{years} + \beta_2 \cdot \texttt{gamesyr} + \beta_3 \cdot \texttt{bavg} + \beta_4 \cdot \texttt{hrunsyr} + \beta_5 \cdot \texttt{rbisyr} + u. \quad (4.9)$$

We want to test whether the performance measures batting average (`bavg`), home runs per year (`hrunsyr`), and runs batted in per year (`rbisyr`) have an impact on the salary once we control for the number of years as an active player (`years`) and the number of games played per year (`gamesyr`). So we state our null hypothesis as $H_0 : \beta_3 = 0, \beta_4 = 0, \beta_5 = 0$ versus $H_1 : H_0$ is false, i.e. at least one of the performance measures matters.

The test statistic of the F test is based on the relative difference between the sum of squared residuals in the general (unrestricted) model and a restricted model in which the hypotheses are imposed SSR_{ur} and SSR_r, respectively. In our example, the restricted model is one in which `bavg`, `hrunsyr`, and `rbisyr` are excluded as regressors. If both models involve the same dependent variable, it can also be written in terms of the coefficient of determination in the unrestricted and the restricted model R^2_{ur} and R^2_r, respectively:

$$F = \frac{\text{SSR}_r - \text{SSR}_{ur}}{\text{SSR}_{ur}} \cdot \frac{n-k-1}{q} = \frac{R^2_{ur} - R^2_r}{1 - R^2_{ur}} \cdot \frac{n-k-1}{q}, \quad (4.10)$$

where q is the number of restrictions (in our example, $q = 3$). Intuitively, if the null hypothesis is correct, then imposing it as a restriction will not lead to a significant drop in the model fit and the F test statistic should be relatively small. It can be shown that under the CLM assumptions and the null hypothesis, the statistic has an F distribution with the numerator degrees of freedom equal to q and the denominator degrees of freedom of $n - k - 1$. Given a significance level α, we will reject H_0 if $F > c$, where the critical value c is the $1 - \alpha$ quantile of the relevant $F_{q,n-k-1}$ distribution. In our example, $n = 353, k = 5, q = 3$. So with $\alpha = 1\%$, the critical value is 3.84 and can be calculated using the **`f.ppf`** function in **`scipy.stats`** as

```
f.ppf(1 - 0.01, 3, 347)
```

Script 4.6 (`F-Test.py`) shows the calculations for this example. The result is F = 9.55 > 3.84, so we clearly reject H_0. We also calculate the p value for this test. It is $p = 4.47 \cdot 10^{-06} = 0.00000447$, so we reject H_0 for any reasonable significance level.

Script 4.6: **F-Test.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf
import scipy.stats as stats

mlb1 = woo.dataWoo('mlb1')
n = mlb1.shape[0]

# unrestricted OLS regression:
reg_ur = smf.ols(
    formula='np.log(salary) ~ years + gamesyr + bavg + hrunsyr + rbisyr',
    data=mlb1)
fit_ur = reg_ur.fit()
r2_ur = fit_ur.rsquared
print(f'r2_ur: {r2_ur}\n')

# restricted OLS regression:
reg_r = smf.ols(formula='np.log(salary) ~ years + gamesyr', data=mlb1)
fit_r = reg_r.fit()
r2_r = fit_r.rsquared
print(f'r2_r: {r2_r}\n')

# F statistic:
fstat = (r2_ur - r2_r) / (1 - r2_ur) * (n - 6) / 3
print(f'fstat: {fstat}\n')

# CV for alpha=1% using the F distribution with 3 and 347 d.f.:
cv = stats.f.ppf(1 - 0.01, 3, 347)
print(f'cv: {cv}\n')

# p value = 1-cdf of the appropriate F distribution:
fpval = 1 - stats.f.cdf(fstat, 3, 347)
print(f'fpval: {fpval}\n')
```

Output of Script 4.6: **F-Test.py**

```
r2_ur: 0.6278028485187442

r2_r: 0.5970716339066895

fstat: 9.550253521951914

cv: 3.838520048496057

fpval: 4.473708139829391e-06
```

It should not be surprising that there is a more convenient way to do this. The module **statsmodels** provides a command **f_test** which is well suited for these kinds of tests. Given the object with regression results, for example **results**, an F test is conducted with

```
hypotheses = ['var_name1 = 0', 'var_name2 = 0', ...]
ftest = results.f_test(hypotheses)
```

where **hypotheses** collects null hypothesis to be tested. It is a list of length q where each restriction is described as a text in which the variable name takes the place of its parameter. In our example, H_0 is that the three parameters of `bavg`, `hrunsyr`, and `rbisyr` are all equal to zero, which translates as **hypotheses = ['bavg = 0', 'hrunsyr = 0', 'rbisyr = 0']**. Script 4.7 (`F-Test-Automatic.py`) implements this for the same test as the manual calculations done in Script 4.6 (`F-Test.py`) and results in exactly the same F statistic and p value.

Script 4.7: F-Test-Automatic.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

mlb1 = woo.dataWoo('mlb1')

# OLS regression:
reg = smf.ols(
    formula='np.log(salary) ~ years + gamesyr + bavg + hrunsyr + rbisyr',
    data=mlb1)
results = reg.fit()

# automated F test:
hypotheses = ['bavg = 0', 'hrunsyr = 0', 'rbisyr = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Output of Script 4.7: F-Test-Automatic.py

```
fstat: 9.550253521951783

fpval: 4.473708139839581e-06
```

This function can also be used to test more complicated null hypotheses. For example, suppose a sports reporter claims that the batting average plays no role and that the number of home runs has twice the impact as the number of runs batted in. This translates (using variable names instead of numbers as subscripts) as $H_0 : \beta_{bavg} = 0, \ \beta_{hrunsyr} = 2 \cdot \beta_{rbisyr}$. For *Python*, we translate it as **hypotheses = ['bavg = 0', 'hrunsyr = 2*rbisyr']**. The output of Script 4.8 (`F-Test-Automatic2.py`) shows the results of this test. The p value is $p = 0.6$, so we cannot reject H_0.

Script 4.8: `F-Test-Automatic2.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

mlb1 = woo.dataWoo('mlb1')

# OLS regression:
reg = smf.ols(
    formula='np.log(salary) ~ years + gamesyr + bavg + hrunsyr + rbisyr',
    data=mlb1)
results = reg.fit()

# automated F test:
hypotheses = ['bavg = 0', 'hrunsyr = 2*rbisyr']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Output of Script 4.8: `F-Test-Automatic2.py`

```
fstat: 0.5117822576247235

fpval: 0.5998780329146685
```

Both the most important and the most straightforward F test is the one for **overall significance**. The null hypothesis is that all parameters except for the constant are equal to zero. If this null hypothesis holds, the regressors do not have any joint explanatory power for y. The results of such a test are automatically included in the upper part of the **`summary`** output as **`F-statistic`** (F statistic) and **`Prob(F-statistic)`** (p value). As an example, see Script 4.5 (`Example-4-8.py`). The null hypothesis that neither the sales nor the margin have any relation to R&D spending is clearly rejected with an F statistic of 162.2 and a p value smaller than 10^{-15}.

5. Multiple Regression Analysis: OLS Asymptotics

Asymptotic theory allows us to relax some assumptions needed to derive the sampling distribution of estimators if the sample size is large enough. For running a regression in a software package, it does not matter whether we rely on stronger assumptions or on asymptotic arguments. So we don't have to learn anything new regarding the implementation.

Instead, this chapter aims to improve on our intuition regarding the workings of asymptotics by looking at some simulation exercises in Section 5.1. Section 5.2 briefly discusses the implementation of the regression-based Lagrange multiplier (LM) test presented by Wooldridge (2019, Section 5.2).

5.1. Simulation Exercises

In Section 2.7, we already used Monte Carlo Simulation methods to study the mean and variance of OLS estimators under the assumptions SLR.1–SLR.5. Here, we will conduct similar experiments but will look at the whole sampling distribution of OLS estimators similar to Section 1.9.2 where we demonstrated the central limit theorem for the sample mean. Remember that the sampling distribution is important since confidence intervals, t and F tests and other tools of inference rely on it.

Theorem 4.1 of Wooldridge (2019) gives the normal distribution of the OLS estimators (conditional on the regressors) based on assumptions MLR.1 through MLR.6. In contrast, Theorem 5.2 states that *asymptotically*, the distribution is normal by assumptions MLR.1 through MLR.5 only. Assumption MLR.6 – the normal distribution of the error terms – is not required if the sample is large enough to justify asymptotic arguments.

In other words: In small samples, the parameter estimates have a normal sampling distribution only if

- the error terms are normally distributed and
- we condition on the regressors.

To see how this works out in practice, we set up a series of simulation experiments. Section 5.1.1 simulates a model consistent with MLR.1 through MLR.6 and keeps the regressors fixed. Theory suggests that the sampling distribution of $\hat{\beta}$ is normal, independent of the sample size. Section 5.1.2 simulates a violation of assumption MLR.6. Normality of $\hat{\beta}$ only holds asymptotically, so for small sample sizes we suspect a violation. Finally, we will look closer into what "conditional on the regressors" means and simulate a (very plausible) violation of this in Section 5.1.3.

5.1.1. Normally Distributed Error Terms

Script 5.1 (`Sim-Asy-OLS-norm.py`) draws 10 000 samples of a given size (which has to be stored in variable **`n`** before) from a population that is consistent with assumptions MLR.1 through MLR.6. The error terms are specified to be standard normal. The slope estimate $\hat{\beta}_1$ is stored for each of the generated samples in the array **b1**. For a more detailed discussion of the implementation, see Section 2.7.2 where a very similar simulation exercise is introduced.

Script 5.1: Sim-Asy-OLS-norm.py

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(ex, sx, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u (std. normal):
    u = stats.norm.rvs(0, 1, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate conditional OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b1[i] = results.params['x']
```

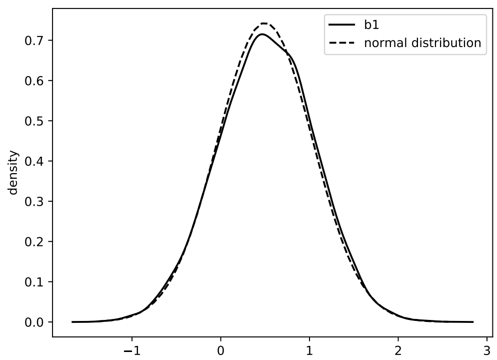
This code was run for different sample sizes. The density estimate together with the corresponding normal density are shown in Figure 5.1. Not surprisingly, all distributions look very similar to the normal distribution – this is what Theorem 4.1 predicted. Note that the fact that the sampling variance decreases as n rises is only obvious if we pay attention to the different scales of the axes.

5.1.2. Non-Normal Error Terms

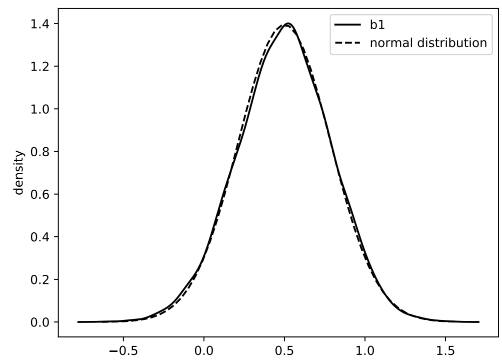
The next step is to simulate a violation of assumption MLR.6. In order to implement a rather drastic violation of the normality assumption similar to Section 1.9.2, we implement a "standardized" χ^2 distribution with one degree of freedom. More specifically, let v be distributed as $\chi^2_{[1]}$. Because this distribution has a mean of 1 and a variance of 2, the error term $u = \frac{v-1}{\sqrt{2}}$ has a mean of 0 and a variance of 1. This simplifies the comparison to the exercise with the standard normal errors above. Figure 5.2 plots the density functions of the standard normal distribution used above and the "standardized" χ^2 distribution. Both have a mean of 0 and a variance of 1 but very different shapes.

Script 5.2 (`Sim-Asy-OLS-chisq.py`) implements a simulation of this model and is listed in the appendix (p. 350). The only line of code we changed compared to the previous Script 5.1

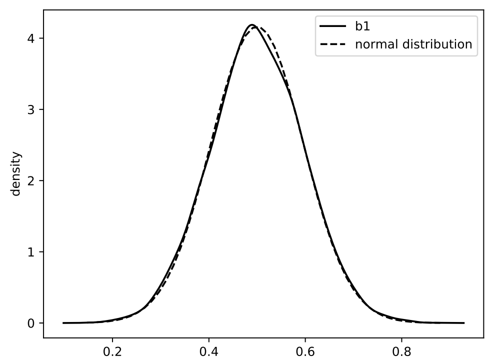
Figure 5.1. Density of $\hat{\beta}_1$ with Different Sample Sizes: Normal Error Terms



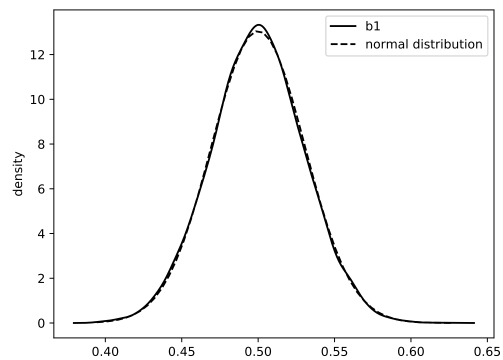
(a) $n = 5$



(b) $n = 10$

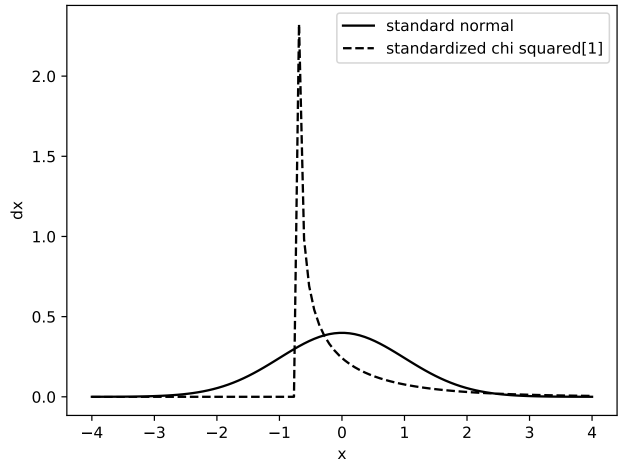


(c) $n = 100$



(d) $n = 1\,000$

Figure 5.2. Density Functions of the Simulated Error Terms



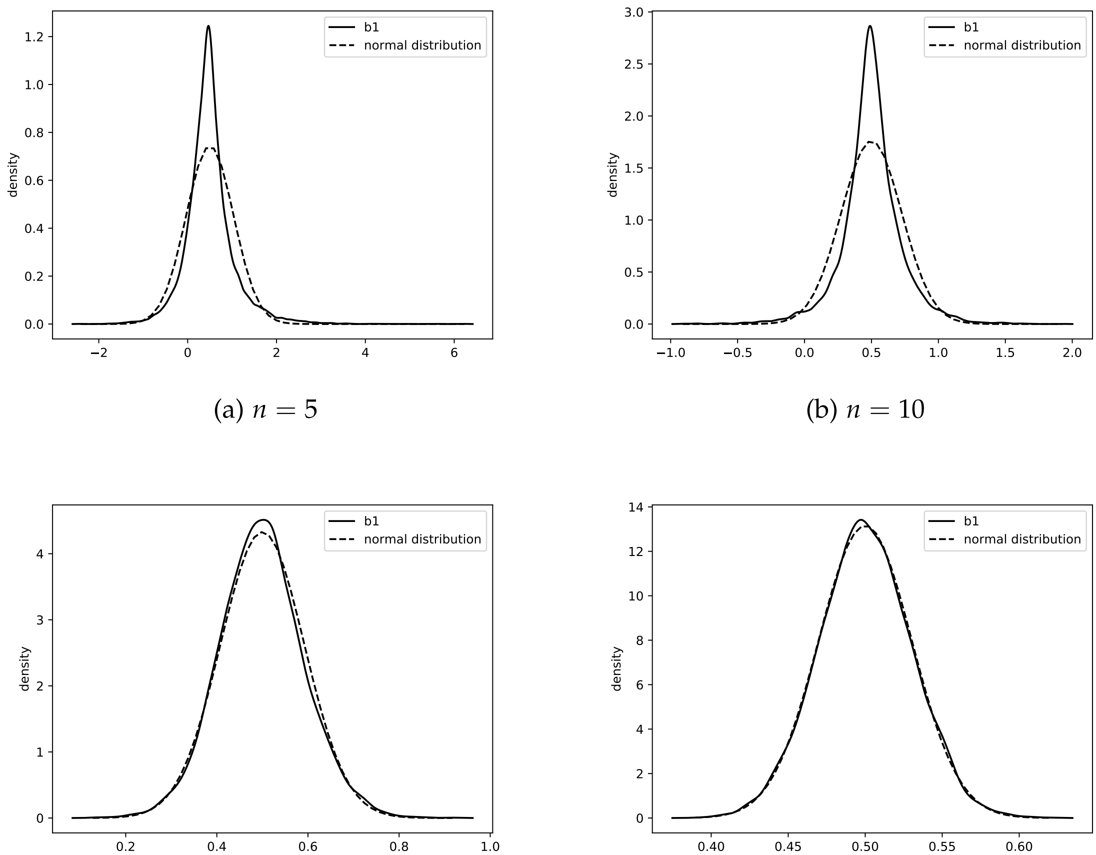
(`Sim-Asy-OLS-norm.py`) is the sampling of **u** where we replace drawing from a standard normal distribution using **u = stats.norm.rvs(0, 1, size=n)** with sampling from the standardized $\chi^2_{[1]}$ distribution with

```
u = (stats.chi2.rvs(1, size=n) - 1) / np.sqrt(2)
```

For each of the same sample sizes used above, we again estimate the slope parameter for 10 000 samples. The densities of $\hat{\beta}_1$ are plotted in Figure 5.3 together with the respective normal distributions with the corresponding variances. For the small sample sizes, the deviation from the normal distribution is strong. Note that the dashed normal distributions have the same mean and variance. The main difference is the kurtosis which is larger than 8 in the simulations for $n = 5$ compared to the normal distribution for which the kurtosis is equal to 3.

For larger sample sizes, the sampling distribution of $\hat{\beta}_1$ converges to the normal distribution. For $n = 100$, the difference is much smaller but still discernible. For $n = 1\,000$, it cannot be detected anymore in our simulation exercise. How large the sample needs to be depends among other things on the severity of the violations of MLR.6. If the distribution of the error terms is not as extremely non-normal as in our simulations, smaller sample sizes like the rule of thumb $n = 30$ might suffice for valid asymptotics.

Figure 5.3. Density of $\hat{\beta}_1$ with Different Sample Sizes: Non-Normal Error Terms



(a) $n = 5$ (b) $n = 10$

(c) $n = 100$ (d) $n = 1\,000$

5.1.3. (Not) Conditioning on the Regressors

There is a more subtle difference between the finite-sample results regarding the variance (Theorem 3.2) and distribution (Theorem 4.1) on one hand and the corresponding asymptotic results (Theorem 5.2). The former results describe the sampling distribution "conditional on the sample values of the independent variables". This implies that as we draw different samples, the values of the regressors $x_1, \ldots, x_k$ remain the same and only the error terms and dependent variables change.

In our previous simulation exercises in Scripts like 2.16 (`SLR-Sim-Model-Condx.py`), 5.1 (`Sim-Asy-OLS-norm.py`), and 5.2 (`Sim-Asy-OLS-chisq.py`), this is implemented by making random draws of x outside of the simulation loop. This is a realistic description of how data is generated only in some simple experiments: The experimenter chooses the regressors for the sample, conducts the experiment and measures the dependent variable.

In most applications we are concerned with, this is an unrealistic description of how we obtain our data. If we draw a sample of individuals, both their dependent and independent variables differ across samples. In these cases, the distribution "conditional on the sample values of the independent variables" can only serve as an approximation of the actual distribution with varying regressors. For large samples, this distinction is irrelevant and the asymptotic distribution is the same.

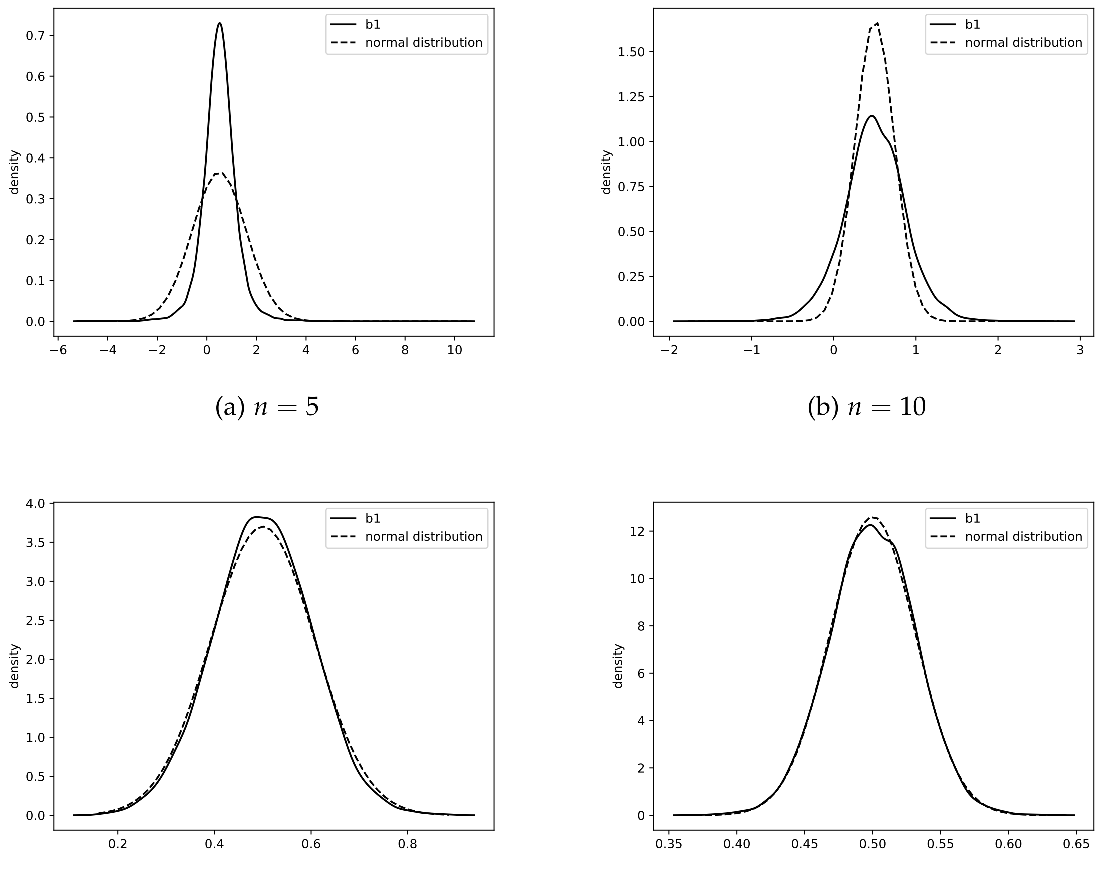
Let's see how this plays out in an example. Script 5.3 (`Sim-Asy-OLS-uncond.py`) differs from Script 5.1 (`Sim-Asy-OLS-norm.py`) only by moving the generation of the regressors into the loop in which the 10 000 samples are generated. This is inconsistent with Theorem 4.1, so for small samples, we don't know the distribution of $\hat{\beta}_1$. Theorem 5.2 is applicable, so for (very) large samples, we know that the estimator is normally distributed.

Figure 5.4 shows the distribution of the 10 000 estimates generated by Script 5.3 (`Sim-Asy-OLS-uncond.py`) for $n = 5, 10, 100$, and 1 000. As we expected from theory, the distribution is (close to) normal for large samples. For small samples, it deviates quite a bit. The kurtosis is 8.7 for a sample size of $n = 5$ which is far away from the kurtosis of 3 of a normal distribution.

Script 5.3: `Sim-Asy-OLS-uncond.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = np.empty(r)

# repeat r times:
for i in range(r):
    # draw a sample of x, varying over replications:
    x = stats.norm.rvs(ex, sx, size=n)

    # draw a sample of u (std. normal):
    u = stats.norm.rvs(0, 1, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate unconditional OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b1[i] = results.params['x']
```

Figure 5.4. Density of $\hat{\beta}_1$ with Different Sample Sizes: Varying Regressors



(a) $n = 5$

(b) $n = 10$

(c) $n = 100$

(d) $n = 1\,000$

5.2. LM Test

As an alternative to the F tests discussed in Section 4.3, LM tests for the same sort of hypotheses can be very useful with large samples. In the linear regression setup, the test statistic is

$$LM = n \cdot R^2_{\tilde{u}},$$

where n is the sample size and $R^2_{\tilde{u}}$ is the usual R^2 statistic in a regression of the residual $\tilde{u}$ from the restricted model on the unrestricted set of regressors. Under the null hypothesis, it is asymptotically distributed as χ^2_q with q denoting the number of restrictions. Details are given in Wooldridge (2019, Section 5.2).

The implementation in **`statsmodels`** is straightforward if we remember that the residuals can be obtained with the **`resid`** attribute.

Wooldridge, Example 5.3: Economic Model of Crime

We analyze the same data on the number of arrests as in Example 3.5. The unrestricted regression model equation is

$$\texttt{narr86} = \beta_0 + \beta_1 \texttt{pcnv} + \beta_2 \texttt{avgsen} + \beta_3 \texttt{tottime} + \beta_4 \texttt{ptime86} + \beta_5 \texttt{qemp86} + u.$$

The dependent variable `narr86` reflects the number of times a man was arrested and is explained by the proportion of prior arrests (`pcnv`), previous average sentences (`avgsen`), the time spend in prison before 1986 (`tottime`), the number of months in prison in 1986 (`ptime86`), and the number of quarters unemployed in 1986 (`qemp86`).
The joint null hypothesis is

$$H_0 : \beta_2 = \beta_3 = 0,$$

so the restricted set of regressors excludes `avgsen` and `tottime`. Script 5.4 (`Example-5-3.py`) shows an implementation of this LM test. The restricted model is estimated and its residuals **`utilde`**$=\tilde{u}$ are calculated. They are regressed on the unrestricted set of regressors. The R^2 from this regression is 0.001494, so the LM test statistic is calculated to be around $LM = 0.001494 \cdot 2725 = 4.071$. This is smaller than the critical value for a significance level of $\alpha = 10\%$, so we do not reject the null hypothesis. We can also easily calculate the p value using the χ^2 CDF **`chi2.cdf`**. It turns out to be 0.1306.
The same hypothesis can be tested using the F test presented in Section 4.3 using the command **`f_test`**. In this example, it delivers the same p value up to three digits.

Script 5.4: `Example-5-3.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

crime1 = woo.dataWoo('crime1')

# 1. estimate restricted model:
reg_r = smf.ols(formula='narr86 ~ pcnv + ptime86 + qemp86', data=crime1)
fit_r = reg_r.fit()
r2_r = fit_r.rsquared
print(f'r2_r: {r2_r}\n')

# 2. regression of residuals from restricted model:
crime1['utilde'] = fit_r.resid
reg_LM = smf.ols(formula='utilde ~ pcnv + ptime86 + qemp86 + avgsen + tottime',
                 data=crime1)
fit_LM = reg_LM.fit()
r2_LM = fit_LM.rsquared
print(f'r2_LM: {r2_LM}\n')

# 3. calculation of LM test statistic:
LM = r2_LM * fit_LM.nobs
print(f'LM: {LM}\n')

# 4. critical value from chi-squared distribution, alpha=10%:
cv = stats.chi2.ppf(1 - 0.10, 2)
print(f'cv: {cv}\n')

# 5. p value (alternative to critical value):
pval = 1 - stats.chi2.cdf(LM, 2)
print(f'pval: {pval}\n')

# 6. compare to F-test:
reg = smf.ols(formula='narr86 ~ pcnv + ptime86 + qemp86 + avgsen + tottime',
              data=crime1)
results = reg.fit()
hypotheses = ['avgsen = 0', 'tottime = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue
print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Output of Script 5.4: `Example-5-3.py`

```
r2_r: 0.04132330770123016

r2_LM: 0.0014938456737880745

LM: 4.070729461072503

cv: 4.605170185988092

pval: 0.13063282803261256

fstat: 2.0339215584351407

fpval: 0.13102048172760739
```

6. Multiple Regression Analysis: Further Issues

In this chapter, we cover some issues regarding the implementation of regression analyses. Section 6.1 discusses more flexible specification of regression equations such as variable scaling, standardization, polynomials and interactions. They can be conveniently included in the **`formula`** and used in the **`statsmodels`** OLS estimation. Section 6.2 is concerned with predictions and their confidence and prediction intervals.

6.1. Model Formulae

If we run a regression in **`statsmodels`** using a syntax like

```
smf.ols('y ~ x1 + x2 + x3', data=sample)
```

the expression **`y ~ x1 + x2 + x3`** is referred to as a model **`formula`**. It is a compact symbolic way to describe our regression equation. The dependent variable is separated from the regressors by a "**`~`**" and the regressors are separated by a "**`+`**" indicating that they enter the equation in a linear fashion. A constant is added by default. Such formulae can be specified in more complex ways to indicate different kinds of regression equations. We will cover the most important ones in this section.

6.1.1. Data Scaling: Arithmetic Operations Within a Formula

Wooldridge (2019) discusses how different scaling of the variables in the model affects the parameter estimates and other statistics in Section 6.1. As an example, a model relating the birth weight to cigarette smoking of the mother during pregnancy and the family income. The basic model equation is

$$\texttt{bwght} = \beta_0 + \beta_1 \texttt{cigs} + \beta_2 \texttt{faminc} + u \tag{6.1}$$

which translates into formula syntax as **`bwght ~ cigs + faminc`**.

If we want to measure the weight in pounds rather than ounces, there are two ways to implement different rescaling in *Python*. We can

- Define a different variable like **`bwghtlbs = bwght/16`** and use this variable in the formula: **`bwghtlbs ~ cigs + faminc`**
- Specify this rescaling directly in the formula: **`I(bwght/16) ~ cigs + faminc`**

The latter approach can be more convenient. Note that the **`I(...)`** brackets describe any parts of the formula in which we specify arithmetic transformations.

If we want to measure the number of cigarettes smoked per day in packs, we could again define a new variable **`packs = cigs/20`** and use it as a regressor or simply specify the formula **`bwght ~ I(cigs/20) + faminc`**. Here, the importance to use the **`I`** function is easy to see. If we specified the formula **bwght ~ I(cigs/20 + faminc)** instead, we would have a (nonsense) model with only one regressor: the sum of the packs smoked and the income.

Script 6.1 (`Data-Scaling.py`) demonstrates these features. As discussed in Wooldridge (2019, Section 6.1), dividing the dependent variable by 16 changes all coefficients by the same factor $\frac{1}{16}$ and dividing a regressor by 20 changes its coefficient by the factor 20. Other statistics like R^2 are unaffected.

Script 6.1: Data-Scaling.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

bwght = woo.dataWoo('bwght')

# regress and report coefficients:
reg = smf.ols(formula='bwght ~ cigs + faminc', data=bwght)
results = reg.fit()

# weight in pounds, manual way:
bwght['bwght_lbs'] = bwght['bwght'] / 16
reg_lbs = smf.ols(formula='bwght_lbs ~ cigs + faminc', data=bwght)
results_lbs = reg_lbs.fit()

# weight in pounds, direct way:
reg_lbs2 = smf.ols(formula='I(bwght/16) ~ cigs + faminc', data=bwght)
results_lbs2 = reg_lbs2.fit()

# packs of cigarettes:
reg_packs = smf.ols(formula='bwght ~ I(cigs/20) + faminc', data=bwght)
results_packs = reg_packs.fit()

# compare results:
table = pd.DataFrame({'b': round(results.params, 4),
                      'b_lbs': round(results_lbs.params, 4),
                      'b_lbs2': round(results_lbs2.params, 4),
                      'b_packs': round(results_packs.params, 4)})
print(f'table: \n{table}\n')
```

Output of Script 6.1: Data-Scaling.py

```
table:
                     b    b_lbs  b_lbs2   b_packs
I(cigs / 20)       NaN      NaN     NaN   -9.2682
Intercept     116.9741   7.3109  7.3109  116.9741
cigs           -0.4634  -0.0290 -0.0290       NaN
faminc          0.0928   0.0058  0.0058    0.0928
```

6.1.2. Standardization: Beta Coefficients

A specific arithmetic operation is the standardization. A variable is standardized by subtracting its mean and dividing by its standard deviation. For example, the standardized dependent variable y and regressor x_1 are

$$z_y = \frac{y - \bar{y}}{\text{sd}(y)} \quad \text{and} \quad z_{x_1} = \frac{x_1 - \bar{x}_1}{\text{sd}(x_1)}. \tag{6.2}$$

If the regression model only contains standardized variables, the coefficients have a special interpretation. They measure by how many *standard deviations* y changes as the respective independent variable increases by *one standard deviation*. Inconsistent with the notation used here, they are sometimes referred to as beta coefficients.

In *Python*, we can use the same type of arithmetic transformations as in Section 6.1.1 to subtract the mean and divide by the standard deviation. It can be done more conveniently by defining and using a function **scale** directly for all variables we want to standardize. Defining a function was introduced in Section 1.8.3 and Script 6.2 (`Example-6-1.py`) demonstrates the use of **scale** in the context of a regression.

Wooldridge, Example 6.1: Effects of Pollution on Housing Prices

We are interested in how air pollution (`nox`) and other neighborhood characteristics affect the value of a house. A model using standardization for all variables is expressed in a formula as

price_sc ~ 0 + nox_sc + crime_sc + rooms_sc + dist_sc + stratio_sc

with **variable_sc** denoting the scaled version of **variable**. The output of Script 6.2 (`Example-6-1.py`) shows the parameter estimates of this model. The house price drops by 0.34 standard deviations as the air pollution increases by one standard deviation.

Script 6.2: Example-6-1.py

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf


# define a function for the standardization:
def scale(x):
    x_mean = np.mean(x)
    x_var = np.var(x, ddof=1)
    x_scaled = (x - x_mean) / np.sqrt(x_var)
    return x_scaled


# standardize and estimate:
hprice2 = woo.dataWoo('hprice2')
hprice2['price_sc'] = scale(hprice2['price'])
hprice2['nox_sc'] = scale(hprice2['nox'])
hprice2['crime_sc'] = scale(hprice2['crime'])
hprice2['rooms_sc'] = scale(hprice2['rooms'])
hprice2['dist_sc'] = scale(hprice2['dist'])
hprice2['stratio_sc'] = scale(hprice2['stratio'])

reg = smf.ols(
    formula='price_sc ~ 0 + nox_sc + crime_sc + rooms_sc + dist_sc + stratio_sc',
    data=hprice2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 6.2: `Example-6-1.py`

```
table:
                 b      se        t  pval
nox_sc     -0.3404  0.0445  -7.6511   0.0
crime_sc   -0.1433  0.0307  -4.6693   0.0
rooms_sc    0.5139  0.0300  17.1295   0.0
dist_sc    -0.2348  0.0430  -5.4641   0.0
stratio_sc -0.2703  0.0299  -9.0274   0.0
```

6.1.3. Logarithms

We have already seen in Section 2.4 that we can include the **`numpy`** function **`log`** directly in formulas to represent logarithmic and semi-logarithmic models. A simple example of a partially logarithmic model and its formula would be

$$\log(y) = \beta_0 + \beta_1 \log(x_1) + \beta_2 x_2 + u \tag{6.3}$$

which can be expressed as **`np.log(y) ~ np.log(x1) + x2`**.

Script 6.3 (`Formula-Logarithm.py`) shows this again for the house price example. As the air pollution `nox` increases by *one percent*, the house price drops by about 0.72 *percent*. As the number of rooms increases by *one*, the value of the house increases by roughly 30.6%. Wooldridge (2019, Section 6.2) discusses how the latter value is only an approximation and the actual estimated effect is $(\exp(0.306) - 1) = 0.358$ which is 35.8%.

Script 6.3: `Formula-Logarithm.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

hprice2 = woo.dataWoo('hprice2')

reg = smf.ols(formula='np.log(price) ~ np.log(nox) + rooms', data=hprice2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 6.3: `Formula-Logarithm.py`

```
table:
                  b      se        t  pval
Intercept    9.2337  0.1877  49.1835   0.0
np.log(nox) -0.7177  0.0663 -10.8182   0.0
rooms        0.3059  0.0190  16.0863   0.0
```

6.1.4. Quadratics and Polynomials

Specifying quadratic terms or higher powers of regressors can be a useful way to make a model more flexible by allowing the partial effects or (semi-)elasticities to decrease or increase with the value of the regressor.

Instead of creating additional variables containing the squared value of a regressor, in *Python* we can simply add **I(x**2)** to a formula. Higher order terms are specified accordingly. A simple cubic model and its corresponding formula are

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + u \tag{6.4}$$

which translates to **y ~ x + I(x**2) + I(x**3)** in formula syntax.

For nonlinear models like this, it is often useful to get a graphical illustration of the effects. Section 6.2.2 shows how to conveniently generate these.

Wooldridge, Example 6.2: Effects of Pollution on Housing Prices

This example of Wooldridge (2019) demonstrates the combination of logarithmic and quadratic specifications. The model for house prices is

$$\log(\texttt{price}) = \beta_0 + \beta_1 \log(\texttt{nox}) + \beta_2 \log(\texttt{dist}) + \beta_3 \texttt{rooms} + \beta_4 \texttt{rooms}^2 + \beta_5 \texttt{stratio} + u.$$

Script 6.4 (`Example-6-2.py`) implements this model and presents detailed results including t statistics and their p values. The quadratic term of `rooms` has a significantly positive coefficient $\hat{\beta}_4$ implying that the semi-elasticity increases with more rooms. The negative coefficient for `rooms` and the positive coefficient for $\texttt{rooms}^2$ imply that for "small" numbers of rooms, the price *decreases* with the number of rooms and for "large" values, it *increases*. The number of rooms implying the smallest price can be found as[1]

$$\texttt{rooms}^* = \frac{-\beta_3}{2\beta_4} \approx 4.4.$$

Script 6.4: Example-6-2.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

hprice2 = woo.dataWoo('hprice2')

reg = smf.ols(
    formula='np.log(price) ~ np.log(nox)+np.log(dist)+rooms+I(rooms**2)+stratio',
    data=hprice2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

[1] We need to find $\texttt{rooms}^*$ to minimize $\beta_3\texttt{rooms} + \beta_4\texttt{rooms}^2$. Setting the first derivative $\beta_3 + 2\beta_4\texttt{rooms}$ equal to zero and solving for `rooms` delivers the result.

Output of Script 6.4: Example-6-2.py

```
table:
                     b      se        t    pval
Intercept      13.3855  0.5665  23.6295  0.0000
np.log(nox)    -0.9017  0.1147  -7.8621  0.0000
np.log(dist)   -0.0868  0.0433  -2.0051  0.0455
rooms          -0.5451  0.1655  -3.2946  0.0011
I(rooms ** 2)   0.0623  0.0128   4.8623  0.0000
stratio        -0.0476  0.0059  -8.1293  0.0000
```

6.1.5. Hypothesis Testing

A natural question to ask is whether a regressor has additional statistically significant explanatory power in a regression model, given all the other regressors. In simple model specifications, this question can be answered by a simple t test, so the results for all regressors are available with a quick look at the standard regression table.[2] When working with polynomials or other specifications, the influence of one regressor is captured by several parameters. We can test its significance with an F test of the joint null hypothesis that all of these parameters are equal to zero. As an example, let's revisit Example 6.2:

$$\log(\texttt{price}) = \beta_0 + \beta_1 \log(\texttt{nox}) + \beta_2 \log(\texttt{dist}) + \beta_3 \texttt{rooms} + \beta_4 \texttt{rooms}^2 + \beta_5 \texttt{stratio} + u$$

The significance of `rooms` can be assessed with an F test of $H_0 : \beta_3 = \beta_4 = 0$. As discussed in Section 4.3, such a test can be performed with the command **`f_test`** from the module **`statsmodels`**. This is shown in Script 6.5 (`Example-6-2-Ftest.py`).

Script 6.5: Example-6-2-Ftest.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

hprice2 = woo.dataWoo('hprice2')
n = hprice2.shape[0]

reg = smf.ols(
    formula='np.log(price) ~ np.log(nox)+np.log(dist)+rooms+I(rooms**2)+stratio',
    data=hprice2)
results = reg.fit()

# implemented F test for rooms:
hypotheses = ['rooms = 0', 'I(rooms ** 2) = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Output of Script 6.5: Example-6-2-Ftest.py

```
fstat: 110.4187819267064

fpval: 1.9193250019375434e-40
```

[2] Section 4.1 discusses t tests.

6.1.6. Interaction Terms

Models with interaction terms allow the effect of one variable x_1 to depend on the value of another variable x_2. A simple model including an interaction term would be

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + u. \tag{6.5}$$

Of course, we can implement this in *Python* by defining a new variable containing the product of the two regressors. But again, a direct specification in the model formula is more convenient. The expression **x1:x2** within a formula adds the interaction term $x_1 x_2$. Even more conveniently, **x1*x2** adds not only the interaction but also both original variables allowing for a very concise syntax. So the model in Equation 6.5 can be specified in *Python* as either of the two formulas:

y ~ x1 + x2 + x1:x2 $\Leftrightarrow$ **y ~ x1*x2**

If one variable x_1 is interacted with a set of other variables, they can be grouped by parentheses to allow for a compact syntax. For example, the shortest way to express the model equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1 x_2 + \beta_5 x_1 x_3 + u \tag{6.6}$$

in *Python* syntax is **y ~ x1*(x2 + x3)**.

Wooldridge, Example 6.3: Effects of Attendance on Final Exam Performance

This example analyzes a model including a standardized dependent variable, quadratic terms and an interaction. Standardized scores in the final exam are explained by class attendance, prior performance and an interaction term:

$$\texttt{stndfnl} = \beta_0 + \beta_1 \texttt{atndrte} + \beta_2 \texttt{priGPA} + \beta_3 \texttt{ACT} + \beta_4 \texttt{priGPA}^2 + \beta_5 \texttt{ACT}^2 + \beta_6 \texttt{priGPA} \cdot \texttt{atndrte} + u$$

Script 6.6 (`Example-6-3.py`) estimates this model.
The effect of attending classes is

$$\frac{\partial \texttt{stndfnl}}{\partial \texttt{atndrte}} = \beta_1 + \beta_6 \texttt{priGPA}.$$

For the average $\overline{\texttt{priGPA}} = 2.59$, the script estimates this partial effect to be around 0.0078. It tests the null hypothesis that this effect is zero using a simple F test, see Section 4.3. With a p value of 0.0034, this hypothesis can be rejected at all common significance levels.

Script 6.6: `Example-6-3.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

attend = woo.dataWoo('attend')
n = attend.shape[0]

reg = smf.ols(formula='stndfnl ~ atndrte*priGPA + ACT + I(priGPA**2) + I(ACT**2)',
              data=attend)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
```

```
print(f'table: \n{table}\n')

# estimate for partial effect at priGPA=2.59:
b = results.params
partial_effect = b['atndrte'] + 2.59 * b['atndrte:priGPA']
print(f'partial_effect: {partial_effect}\n')

# F test for partial effect at priGPA=2.59:
hypotheses = 'atndrte + 2.59 * atndrte:priGPA = 0'
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Output of Script 6.6: `Example-6-3.py`

```
table:
                     b      se       t    pval
Intercept       2.0503  1.3603  1.5072  0.1322
atndrte        -0.0067  0.0102 -0.6561  0.5120
priGPA         -1.6285  0.4810 -3.3857  0.0008
atndrte:priGPA  0.0056  0.0043  1.2938  0.1962
ACT            -0.1280  0.0985 -1.3000  0.1940
I(priGPA ** 2)  0.2959  0.1010  2.9283  0.0035
I(ACT ** 2)     0.0045  0.0022  2.0829  0.0376

partial_effect: 0.007754572228608965

fstat: 8.632581056740811

fpval: 0.003414992399585439
```

6.2. Prediction

In this section, we are concerned with predicting the value of the dependent variable y given certain values of the regressors $x_1, \ldots, x_k$. If these are the regressor values in our estimation sample, we called these predictions "fitted values" and discussed their calculation in Section 2.2. Now, we generalize this to arbitrary values and add standard errors, confidence intervals, and prediction intervals.

6.2.1. Confidence and Prediction Intervals for Predictions

Confidence intervals reflect the uncertainty about the *expected value* of the dependent variable given values of the regressors. If we are interested in predicting the college GPA of an *individual*, prediction intervals account for the additional uncertainty regarding the unobserved characteristics reflected by the error term u.

Given a model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + u \tag{6.7}$$

we are interested in the expected value of y given the regressors take specific values $c_1, c_2, \ldots, c_k$:

$$\theta_0 = \mathrm{E}(y|x_1 = c_1, \ldots, x_k = c_k) = \beta_0 + \beta_1 c_1 + \beta_2 c_2 + \cdots + \beta_k c_k. \tag{6.8}$$

The natural point estimates are

$$\hat{\theta}_0 = \hat{\beta}_0 + \hat{\beta}_1 c_1 + \hat{\beta}_2 c_2 + \cdots + \hat{\beta}_k c_k \tag{6.9}$$

and can readily be obtained once the parameter estimates $\hat{\beta}_0, \ldots, \hat{\beta}_k$ are calculated.

Standard errors and confidence intervals are less straightforward to compute. Wooldridge (2019, Section 6.4) suggests a smart way to obtain these from a modified regression. **statsmodels** provides an even simpler and more convenient approach.

The method **predict** automatically calculates $\hat{\theta}_0$. The method can be called on an object created by the **fit** method. Its argument is a data frame containing the values of the regressors $c_1, \ldots c_k$ of the regressors $x_1, \ldots x_k$ with the same variable names as in the data frame used for estimation. If we don't have one yet, it can for example be specified with **pandas** as

```
pd.DataFrame({'x1':[c1], 'x2':[c2],...,'xk':[ck]}, index=['newobservation1'])
```

where **x1** through **xk** are the variable names and **c1** through **ck** are the values which can also be specified as lists to get predictions at several values of the regressors. See Section 1.2.4 for more on data frames and Script 6.7 (`Predictions.py`) for an example.

Script 6.7: `Predictions.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import pandas as pd

gpa2 = woo.dataWoo('gpa2')

reg = smf.ols(formula='colgpa ~ sat + hsperc + hsize + I(hsize**2)', data=gpa2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# generate data set containing the regressor values for predictions:
cvalues1 = pd.DataFrame({'sat': [1200], 'hsperc': [30],
                        'hsize': [5]}, index=['newPerson1'])
print(f'cvalues1: \n{cvalues1}\n')

# point estimate of prediction (cvalues1):
colgpa_pred1 = results.predict(cvalues1)
print(f'colgpa_pred1: \n{colgpa_pred1}\n')

# define three sets of regressor variables:
cvalues2 = pd.DataFrame({'sat': [1200, 900, 1400, ],
                        'hsperc': [30, 20, 5], 'hsize': [5, 3, 1]},
                       index=['newPerson1', 'newPerson2', 'newPerson3'])
print(f'cvalues2: \n{cvalues2}\n')

# point estimate of prediction (cvalues2):
colgpa_pred2 = results.predict(cvalues2)
print(f'colgpa_pred2: \n{colgpa_pred2}\n')
```

Output of Script 6.7: `Predictions.py`

```
table:
                     b      se        t    pval
Intercept       1.4927  0.0753  19.8118  0.0000
sat             0.0015  0.0001  22.8864  0.0000
hsperc         -0.0139  0.0006 -24.6981  0.0000
hsize          -0.0609  0.0165  -3.6895  0.0002
I(hsize ** 2)   0.0055  0.0023   2.4056  0.0162

cvalues1:
             sat  hsperc  hsize
newPerson1  1200      30      5

colgpa_pred1:
newPerson1    2.700075
dtype: float64

cvalues2:
             sat  hsperc  hsize
newPerson1  1200      30      5
newPerson2   900      20      3
newPerson3  1400       5      1

colgpa_pred2:
newPerson1    2.700075
newPerson2    2.425282
newPerson3    3.457448
dtype: float64
```

The method **`get_prediction`** calculates not only $\hat{\theta}_0$ (i.e. the exact same predictions as the method **`predict`**), but also

- standard errors of the predictions (column **`mean_se`**),
- confidence intervals (columns **`mean_ci_lower`** and **`mean_ci_upper`**) and
- prediction intervals (columns **`obs_ci_lower`** and **`obs_ci_upper`**). Wooldridge (2019) explains how to calculate the prediction interval manually.

All you have to do is calling a second method **`summary_frame`** to provide the significance level. Script 6.8 (`Example-6-5.py`) demonstrates the procedure for $\alpha = 5\%$ and 1%.

Wooldridge, Example 6.5: Confidence Interval for Predicted College GPA

We try to predict the college GPA, for example to support the admission decisions for our college. Our regression model equation is

$$\texttt{colgpa} = \beta_0 + \beta_1 \texttt{sat} + \beta_2 \texttt{hsperc} + \beta_3 \texttt{hsize} + \beta_4 \texttt{hsize}^2 + u.$$

Script 6.8 (`Example-6-5.py`) shows the implementation of the estimation and prediction. The estimation results are stored as the variable **`results`**. The values of the regressors for which we want to do the prediction are stored in the new data frame **`cvalues2`**. Then the commands **`get_prediction`** and **`summary_frame`** are called. For an SAT score of 1200, a high school percentile of 30 and a high school size of 5 (i.e. 500 students), the predicted college GPA is 2.7. Wooldridge (2019) obtains the same value using a general but more cumbersome regression approach. We define two other types of students with different values of `sat`, `hsperc`, and `hsize` in the data frame **`cvalues2`**.
Script 6.8 (`Example-6-5.py`) also calculates the 95% and 99% confidence and prediction intervals. The object **`colgpa_PICI_95`** contains the 95% confidence interval, for example, which is reported in columns **`mean_ci_lower`** and **`mean_ci_upper`**. With 95% confidence we can say that the expected

college GPA for students with the features of the student named **newPerson1** is between 2.66 and 2.74. The object **colgpa_PICI_99** contains the 99% prediction interval, for example, which is reported in columns **obs_ci_lower** and **obs_ci_upper**. All results are the same as those manually calculated by Wooldridge (2019).

Script 6.8: Example-6-5.py

```
import wooldridge as woo
import statsmodels.formula.api as smf
import pandas as pd

gpa2 = woo.dataWoo('gpa2')

reg = smf.ols(formula='colgpa ~ sat + hsperc + hsize + I(hsize**2)', data=gpa2)
results = reg.fit()

# define three sets of regressor variables:
cvalues2 = pd.DataFrame({'sat': [1200, 900, 1400, ],
                         'hsperc': [30, 20, 5], 'hsize': [5, 3, 1]},
                        index=['newPerson1', 'newPerson2', 'newPerson3'])

# point estimates and 95% confidence and prediction intervals:
colgpa_PICI_95 = results.get_prediction(cvalues2).summary_frame(alpha=0.05)
print(f'colgpa_PICI_95: \n{colgpa_PICI_95}\n')

# point estimates and 99% confidence and prediction intervals:
colgpa_PICI_99 = results.get_prediction(cvalues2).summary_frame(alpha=0.01)
print(f'colgpa_PICI_99: \n{colgpa_PICI_99}\n')
```

Output of Script 6.8: Example-6-5.py

```
colgpa_PICI_95:
       mean   mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  obs_ci_upper
0  2.700075  0.019878       2.661104       2.739047      1.601749      3.798402
1  2.425282  0.014258       2.397329       2.453235      1.327292      3.523273
2  3.457448  0.027891       3.402766       3.512130      2.358452      4.556444

colgpa_PICI_99:
       mean   mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  obs_ci_upper
0  2.700075  0.019878       2.648850       2.751301      1.256386      4.143765
1  2.425282  0.014258       2.388540       2.462025      0.982034      3.868530
2  3.457448  0.027891       3.385572       3.529325      2.012879      4.902018
```

6.2.2. Effect Plots for Nonlinear Specifications

In models with quadratic or other nonlinear terms, the coefficients themselves are often difficult to interpret directly. We have to do additional calculations to obtain the partial effect at different values of the regressors or derive the extreme points. In Example 6.2, we found the number of rooms implying the minimum predicted house price to be around 4.4.

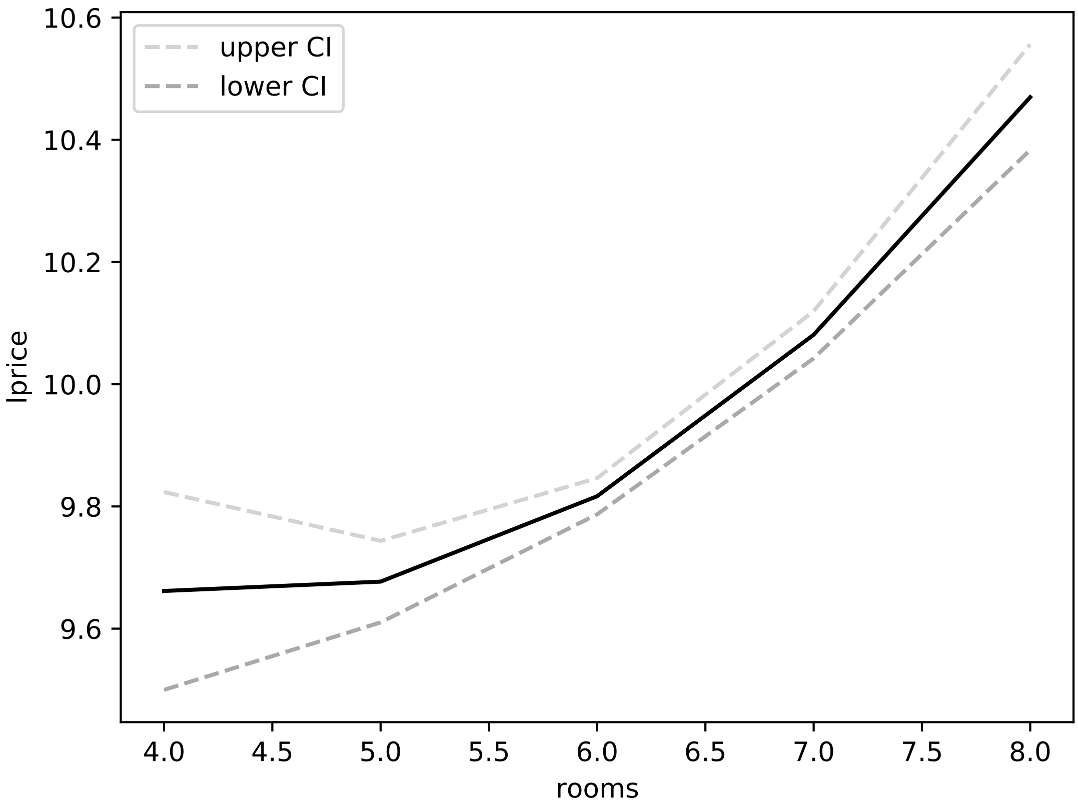
For a better visual understanding of the implications of our model, it is often useful to calculate predictions for *different values of one regressor* of interest while keeping *the other regressors fixed* at certain values like their overall sample means. By plotting the results against the regressor value, we get a very intuitive graph showing the estimated *ceteris paribus* effects of the regressor.

We already know how to calculate predictions and their confidence intervals from Section 6.2.1. Script 6.9 (`Effects-Manual.py`) repeats the regression from Example 6.2 and creates an effects plot for the number of rooms. The number of rooms is varied between 4 and 8 and the other variables are set to their respective sample means for all predictions. The regressor values and the implied predictions are shown in a table and then plotted with their confidence bands. We see the minimum at a number of rooms of around 4. The resulting graph is shown in Figure 6.1.

Script 6.9: `Effects-Manual.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

hprice2 = woo.dataWoo('hprice2')

# repeating the regression from Example 6.2:
reg = smf.ols(
    formula='np.log(price) ~ np.log(nox)+np.log(dist)+rooms+I(rooms**2)+stratio',
    data=hprice2)
results = reg.fit()

# predictions with rooms = 4-8, all others at the sample mean:
nox_mean = np.mean(hprice2['nox'])
dist_mean = np.mean(hprice2['dist'])
stratio_mean = np.mean(hprice2['stratio'])
X = pd.DataFrame({'rooms': np.linspace(4, 8, num=5),
                  'nox': nox_mean,
                  'dist': dist_mean,
                  'stratio': stratio_mean})
print(f'X: \n{X}\n')

# calculate 95% confidence interval:
lpr_PICI = results.get_prediction(X).summary_frame(alpha=0.05)
lpr_CI = lpr_PICI[['mean', 'mean_ci_lower', 'mean_ci_upper']]
print(f'lpr_CI: \n{lpr_CI}\n')

# plot:
plt.plot(X['rooms'], lpr_CI['mean'], color='black',
         linestyle='-', label='')
plt.plot(X['rooms'], lpr_CI['mean_ci_upper'], color='lightgrey',
         linestyle='--', label='upper CI')
plt.plot(X['rooms'], lpr_CI['mean_ci_lower'], color='darkgrey',
         linestyle='--', label='lower CI')
plt.ylabel('lprice')
plt.xlabel('rooms')
plt.legend()
plt.savefig('PyGraphs/Effects-Manual.pdf')
```

Figure 6.1. Nonlinear Effects in Example 6.2



Output of Script 6.9: `Effects-Manual.py`

```
X:
   rooms       nox      dist    stratio
0    4.0  5.549783  3.795751  18.459289
1    5.0  5.549783  3.795751  18.459289
2    6.0  5.549783  3.795751  18.459289
3    7.0  5.549783  3.795751  18.459289
4    8.0  5.549783  3.795751  18.459289

lpr_CI:
        mean  mean_ci_lower  mean_ci_upper
0   9.661702       9.499811       9.823593
1   9.676940       9.610215       9.743665
2   9.816700       9.787055       9.846345
3  10.080983      10.042409      10.119557
4  10.469788      10.383361      10.556215
```

7. Multiple Regression Analysis with Qualitative Regressors

Many variables of interest are qualitative rather than quantitative. Examples include gender, race, labor market status, marital status, and brand choice. In this chapter, we discuss the use of qualitative variables as regressors. Wooldridge (2019, Section 7.5) also covers linear probability models with a binary dependent variable in a linear regression. Since this does not change the implementation, we will skip this topic here and cover binary dependent variables in Chapter 17.

Qualitative information can be represented as binary or dummy variables which can only take the value zero or one. In Section 7.1, we see that dummy variables can be used as regressors just as any other variable. An even more natural way to store yes/no type of information in *Python* is to use Boolean variables which can also be directly used as regressors, see Section 7.2.

While qualitative variables with more than two outcomes can be represented by a set of dummy variables, the more natural and convenient way to do this are categorical variables as covered in Section 1.2.4. A special case in which we wish to break a numeric variable into categories is discussed in Section 7.4. Finally, Section 7.5 revisits interaction effects and shows how these can be used with categorical variables to conveniently allow and test for difference in the regression equation.

7.1. Linear Regression with Dummy Variables as Regressors

If qualitative data are stored as dummy variables (i.e. variables taking the values zero or one), these can easily be used as regressors in linear regression. If a single dummy variable is used in a model, its coefficient represents the difference in the intercept between groups, see Wooldridge (2019, Section 7.2).

A qualitative variable can also take $g > 2$ values. A variable `MobileOS` could for example take one of the $g = 4$ values “Android”, “iOS”, “Windows”, or “other”. This information can be represented by $g - 1$ dummy variables, each taking the values zero or one, where one category is left out to serve as a reference category. They take the value one if the respective operating system is used and zero otherwise. Wooldridge (2019, Section 7.3) gives more information on these variables and their interpretation.

Here, we are concerned with implementing linear regressions with dummy variables as regressors. Everything works as before once we have generated the dummy variables. In the example data sets provided with Wooldridge (2019), this has usually already been done for us, so we don’t have to learn anything new in terms of implementation. We show two examples.

Wooldridge, Example 7.1: Hourly Wage Equation

We are interested in the wage differences by gender and regress the hourly wage on a dummy variable which is equal to one for females and zero for males. We also include regressors for education, experience, and tenure. The implementation with **`statsmodels`** is standard and the dummy variable **`female`** is used just as any other regressor as shown in Script 7.1 (`Example-7-1.py`). Its estimated coefficient of -1.81 indicates that on average, a woman makes \$1.81 per hour less than a man *with the same education, experience, and tenure.*

Script 7.1: `Example-7-1.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='wage ~ female + educ + exper + tenure', data=wage1)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 7.1: `Example-7-1.py`

```
table:
                b      se        t    pval
Intercept -1.5679  0.7246  -2.1640  0.0309
female    -1.8109  0.2648  -6.8379  0.0000
educ       0.5715  0.0493  11.5836  0.0000
exper      0.0254  0.0116   2.1951  0.0286
tenure     0.1410  0.0212   6.6632  0.0000
```

Wooldridge, Example 7.6: Log Hourly Wage Equation

We used log wage as the dependent variable and distinguish gender and marital status using a qualitative variable with the four outcomes "single female", "single male", "married female", and "married male". We actually implement this regression using an interaction term between **`married`** and **`female`** in Script 7.2 (`Example-7-6.py`). *Relative to the reference group* of single males *with the same education, experience, and tenure*, married males make about 21.3% more (the coefficient of **`married`**), and single females make about 11.0% less (the coefficient of **`female`**). The coefficient of the interaction term implies that married females make around 30.1%-21.3%=8.7% less *than single females*, 30.1%+11.0%=41.1% less than married males, and 30.1%+11.0%-21.3%=19.8% less than single males. Note once again that the approximate interpretation as percent may be inaccurate, see Section 6.1.3.

Script 7.2: `Example-7-6.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ married*female + educ + exper +'
                      'I(exper**2) + tenure + I(tenure**2)', data=wage1)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 7.2: `Example-7-6.py`

```
table:
                     b      se        t    pval
Intercept       0.3214  0.1000   3.2135  0.0014
married         0.2127  0.0554   3.8419  0.0001
female         -0.1104  0.0557  -1.9797  0.0483
married:female -0.3006  0.0718  -4.1885  0.0000
educ            0.0789  0.0067  11.7873  0.0000
exper           0.0268  0.0052   5.1118  0.0000
I(exper ** 2)  -0.0005  0.0001  -4.8471  0.0000
tenure          0.0291  0.0068   4.3016  0.0000
I(tenure ** 2) -0.0005  0.0002  -2.3056  0.0215
```

7.2. Boolean Variables

A natural way for storing qualitative yes/no information in *Python* is to use Boolean variables introduced in Section 1.2.2. They can take the values **True** or **False** and can be transformed into a 0/1 dummy variable with the function **int** where **True=1** and **False=0**. 0/1-coded dummies can *vice versa* be transformed into logical variables with the function **bool**.

Instead of transforming Boolean variables into dummies, they can be directly used as regressors. The coefficient is then named **varname[T.True]** indicating that **True** was treated as **1**. Script 7.3 (Example-7-1-Boolean.py) repeats the analysis of Example 7.1 with the regressor **female** being coded as **bool** instead of a 0/1 dummy variable.[1]

Script 7.3: **Example-7-1-Boolean.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

# regression with boolean variable:
wage1['isfemale'] = (wage1['female'] == 1)
reg = smf.ols(formula='wage ~ isfemale + educ + exper + tenure', data=wage1)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 7.3: **Example-7-1-Boolean.py**

```
table:
                       b      se        t    pval
Intercept        -1.5679  0.7246  -2.1640  0.0309
isfemale[T.True] -1.8109  0.2648  -6.8379  0.0000
educ              0.5715  0.0493  11.5836  0.0000
exper             0.0254  0.0116   2.1951  0.0286
tenure            0.1410  0.0212   6.6632  0.0000
```

In real-world data sets, qualitative information is often not readily coded as logical or dummy variables, so we might want to create our own regressors. Suppose a qualitative variable saved as the **numpy** array **OS** takes one of the three string values “Android”, “iOS”, “Windows”, or “other”. We can manually define the three relevant logical variables with “Android” as the reference category with

```
iOS  = OS=='iOS'
wind = OS=='Windows'
oth  = OS=='other'
```

A more convenient and elegant way to deal with qualitative variables are categorical variables discussed in the next section.

[1] To be more precise, a **numpy** version of the type **bool** is used internally to allow for vectorized operations.

7.3. Categorical Variables

We have introduced categorical variables of type **Categorical** in Section 1.2.4. They take one of a given set of outcomes which can be labeled arbitrarily. This makes them the natural variable type to store qualitative information.

In a linear regression performed by **statsmodels** we can easily transform any variable into a categorical variable using the function **C** in the definition of the formula. The function **ols** is clever enough to implicitly add $g-1$ dummy variables if the variable has g outcomes. As a reference category, the first category is left out by default.

Script 7.4 (Regr-Categorical.py) shows how categorical variables are used. It uses the data set CPS1985.[2] This data set is similar to the one used in Examples 7.1 and 7.6 in that it contains wage and other data for 534 individuals. The frequency tables for the two variables **gender** and **occupation** are shown in the output. The variable **gender** has two categories **male** and **female**. The variable **occupation** has six categories.

In the output, the coefficients are labeled with a combination of the variable and category name. As an example, the estimated coefficient of 0.224 for **C(gender)[T.male]** in **results** implies that men make about 22.4% more than women who are the same in terms of the other regressors. Employees in technical positions earn around 1% (see coefficient of **C(occupation)[T.technical]**) less than otherwise equal management positions (who are the reference category).

We can choose different reference categories using a second argument of the **C** command, where we provide a new reference group **somegroup** with the command **Treatment('somegroup')**. In the specification **results_newref**, we choose **male** and **technical**. When we rerun the same regression command, we see the expected results: Variables like **education** and **experience** get the same coefficients. The dummy variable for females gets the negative of what the males got previously. Obviously, it is equivalent to say "female log wages are lower by 0.224" and "male log wages are higher by 0.224".

The coefficients for the occupation are now relative to **technical**. From the first regression we already knew that technical positions make 1% less than managers, so it is not surprising that in the second regression we find that managers make 1% more than technical positions. The other occupation coefficients are higher by 0.010085 implying the same relative comparisons as in the first specification.

Script 7.4: Regr-Categorical.py

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf

CPS1985 = pd.read_csv('data/CPS1985.csv')
# rename variable to make outputs more compact:
CPS1985['oc'] = CPS1985['occupation']

# table of categories and frequencies for two categorical variables:
freq_gender = pd.crosstab(CPS1985['gender'], columns='count')
print(f'freq_gender: \n{freq_gender}\n')

freq_occupation = pd.crosstab(CPS1985['oc'], columns='count')
print(f'freq_occupation: \n{freq_occupation}\n')
```

[2] The data set is included in the R package **AER**, see https://cran.r-project.org/web/packages/AER/index.html.

```
# directly using categorical variables in regression formula:
reg = smf.ols(formula='np.log(wage) ~ education +'
                      'experience + C(gender) + C(oc)', data=CPS1985)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# rerun regression with different reference category:
reg_newref = smf.ols(formula='np.log(wage) ~ education + experience + '
                             'C(gender, Treatment("male")) + '
                             'C(oc, Treatment("technical"))', data=CPS1985)
results_newref = reg_newref.fit()

# print results:
table_newref = pd.DataFrame({'b': round(results_newref.params, 4),
                             'se': round(results_newref.bse, 4),
                             't': round(results_newref.tvalues, 4),
                             'pval': round(results_newref.pvalues, 4)})
print(f'table_newref: \n{table_newref}\n')
```

Output of Script 7.4: `Regr-Categorical.py`

```
freq_gender:
col_0   count
gender
female    245
male      289

freq_occupation:
col_0       count
oc
management     55
office         97
sales          38
services       83
technical     105
worker        156

table:
                          b      se       t    pval
Intercept            0.9050  0.1717  5.2718  0.0000
C(gender)[T.male]    0.2238  0.0423  5.2979  0.0000
C(oc)[T.office]     -0.2073  0.0776 -2.6699  0.0078
C(oc)[T.sales]      -0.3601  0.0936 -3.8455  0.0001
C(oc)[T.services]   -0.3626  0.0818 -4.4305  0.0000
C(oc)[T.technical] -0.0101  0.0740 -0.1363  0.8916
C(oc)[T.worker]     -0.1525  0.0763 -1.9981  0.0462
education            0.0759  0.0101  7.5449  0.0000
experience           0.0119  0.0017  7.0895  0.0000
```

```
table_newref:
                                                 b      se       t    pval
Intercept                                   1.1187  0.1765  6.3393  0.0000
C(gender, Treatment("male"))[T.female]     -0.2238  0.0423 -5.2979  0.0000
C(oc, Treatment("technical"))[T.management]  0.0101  0.0740  0.1363  0.8916
C(oc, Treatment("technical"))[T.office]    -0.1972  0.0678 -2.9082  0.0038
C(oc, Treatment("technical"))[T.sales]     -0.3500  0.0863 -4.0541  0.0001
C(oc, Treatment("technical"))[T.services]  -0.3525  0.0750 -4.7030  0.0000
C(oc, Treatment("technical"))[T.worker]    -0.1425  0.0705 -2.0218  0.0437
education                                   0.0759  0.0101  7.5449  0.0000
experience                                  0.0119  0.0017  7.0895  0.0000
```

7.3.1. ANOVA Tables

A natural question to ask is whether a regressor has additional statistically significant explanatory power in a regression model, given all the other regressors. In simple model specifications, this question can be answered by a simple t test, so the results for all regressors are available with a quick look at the standard regression table.[3] When working with categorical variables, polynomials or other specifications, the influence of one variable is captured by several regressors. In the example of Script 7.4 (`Regr-Categorical.py`), the effect of **`occupation`** is captured by the five regressors of the respective dummy variables.

We can test its significance with an F test of the joint null hypothesis that all of these parameters are equal to zero. As an example, let's revisit the underlying model in **`reg`** from Script 7.4 (`Regr-Categorical.py`):

$$\begin{aligned}\log(\texttt{wage}) =&\beta_0 + \beta_1\texttt{education} + \beta_2\texttt{experience} + \beta_3\texttt{gender} + \beta_4\texttt{office} \\ &+ \beta_5\texttt{sales} + \beta_6\texttt{services} + \beta_7\texttt{technical} + \beta_8\texttt{worker} + u\end{aligned}$$

The significance of **`occupation`** can be assessed with an F test of $H_0 : \beta_4 = \beta_5 = \beta_6 = \beta_7 = \beta_8 = 0$. As discussed in Section 4.3, such a test can be performed with the command **`f_test`** from the module **`statsmodels`**.

A Type II ANOVA (analysis of variance) table does exactly this for each variable in the model and displays the results in a clearly arranged table. **`statsmodels`** implements this in the method **`anova_lm`**.[4] The example in Script 7.5 (`Regr-Categorical-Anova.py`) shows that all the relevant results from our previous F test can be found again in the row labelled **`occupation`**. Column **`df`** indicates that this test involves five parameters. All other variables enter the model with a single parameter. Consequently the value of their F test statistics corresponds to the respective squared t statistics in the object **`results`**.

The ANOVA table also allows to quickly compare the relevance of the regressors. The first column shows the sum of squared deviations explained by the variables after all the other regressors are controlled for. ANOVA tables of Types I and III are less often of interest. They differ in what other variables are controlled for when testing for the effect of one regressor.

Script 7.5 (`Regr-Categorical-Anova.py`) shows the ANOVA Type II table. We see that **`education`** has the highest explanatory power. Moreover, **`occupation`** has a highly significant effect on wages. The explained sum of squares (after controlling for all other regressors) is higher than that of **`gender`**. But since it is based on five parameters instead of one, the F statistic is lower.

[3] Section 4.1 discusses t tests.

[4] In **`statsmodels`**, this functionality is not located in **`statsmodels.formula.api`**, where we find formula based estimation routines. Instead it is in **`statsmodels.api`**, so we import another part of the module as the alias **`sm`**.

Script 7.5: `Regr-Categorical-Anova.py`

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

CPS1985 = pd.read_csv('data/CPS1985.csv')

# run regression:
reg = smf.ols(
    formula='np.log(wage) ~ education + experience + gender + occupation',
    data=CPS1985)
results = reg.fit()

# print regression table:
table_reg = pd.DataFrame({'b': round(results.params, 4),
                          'se': round(results.bse, 4),
                          't': round(results.tvalues, 4),
                          'pval': round(results.pvalues, 4)})
print(f'table_reg: \n{table_reg}\n')

# ANOVA table:
table_anova = sm.stats.anova_lm(results, typ=2)
print(f'table_anova: \n{table_anova}\n')
```

Output of Script 7.5: `Regr-Categorical-Anova.py`

```
table_reg:
                               b      se       t    pval
Intercept                 0.9050  0.1717  5.2718  0.0000
gender[T.male]            0.2238  0.0423  5.2979  0.0000
occupation[T.office]     -0.2073  0.0776 -2.6699  0.0078
occupation[T.sales]      -0.3601  0.0936 -3.8455  0.0001
occupation[T.services]   -0.3626  0.0818 -4.4305  0.0000
occupation[T.technical] -0.0101  0.0740 -0.1363  0.8916
occupation[T.worker]     -0.1525  0.0763 -1.9981  0.0462
education                 0.0759  0.0101  7.5449  0.0000
experience                0.0119  0.0017  7.0895  0.0000

table_anova:
                sum_sq     df          F        PR(>F)
gender        5.414018    1.0  28.067296  1.727015e-07
occupation    7.152529    5.0   7.416013  9.805485e-07
education    10.980589    1.0  56.925450  2.010374e-13
experience    9.695055    1.0  50.261001  4.365391e-12
Residual    101.269451  525.0        NaN           NaN
```

7.4. Breaking a Numeric Variable Into Categories

Sometimes, we do not use a numeric variable directly in a regression model because the implied linear relation seems implausible or inconvenient to interpret. As an alternative to working with transformations such as logs and quadratic terms, it sometimes makes sense to estimate different levels for different ranges of the variable. Wooldridge (2019, Example 7.8) gives the example of the ranking of a law school and how it relates to the starting salary of its graduates.

Given a numeric variable, we need to generate a categorical variable to represent the range into which the rank of a school falls. In *Python*, the command **cut** from **pandas** is very convenient for this. It takes a numeric variable and a list of cut points and returns a categorical variable. By default, the upper cut points are included in the corresponding range.

Wooldridge, Example 7.8: Effects of Law School Rankings on Starting Salaries

The variable **rank** of the data set LAWSCH85 is the rank of the law school as a number between 1 and 175. We would like to compare schools in the top 10, ranks 11–25, 26–40, 41–60, and 61–100 to the reference group of ranks above 100. So in Script 7.6 (Example-7-8.py), we store the cut points 0, 10, 25, 40, 60, 100, and 175 in a variable **cutpts**. In the data frame **lawsch85**, we create our new variable **rc** using the **cut** command.

To be consistent with Wooldridge (2019), we do not want the top 10 schools as a reference category but the last category. It is chosen with the second argument of the **C** command. The regression results imply that graduates from the top 10 schools collect a starting salary which is around 70% higher than those of the schools below rank 100. In fact, this approximation is inaccurate with these large numbers and the coefficient of 0.7 actually implies a difference of exp(0.7)-1=1.013 or 101.3%.

The ANOVA table at the end of the output shows that at a 5% significance level, the school rank is the only variable that has a significant explanatory power for the salary in this specification.

Script 7.6: **Example-7-8.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

lawsch85 = woo.dataWoo('lawsch85')

# define cut points for the rank:
cutpts = [0, 10, 25, 40, 60, 100, 175]

# create categorical variable containing ranges for the rank:
lawsch85['rc'] = pd.cut(lawsch85['rank'], bins=cutpts,
                        labels=['(0,10]', '(10,25]', '(25,40]',
                                '(40,60]', '(60,100]', '(100,175]'])

# display frequencies:
freq = pd.crosstab(lawsch85['rc'], columns='count')
print(f'freq: \n{freq}\n')

# run regression:
reg = smf.ols(formula='np.log(salary) ~ C(rc, Treatment("(100,175]")) +'
                      'LSAT + GPA + np.log(libvol) + np.log(cost)',
              data=lawsch85)
results = reg.fit()
```

```
# print regression table:
table_reg = pd.DataFrame({'b': round(results.params, 4),
                          'se': round(results.bse, 4),
                          't': round(results.tvalues, 4),
                          'pval': round(results.pvalues, 4)})
print(f'table_reg: \n{table_reg}\n')

# ANOVA table:
table_anova = sm.stats.anova_lm(results, typ=2)
print(f'table_anova: \n{table_anova}\n')
```

Output of Script 7.6: Example-7-8.py

```
freq:
col_0      count
rc
(0,10]        10
(10,25]       16
(25,40]       13
(40,60]       18
(60,100]      37
(100,175]     62

table_reg:
                                                b      se        t    pval
Intercept                                  9.1653  0.4114  22.2770  0.0000
C(rc, Treatment("(100,175]"))[T.(0,10]]    0.6996  0.0535  13.0780  0.0000
C(rc, Treatment("(100,175]"))[T.(10,25]]   0.5935  0.0394  15.0493  0.0000
C(rc, Treatment("(100,175]"))[T.(25,40]]   0.3751  0.0341  11.0054  0.0000
C(rc, Treatment("(100,175]"))[T.(40,60]]   0.2628  0.0280   9.3991  0.0000
C(rc, Treatment("(100,175]"))[T.(60,100]]  0.1316  0.0210   6.2540  0.0000
LSAT                                       0.0057  0.0031   1.8579  0.0655
GPA                                        0.0137  0.0742   0.1850  0.8535
np.log(libvol)                             0.0364  0.0260   1.3976  0.1647
np.log(cost)                               0.0008  0.0251   0.0335  0.9734

table_anova:
                                 sum_sq     df          F        PR(>F)
C(rc, Treatment("(100,175]"))  1.868867    5.0  50.962988  1.174406e-28
LSAT                           0.025317    1.0   3.451900  6.551320e-02
GPA                            0.000251    1.0   0.034225  8.535262e-01
np.log(libvol)                 0.014327    1.0   1.953419  1.646748e-01
np.log(cost)                   0.000008    1.0   0.001120  9.733564e-01
Residual                       0.924111  126.0        NaN           NaN
```

7.5. Interactions and Differences in Regression Functions Across Groups

Dummy and categorical variables can be interacted just like any other variable. Wooldridge (2019, Section 7.4) discusses the specification and interpretation in this setup. An important case is a model in which one or more dummy variables are interacted with all other regressors. This allows the whole regression model to differ by groups of observations identified by the dummy variable(s).

The example from Wooldridge (2019, Section 7.4-c) is replicated in Script 7.7 (`Dummy-Interact.py`). Note that the example only applies to the subset of data with **`spring==1`**. We use the **`subset`** option of **`ols`** directly to define the estimation sample. Other than that, the script does not introduce any new syntax but combines two tricks we have seen previously:

- The dummy variable **`female`** is interacted with all other regressors using the "**`*`**" formula syntax with the other variables contained in parentheses, see Section 6.1.6.
- The F test for all interaction effects is performed using the command **`f_test`**.

Script 7.7: `Dummy-Interact.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# model with full interactions with female dummy (only for spring data):
reg = smf.ols(formula='cumgpa ~ female * (sat + hsperc + tothrs)',
              data=gpa3, subset=(gpa3['spring'] == 1))
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# F-Test for H0 (the interaction coefficients of 'female' are zero):
hypotheses = ['female = 0', 'female:sat = 0',
              'female:hsperc = 0', 'female:tothrs = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Output of Script 7.7: Dummy-Interact.py

```
table:
                    b      se       t    pval
Intercept      1.4808  0.2073  7.1422  0.0000
female        -0.3535  0.4105 -0.8610  0.3898
sat            0.0011  0.0002  5.8073  0.0000
hsperc        -0.0085  0.0014 -6.1674  0.0000
tothrs         0.0023  0.0009  2.7182  0.0069
female:sat     0.0008  0.0004  1.9488  0.0521
female:hsperc -0.0005  0.0032 -0.1739  0.8621
female:tothrs -0.0001  0.0016 -0.0712  0.9433

fstat: 8.179111637047

fpval: 2.544637191818678e-06
```

We can estimate the same model parameters by running two separate regressions, one for females and one for males, see Script 7.8 (`Dummy-Interact-Sep.py`). We see that in the joint model, the parameters without interactions (**`Intercept`**, **`sat`**, **`hsperc`**, and **`tothrs`**) apply to the males and the interaction parameters reflect the *differences* to the males.

To reconstruct the parameters for females from the joint model, we need to add the two respective parameters. The intercept for females is $1.4808 - 0.3535 = 1.1273$ and the coefficient of **`sat`** for females is $0.0011 + 0.0008 \approx 0.0018$.

Script 7.8: Dummy-Interact-Sep.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# estimate model for males (& spring data):
reg_m = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs',
                data=gpa3,
                subset=(gpa3['spring'] == 1) & (gpa3['female'] == 0))
results_m = reg_m.fit()

# print regression table:
table_m = pd.DataFrame({'b': round(results_m.params, 4),
                        'se': round(results_m.bse, 4),
                        't': round(results_m.tvalues, 4),
                        'pval': round(results_m.pvalues, 4)})
print(f'table_m: \n{table_m}\n')

# estimate model for females (& spring data):
reg_f = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs',
                data=gpa3,
                subset=(gpa3['spring'] == 1) & (gpa3['female'] == 1))
results_f = reg_f.fit()

# print regression table:
table_f = pd.DataFrame({'b': round(results_f.params, 4),
                        'se': round(results_f.bse, 4),
                        't': round(results_f.tvalues, 4),
                        'pval': round(results_f.pvalues, 4)})
print(f'table_f: \n{table_f}\n')
```

Output of Script 7.8: `Dummy-Interact-Sep.py`

```
table_m:
                b      se       t    pval
Intercept  1.4808  0.2060  7.1894  0.0000
sat        0.0011  0.0002  5.8458  0.0000
hsperc    -0.0085  0.0014 -6.2082  0.0000
tothrs     0.0023  0.0009  2.7362  0.0066

table_f:
                b      se       t    pval
Intercept  1.1273  0.3616  3.1176  0.0025
sat        0.0018  0.0003  5.1950  0.0000
hsperc    -0.0090  0.0029 -3.0956  0.0027
tothrs     0.0022  0.0014  1.5817  0.1174
```

8. Heteroscedasticity

The homoscedasticity assumptions SLR.5 for the simple regression model and MLR.5 for the multiple regression model require that the variance of the error terms is unrelated to the regressors, i.e.

$$\text{Var}(u|x_1, \ldots, x_k) = \sigma^2. \tag{8.1}$$

Unbiasedness and consistency (Theorems 3.1, 5.1) do not depend on this assumption, but the sampling distribution (Theorems 3.2, 4.1, 5.2) does. If homoscedasticity is violated, the standard errors are invalid and all inferences from t, F and other tests based on them are unreliable. Also the (asymptotic) efficiency of OLS (Theorems 3.4, 5.3) depends on homoscedasticity. Generally, homoscedasticity is difficult to justify from theory. Different kinds of individuals might have different amounts of unobserved influences in ways that depend on regressors.

We cover three topics: Section 8.1 shows how the formula of the estimated variance-covariance can be adjusted so it does not require homoscedasticity. In this way, we can use OLS to get unbiased and consistent parameter estimates and draw inference from valid standard errors and tests. Section 8.2 presents tests for the existence of heteroscedasticity. Section 8.3 discusses weighted least squares (WLS) as an alternative to OLS. This estimator can be more efficient in the presence of heteroscedasticity.

8.1. Heteroscedasticity-Robust Inference

Wooldridge (2019, Section 8.2) presents formulas for heteroscedasticity-robust standard errors. In **`statsmodels`**, an easy way to do these calculations is to make use of the argument **`cov_type`** in the method **`fit`**. The argument **`cov_type`** can produce several refined versions of the White formula presented by Wooldridge (2019).

If the regression model obtained by **`ols`** is stored in the variable **`reg`**, the variance-covariance matrix can be calculated using

- **`reg.fit(cov_type='nonrobust')`** or **`reg.fit()`** for the default homoscedasticity-based standard errors.
- **`reg.fit(cov_type='HC0')`** for the classical version of White's robust variance-covariance matrix presented by Wooldridge (2019, Equation 8.4 in Section 8.2).
- **`reg.fit(cov_type='HC1')`** for a version of White's robust variance-covariance matrix corrected by degrees of freedom.
- **`reg.fit(cov_type='HC2')`** for a version with a small sample correction. This is the default behavior of Stata.
- **`reg.fit(cov_type='HC3')`** for the refined version of White's robust variance-covariance matrix.

Regression tables with coefficients, standard errors, t statistics and their p values are based on the specified method of variance-covariance estimation. To perform F tests of a joint hypothesis for an estimated model the syntax is the same as in Section 4.3.

Wooldridge, Example 8.2: Heteroscedasticity-Robust Inference

Scripts 8.1 (`Example-8-2.py`) and 8.2 (`Example-8-2-cont.py`) demonstrate these commands. **results_default** and **results_white** use the usual standard errors and the classical White standard errors respectively. This reproduces standard errors reported in Wooldridge (2019).
For the F tests shown in Script 8.2 (`Example-8-2-cont.py`), three versions are calculated and displayed. The results generally do not differ a lot between the different versions. This is an indication that heteroscedasticity might not be a big issue in this example. To be sure, we would like to have a formal test as discussed in the next section.

Script 8.1: Example-8-2.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# define regression model:
reg = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs + female + black + white',
              data=gpa3, subset=(gpa3['spring'] == 1))

# estimate default model (only for spring data):
results_default = reg.fit()

table_default = pd.DataFrame({'b': round(results_default.params, 5),
                              'se': round(results_default.bse, 5),
                              't': round(results_default.tvalues, 5),
                              'pval': round(results_default.pvalues, 5)})
print(f'table_default: \n{table_default}\n')

# estimate model with White SE (only for spring data):
results_white = reg.fit(cov_type='HC0')

table_white = pd.DataFrame({'b': round(results_white.params, 5),
                            'se': round(results_white.bse, 5),
                            't': round(results_white.tvalues, 5),
                            'pval': round(results_white.pvalues, 5)})
print(f'table_white: \n{table_white}\n')

# estimate model with refined White SE (only for spring data):
results_refined = reg.fit(cov_type='HC3')

table_refined = pd.DataFrame({'b': round(results_refined.params, 5),
                              'se': round(results_refined.bse, 5),
                              't': round(results_refined.tvalues, 5),
                              'pval': round(results_refined.pvalues, 5)})
print(f'table_refined: \n{table_refined}\n')
```

Output of Script 8.1: Example-8-2.py

```
table_default:
                b        se         t     pval
Intercept  1.47006  0.22980  6.39706  0.00000
sat        0.00114  0.00018  6.38850  0.00000
hsperc    -0.00857  0.00124 -6.90600  0.00000
tothrs     0.00250  0.00073  3.42551  0.00068
female     0.30343  0.05902  5.14117  0.00000
black     -0.12828  0.14737 -0.87049  0.38462
white     -0.05872  0.14099 -0.41650  0.67730

table_white:
                b        se         t     pval
Intercept  1.47006  0.21856  6.72615  0.00000
sat        0.00114  0.00019  6.01360  0.00000
hsperc    -0.00857  0.00140 -6.10008  0.00000
tothrs     0.00250  0.00073  3.41365  0.00064
female     0.30343  0.05857  5.18073  0.00000
black     -0.12828  0.11810 -1.08627  0.27736
white     -0.05872  0.11032 -0.53228  0.59453

table_refined:
                b        se         t     pval
Intercept  1.47006  0.22938  6.40885  0.00000
sat        0.00114  0.00020  5.84017  0.00000
hsperc    -0.00857  0.00144 -5.93407  0.00000
tothrs     0.00250  0.00075  3.34177  0.00083
female     0.30343  0.06004  5.05388  0.00000
black     -0.12828  0.12819 -1.00074  0.31695
white     -0.05872  0.12044 -0.48758  0.62585
```

Script 8.2: Example-8-2-cont.py

```python
import wooldridge as woo
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# definition of model and hypotheses:
reg = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs + female + black + white',
              data=gpa3, subset=(gpa3['spring'] == 1))
hypotheses = ['black = 0', 'white = 0']

# F-Tests using different variance-covariance formulas:
# ususal VCOV:
results_default = reg.fit()
ftest_default = results_default.f_test(hypotheses)
fstat_default = ftest_default.statistic[0][0]
fpval_default = ftest_default.pvalue
print(f'fstat_default: {fstat_default}\n')
print(f'fpval_default: {fpval_default}\n')

# refined White VCOV:
results_hc3 = reg.fit(cov_type='HC3')
ftest_hc3 = results_hc3.f_test(hypotheses)
fstat_hc3 = ftest_hc3.statistic[0][0]
fpval_hc3 = ftest_hc3.pvalue
print(f'fstat_hc3: {fstat_hc3}\n')
print(f'fpval_hc3: {fpval_hc3}\n')
```

```
# classical White VCOV:
results_hc0 = reg.fit(cov_type='HC0')
ftest_hc0 = results_hc0.f_test(hypotheses)
fstat_hc0 = ftest_hc0.statistic[0][0]
fpval_hc0 = ftest_hc0.pvalue
print(f'fstat_hc0: {fstat_hc0}\n')
print(f'fpval_hc0: {fpval_hc0}\n')
```

Output of Script 8.2: `Example-8-2-cont.py`

```
fstat_default: 0.6796041956073398

fpval_default: 0.5074683622584049

fstat_hc3: 0.6724692957656673

fpval_hc3: 0.5110883633440992

fstat_hc0: 0.7477969818036272

fpval_hc0: 0.4741442714738484
```

8.2. Heteroscedasticity Tests

The Breusch-Pagan (BP) test for heteroscedasticity is easy to implement with basic OLS routines. After a model

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k + u \tag{8.2}$$

is estimated, we obtain the residuals $\hat{u}_i$ for all observations $i = 1, \ldots, n$. We regress their squared value on all independent variables from the original equation. We can either look at the standard F test of overall significance printed for example by the **`summary`** method. Or we can use an LM test by multiplying the R^2 from the second regression with the number of observations.

In **`statsmodels`**, this is easily done. Remember that the residuals from a regression are saved as **`resid`** in the result object that is returned by **`fit`**. Their squared value can be stored in a new variable to be used as a dependent variable in the second stage.

The LM version of the BP test is even more convenient to use with the **`statsmodels`** function **`stats.diagnostic.het_breuschpagan`**. It can be used directly as demonstrated in Script 8.3 (`Example-8-4.py`) to compute the test statistic and corresponding p value.

Wooldridge, Example 8.4: Heteroscedasticity in a Housing Price Equation

Script 8.3 (`Example-8-4.py`) implements the F and LM versions of the BP test. The command **`stats.diagnostic.het_breuschpagan`** simply takes the regression residuals and the regressor matrix as an argument and delivers a test statistic of $LM = 14.09$. The corresponding p value is smaller than 0.003 so we reject homoscedasticity for all reasonable significance levels.
The output also shows the manual implementation of a second stage regression where we regress squared residuals on the independent variables. We can directly interpret the reported F statistic of 5.34 and its p value of 0.002 as the F version of the BP test. We can manually calculate the LM statistic by multiplying the reported $R^2 = 0.16$ with the number of observations $n = 88$.
We replicate the test for an alternative model with logarithms discussed by Wooldridge (2019) together with the White test in Example 8.5 and Script 8.4 (`Example-8-5.py`).

Script 8.3: Example-8-4.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy as pt

hprice1 = woo.dataWoo('hprice1')

# estimate model:
reg = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results = reg.fit()
table_results = pd.DataFrame({'b': round(results.params, 4),
                              'se': round(results.bse, 4),
                              't': round(results.tvalues, 4),
                              'pval': round(results.pvalues, 4)})
print(f'table_results: \n{table_results}\n')

# automatic BP test (LM version):
y, X = pt.dmatrices('price ~ lotsize + sqrft + bdrms',
                    data=hprice1, return_type='dataframe')
result_bp_lm = sm.stats.diagnostic.het_breuschpagan(results.resid, X)
bp_lm_statistic = result_bp_lm[0]
bp_lm_pval = result_bp_lm[1]
print(f'bp_lm_statistic: {bp_lm_statistic}\n')
print(f'bp_lm_pval: {bp_lm_pval}\n')

# manual BP test (F version):
hprice1['resid_sq'] = results.resid ** 2
reg_resid = smf.ols(formula='resid_sq ~ lotsize + sqrft + bdrms', data=hprice1)
results_resid = reg_resid.fit()
bp_F_statistic = results_resid.fvalue
bp_F_pval = results_resid.f_pvalue
print(f'bp_F_statistic: {bp_F_statistic}\n')
print(f'bp_F_pval: {bp_F_pval}\n')
```

Output of Script 8.3: Example-8-4.py

```
table_results:
                 b       se       t    pval
Intercept -21.7703  29.4750 -0.7386  0.4622
lotsize     0.0021   0.0006  3.2201  0.0018
sqrft       0.1228   0.0132  9.2751  0.0000
bdrms      13.8525   9.0101  1.5374  0.1279

bp_lm_statistic: 14.092385504350272

bp_lm_pval: 0.002782059555689044

bp_F_statistic: 5.338919363241436

bp_F_pval: 0.002047744420936033
```

The White test is a variant of the BP test where in the second stage, we do not regress the squared first-stage residuals on the original regressors only. Instead, we add interactions and polynomials of them or include the fitted values $\hat{y}$ and $\hat{y}^2$. This can easily be done in a manual second-stage regression remembering that the fitted values are stored in the regression results object as **`fittedvalues`**.

Conveniently, we can also use the **stats.diagnostic.het_breuschpagan** command to do the calculations of the *LM* version of the test including the p values automatically. All we have to do is to explain that in the second stage we want a different set of regressors.

Wooldridge, Example 8.5: BP and White test in the Log Housing Price Equation

Script 8.4 (`Example-8-5.py`) implements the BP and the White test for a model that now contains logarithms of the dependent variable and two independent variables. The LM versions of both the BP and the White test do not reject the null hypothesis at conventional significance levels with p values of 0.238 and 0.178, respectively.

Script 8.4: Example-8-5.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy as pt

hprice1 = woo.dataWoo('hprice1')

# estimate model:
reg = smf.ols(formula='np.log(price) ~ np.log(lotsize) + np.log(sqrft) + bdrms',
              data=hprice1)
results = reg.fit()

# BP test:
y, X_bp = pt.dmatrices('np.log(price) ~ np.log(lotsize) + np.log(sqrft) + bdrms',
                       data=hprice1, return_type='dataframe')
result_bp = sm.stats.diagnostic.het_breuschpagan(results.resid, X_bp)
bp_statistic = result_bp[0]
bp_pval = result_bp[1]
print(f'bp_statistic: {bp_statistic}\n')
print(f'bp_pval: {bp_pval}\n')

# White test:
X_wh = pd.DataFrame({'const': 1, 'fitted_reg': results.fittedvalues,
                     'fitted_reg_sq': results.fittedvalues ** 2})
result_white = sm.stats.diagnostic.het_breuschpagan(results.resid, X_wh)
white_statistic = result_white[0]
white_pval = result_white[1]
print(f'white_statistic: {white_statistic}\n')
print(f'white_pval: {white_pval}\n')
```

Output of Script 8.4: Example-8-5.py

```
bp_statistic: 4.223245741805286

bp_pval: 0.23834482631493

white_statistic: 3.4472865468750253

white_pval: 0.1784149479413317
```

8.3. Weighted Least Squares

Weighted Least Squares (WLS) attempts to provide a more efficient alternative to OLS. It is a special version of a feasible generalized least squares (FGLS) estimator. Instead of the sum of squared residuals, their weighted sum is minimized. If the weights are inversely proportional to the variance, the estimator is efficient. Also the usual formula for the variance-covariance matrix of the parameter estimates and standard inference tools are valid.

We can obtain WLS parameter estimates by multiplying each variable in the model with the square root of the weight as shown by Wooldridge (2019, Section 8.4). In **statsmodels**, it is more convenient to use the option **weights=...** of the command **wls**. This provides a more concise syntax and takes care of correct residuals, fitted values, predictions, and the like in terms of the original variables. In terms of methods and arguments, **wls** is very similar to the function **ols**.

Wooldridge, Example 8.6: Financial Wealth Equation

Script 8.5 (Example-8-6.py) implements both OLS and WLS estimation for a regression of financial wealth (**nettfa**) on income (**inc**), age (**age**), gender (**male**) and eligibility for a pension plan (**e401k**) using the data set 401ksubs. Following Wooldridge (2019), we assume that the variance is proportional to the income variable **inc**. Therefore, the optimal weight is $\frac{1}{\texttt{inc}}$ which is given as **wls_weight** in the **wls** call.

Script 8.5: **Example-8-6.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

k401ksubs = woo.dataWoo('401ksubs')

# subsetting data:
k401ksubs_sub = k401ksubs[k401ksubs['fsize'] == 1]

# OLS (only for singles, i.e. 'fsize'==1):
reg_ols = smf.ols(formula='nettfa ~ inc + I((age-25)**2) + male + e401k',
                  data=k401ksubs_sub)
results_ols = reg_ols.fit(cov_type='HC0')

# print regression table:
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# WLS:
wls_weight = list(1 / k401ksubs_sub['inc'])
reg_wls = smf.wls(formula='nettfa ~ inc + I((age-25)**2) + male + e401k',
                  weights=wls_weight, data=k401ksubs_sub)
results_wls = reg_wls.fit()

# print regression table:
table_wls = pd.DataFrame({'b': round(results_wls.params, 4),
                          'se': round(results_wls.bse, 4),
                          't': round(results_wls.tvalues, 4),
                          'pval': round(results_wls.pvalues, 4)})
print(f'table_wls: \n{table_wls}\n')
```

Output of Script 8.5: Example-8-6.py

```
table_ols:
                          b      se        t    pval
Intercept          -20.9850  3.4909 -6.0114  0.0000
inc                  0.7706  0.0994  7.7486  0.0000
I((age - 25) ** 2)   0.0251  0.0043  5.7912  0.0000
male                 2.4779  2.0558  1.2053  0.2281
e401k                6.8862  2.2837  3.0153  0.0026

table_wls:
                          b      se         t    pval
Intercept          -16.7025  1.9580  -8.5304  0.0000
inc                  0.7404  0.0643  11.5140  0.0000
I((age - 25) ** 2)   0.0175  0.0019   9.0796  0.0000
male                 1.8405  1.5636   1.1771  0.2393
e401k                5.1883  1.7034   3.0458  0.0024
```

We can also use heteroscedasticity-robust statistics from Section 8.1 to account for the fact that our variance function might be misspecified. Script 8.6 (WLS-Robust.py) repeats the WLS estimation of Example 8.6 but reports non-robust and robust standard errors and t statistics. It replicates Wooldridge (2019, Table 8.2) with the only difference that we use a refined version of the robust SE formula. There is nothing special about the implementation. The fact that we used weights is correctly accounted for in the following calculations.

Script 8.6: WLS-Robust.py

```python
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

k401ksubs = woo.dataWoo('401ksubs')

# subsetting data:
k401ksubs_sub = k401ksubs[k401ksubs['fsize'] == 1]

# WLS:
wls_weight = list(1 / k401ksubs_sub['inc'])
reg_wls = smf.wls(formula='nettfa ~ inc + I((age-25)**2) + male + e401k',
                  weights=wls_weight, data=k401ksubs_sub)

# non-robust (default) results:
results_wls = reg_wls.fit()
table_default = pd.DataFrame({'b': round(results_wls.params, 4),
                              'se': round(results_wls.bse, 4),
                              't': round(results_wls.tvalues, 4),
                              'pval': round(results_wls.pvalues, 4)})
print(f'table_default: \n{table_default}\n')

# robust results (Refined White SE):
results_white = reg_wls.fit(cov_type='HC3')
table_white = pd.DataFrame({'b': round(results_white.params, 4),
                            'se': round(results_white.bse, 4),
                            't': round(results_white.tvalues, 4),
                            'pval': round(results_white.pvalues, 4)})
print(f'table_white: \n{table_white}\n')
```

Output of Script 8.6: `WLS-Robust.py`

```
table_default:
                          b      se        t    pval
Intercept          -16.7025  1.9580  -8.5304  0.0000
inc                  0.7404  0.0643  11.5140  0.0000
I((age - 25) ** 2)   0.0175  0.0019   9.0796  0.0000
male                 1.8405  1.5636   1.1771  0.2393
e401k                5.1883  1.7034   3.0458  0.0024

table_white:
                          b      se       t    pval
Intercept          -16.7025  2.2482 -7.4292  0.0000
inc                  0.7404  0.0752  9.8403  0.0000
I((age - 25) ** 2)   0.0175  0.0026  6.7650  0.0000
male                 1.8405  1.3132  1.4015  0.1611
e401k                5.1883  1.5743  3.2955  0.0010
```

The assumption made in Example 8.6 that the variance is proportional to a regressor is usually hard to justify. Typically, we don't not know the variance function and have to estimate it. This feasible GLS (FGLS) estimator replaces the (allegedly) known variance function with an estimated one.

We can estimate the relation between variance and regressors using a linear regression of the log of the squared residuals from an initial OLS regression $\log(\hat{u}^2)$ as the dependent variable. Wooldridge (2019, Section 8.4) suggests two versions for the selection of regressors:

- the regressors $x_1, \ldots, x_k$ from the original model similar to the BP test
- $\hat{y}$ and $\hat{y}^2$ from the original model similar to the White test

As the estimated error variance, we can use $\exp\left(\widehat{\log(\hat{u}^2)}\right)$. Its inverse can then be used as a weight in WLS estimation.

Wooldridge, Example 8.7: Demand for Cigarettes

Script 8.7 (`Example-8-7.py`) studies the relationship between daily cigarette consumption **`cigs`**, individual characteristics, and restaurant smoking restrictions **`restaurn`**. After the initial OLS regression, a BP test is performed which clearly rejects homoscedasticity (see previous section for the BP test). After the regression of log squared residuals on the regressors, the FGLS weights are calculated and used in the WLS regression. See Wooldridge (2019) for a discussion of the results.

Script 8.7: **Example-8-7.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy as pt

smoke = woo.dataWoo('smoke')

# OLS:
reg_ols = smf.ols(formula='cigs ~ np.log(income) + np.log(cigpric) +'
                          'educ + age + I(age**2) + restaurn',
                  data=smoke)
results_ols = reg_ols.fit()
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# BP test:
y, X = pt.dmatrices('cigs ~ np.log(income) + np.log(cigpric) + educ +'
                    'age + I(age**2) + restaurn',
                    data=smoke, return_type='dataframe')
result_bp = sm.stats.diagnostic.het_breuschpagan(results_ols.resid, X)
bp_statistic = result_bp[0]
bp_pval = result_bp[1]
print(f'bp_statistic: {bp_statistic}\n')
print(f'bp_pval: {bp_pval}\n')

# FGLS (estimation of the variance function):
smoke['logu2'] = np.log(results_ols.resid ** 2)
reg_fgls = smf.ols(formula='logu2 ~ np.log(income) + np.log(cigpric) +'
                           'educ + age + I(age**2) + restaurn', data=smoke)
results_fgls = reg_fgls.fit()
table_fgls = pd.DataFrame({'b': round(results_fgls.params, 4),
                           'se': round(results_fgls.bse, 4),
                           't': round(results_fgls.tvalues, 4),
                           'pval': round(results_fgls.pvalues, 4)})
print(f'table_fgls: \n{table_fgls}\n')

# FGLS (WLS):
wls_weight = list(1 / np.exp(results_fgls.fittedvalues))
reg_wls = smf.wls(formula='cigs ~ np.log(income) + np.log(cigpric) +'
                          'educ + age + I(age**2) + restaurn',
                  weights=wls_weight, data=smoke)
results_wls = reg_wls.fit()
table_wls = pd.DataFrame({'b': round(results_wls.params, 4),
                          'se': round(results_wls.bse, 4),
                          't': round(results_wls.tvalues, 4),
                          'pval': round(results_wls.pvalues, 4)})
print(f'table_wls: \n{table_wls}\n')
```

Output of Script 8.7: `Example-8-7.py`

```
table_ols:
                        b       se        t    pval
Intercept         -3.6398  24.0787 -0.1512  0.8799
np.log(income)     0.8803   0.7278  1.2095  0.2268
np.log(cigpric)   -0.7509   5.7733 -0.1301  0.8966
educ              -0.5015   0.1671 -3.0016  0.0028
age                0.7707   0.1601  4.8132  0.0000
I(age ** 2)       -0.0090   0.0017 -5.1765  0.0000
restaurn          -2.8251   1.1118 -2.5410  0.0112

bp_statistic: 32.25841908120121

bp_pval: 1.4557794830278942e-05

table_fgls:
                        b      se         t    pval
Intercept         -1.9207  2.5630   -0.7494  0.4538
np.log(income)     0.2915  0.0775    3.7634  0.0002
np.log(cigpric)    0.1954  0.6145    0.3180  0.7506
educ              -0.0797  0.0178   -4.4817  0.0000
age                0.2040  0.0170   11.9693  0.0000
I(age ** 2)       -0.0024  0.0002  -12.8931  0.0000
restaurn          -0.6270  0.1183   -5.2982  0.0000

table_wls:
                        b       se        t    pval
Intercept          5.6355  17.8031  0.3165  0.7517
np.log(income)     1.2952   0.4370  2.9639  0.0031
np.log(cigpric)   -2.9403   4.4601 -0.6592  0.5099
educ              -0.4634   0.1202 -3.8570  0.0001
age                0.4819   0.0968  4.9784  0.0000
I(age ** 2)       -0.0056   0.0009 -5.9897  0.0000
restaurn          -3.4611   0.7955 -4.3508  0.0000
```

9. More on Specification and Data Issues

This chapter covers different topics of model specification and data problems. Section 9.1 asks how statistical tests can help us specify the "correct" functional form given the numerous options we have seen in Chapters 6 and 7. Section 9.2 shows some simulation results regarding the effects of measurement errors in dependent and independent variables. Sections 9.3 covers missing values and how *Python* can deal with them. In Section 9.4, we briefly discuss outliers and Section 9.5, the LAD estimator is presented.

9.1. Functional Form Misspecification

We have seen many ways to flexibly specify the relation between the dependent variable and the regressors. An obvious question to ask is whether or not a given specification is the "correct" one. The Regression Equation Specification Error Test (RESET) is a convenient tool to test the null hypothesis that the functional form is adequate.

Wooldridge (2019, Section 9.1) shows how to implement it using a standard F test in a second regression that contains polynomials of fitted values from the original regression. We already know how to obtain fitted values and run an F test, so the implementation is straightforward. Even more convenient is the boxed routine **`reset_ramsey`** from the module **`statsmodels`**. We just have to supply the regression we want to test (argument **`res`**) and the order of included polynomials (argument **`degree`**) and the rest is done automatically.

Wooldridge, Example 9.2: Housing Price Equation

Script 9.1 (`Example-9-2-manual.py`) implements the RESET test using the procedure described by Wooldridge (2019) for the housing price model. As previously, we get the fitted values from the original regression using **`fittedvalues`**. Their polynomials are entered into the formula of the second regression. The F test is easily done using **`f_test`** as described in Section 4.3.

The same results are obtained more conveniently using the command **`reset_ramsey`** in Script 9.2 (`Example-9-2-automatic.py`). Both implementations deliver the same results: The test statistic is $F = 4.67$ with a p value of $p = 0.012$, so we reject the null hypothesis that this equation is correctly specified at a significance level of $\alpha = 5\%$.

Script 9.1: **Example-9-2-manual.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

hprice1 = woo.dataWoo('hprice1')

# original OLS:
reg = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results = reg.fit()

# regression for RESET test:
hprice1['fitted_sq'] = results.fittedvalues ** 2
hprice1['fitted_cub'] = results.fittedvalues ** 3
reg_reset = smf.ols(formula='price ~ lotsize + sqrft + bdrms +'
                            ' fitted_sq + fitted_cub', data=hprice1)
results_reset = reg_reset.fit()

# print regression table:
table = pd.DataFrame({'b': round(results_reset.params, 4),
                      'se': round(results_reset.bse, 4),
                      't': round(results_reset.tvalues, 4),
                      'pval': round(results_reset.pvalues, 4)})
print(f'table: \n{table}\n')

# RESET test (H0: all coeffs including "fitted" are=0):
hypotheses = ['fitted_sq = 0', 'fitted_cub = 0']
ftest_man = results_reset.f_test(hypotheses)
fstat_man = ftest_man.statistic[0][0]
fpval_man = ftest_man.pvalue

print(f'fstat_man: {fstat_man}\n')
print(f'fpval_man: {fpval_man}\n')
```

Output of Script 9.1: **Example-9-2-manual.py**

```
table:
                   b        se       t    pval
Intercept   166.0973  317.4325  0.5233  0.6022
lotsize       0.0002    0.0052  0.0295  0.9765
sqrft         0.0176    0.2993  0.0588  0.9532
bdrms         2.1749   33.8881  0.0642  0.9490
fitted_sq     0.0004    0.0071  0.0498  0.9604
fitted_cub    0.0000    0.0000  0.2358  0.8142

fstat_man: 4.668205534950367

fpval_man: 0.012021711442865948
```

Script 9.2: `Example-9-2-automatic.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import statsmodels.stats.outliers_influence as smo

hprice1 = woo.dataWoo('hprice1')

# original linear regression:
reg = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results = reg.fit()

# automated RESET test:
reset_output = smo.reset_ramsey(res=results, degree=3)
fstat_auto = reset_output.statistic[0][0]
fpval_auto = reset_output.pvalue

print(f'fstat_auto: {fstat_auto}\n')
print(f'fpval_auto: {fpval_auto}\n')
```

Output of Script 9.2: `Example-9-2-automatic.py`

```
fstat_auto: 4.668205534948779

fpval_auto: 0.012021711442883014
```

Wooldridge (2019, Section 9.1-b) also discusses tests of non-nested models. As an example, a test of both models against a comprehensive model containing all regressors is mentioned. Such a test can be implemented in **`statsmodels`** by the command **`anova_lm`** that we already discussed. Script 9.3 (`Nonnested-Test.py`) shows this test in action for a modified version of Example 9.2.

The two alternative models for the housing price are

$$\texttt{price} = \beta_0 + \beta_1 \texttt{lotsize} + \beta_2 \texttt{sqrft} + \beta_3 \texttt{bdrms} + u, \tag{9.1}$$

$$\texttt{price} = \beta_0 + \beta_1 \log(\texttt{lotsize}) + \beta_2 \log(\texttt{sqrft}) + \beta_3 \texttt{bdrms} + u. \tag{9.2}$$

The output shows the test results of testing both models against the encompassing model with all variables. Both models are rejected against this comprehensive model.

Script 9.3: `Nonnested-Test.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

hprice1 = woo.dataWoo('hprice1')

# two alternative models:
reg1 = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results1 = reg1.fit()

reg2 = smf.ols(formula='price ~ np.log(lotsize) +'
                       'np.log(sqrft) + bdrms', data=hprice1)
results2 = reg2.fit()
```

```
# encompassing test of Davidson & MacKinnon:
# comprehensive model:
reg3 = smf.ols(formula='price ~ lotsize + sqrft + bdrms + '
                       'np.log(lotsize) + np.log(sqrft)', data=hprice1)
results3 = reg3.fit()

# model 1 vs. comprehensive model:
anovaResults1 = sm.stats.anova_lm(results1, results3)
print(f'anovaResults1: \n{anovaResults1}\n')

# model 2 vs. comprehensive model:
anovaResults2 = sm.stats.anova_lm(results2, results3)
print(f'anovaResults2: \n{anovaResults2}\n')
```

Output of Script 9.3: Nonnested-Test.py

```
anovaResults1:
   df_resid            ssr  df_diff       ss_diff         F    Pr(>F)
0      84.0  300723.805123      0.0           NaN       NaN       NaN
1      82.0  252340.364481      2.0  48383.440642  7.861291  0.000753

anovaResults2:
   df_resid            ssr  df_diff       ss_diff        F    Pr(>F)
0      84.0  295735.273607      0.0           NaN      NaN       NaN
1      82.0  252340.364481      2.0  43394.909126  7.05076  0.001494
```

9.2. Measurement Error

If a variable is not measured accurately, the consequences depend on whether the measurement error affects the dependent or an explanatory variable. If the **dependent variable** is mismeasured, the consequences can be mild. If the measurement error is unrelated to the regressors, the parameter estimates get less precise, but they are still consistent and the usual inferences from the results are valid.

The simulation exercise in Script 9.4 (Sim-ME-Dep.py) draws 10 000 samples of size $n = 1\,000$ according to the model with measurement error in the dependent variable

$$y^* = \beta_0 + \beta_1 x + u, \qquad y = y^* + e_0. \tag{9.3}$$

The assumption is that we do not observe the true values of the dependent variable y^* but our measure y is contaminated with a measurement error e_0.

Script 9.4: Sim-ME-Dep.py

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import statsmodels.formula.api as smf

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize arrays to store results later (b1 without ME, b1_me with ME):
b1 = np.empty(r)
b1_me = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u:
    u = stats.norm.rvs(0, 1, size=n)

    # draw a sample of ystar:
    ystar = beta0 + beta1 * x + u

    # measurement error and mismeasured y:
    e0 = stats.norm.rvs(0, 1, size=n)
    y = ystar + e0
    df = pd.DataFrame({'ystar': ystar, 'y': y, 'x': x})

    # regress ystar on x and store slope estimate at position i:
    reg_star = smf.ols(formula='ystar ~ x', data=df)
    results_star = reg_star.fit()
    b1[i] = results_star.params['x']

    # regress y on x and store slope estimate at position i:
    reg_me = smf.ols(formula='y ~ x', data=df)
    results_me = reg_me.fit()
    b1_me[i] = results_me.params['x']

# mean with and without ME:
b1_mean = np.mean(b1)
b1_me_mean = np.mean(b1_me)
print(f'b1_mean: {b1_mean}\n')
print(f'b1_me_mean: {b1_me_mean}\n')

# variance with and without ME:
b1_var = np.var(b1, ddof=1)
b1_me_var = np.var(b1_me, ddof=1)
print(f'b1_var: {b1_var}\n')
print(f'b1_me_var: {b1_me_var}\n')
```

Output of Script 9.4: `Sim-ME-Dep.py`

```
b1_mean: 0.5002159846382418

b1_me_mean: 0.4999676458235338

b1_var: 0.0010335543409510668

b1_me_var: 0.0020439380493408005
```

In the simulation, the parameter estimates using both the correct y^* and the mismeasured y are stored as the variables **`b1`** and **`b1_me`**, respectively. As expected, the simulated mean of both variables is close to the expected value of $\beta_1 = 0.5$. Output 9.4 (`Sim-ME-Dep.py`) shows that the variance of **`b1_me`** is around 0.002 which is twice as high as the variance of **`b1`**. This was expected since in our simulation, u and e_0 are both independent standard normal variables, so $\text{Var}(u) = 1$ and $\text{Var}(u + e_0) = 2$.

If an **explanatory variable** is mismeasured, the consequences are usually more dramatic. Even in the classical errors-in-variables case where the measurement error is unrelated to the regressors, the parameter estimates are biased and inconsistent. This model is

$$y = \beta_0 + \beta_1 x^* + u, \qquad x = x^* + e_1 \tag{9.4}$$

where the measurement error e_1 is independent of both x^* and u. Wooldridge (2019, Section 9.4) shows that if we regress y on x instead of x^*,

$$\text{plim}\hat{\beta}_1 = \beta_1 \cdot \frac{\text{Var}(x^*)}{\text{Var}(x^*) + \text{Var}(e_1)}. \tag{9.5}$$

The simulation in Script 9.5 (`Sim-ME-Explan.py`) draws 10 000 samples of size $n = 1\,000$ from this model.

Script 9.5: `Sim-ME-Explan.py`

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import statsmodels.formula.api as smf

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize b1 arrays to store results later:
b1 = np.empty(r)
b1_me = np.empty(r)

# draw a sample of x, fixed over replications:
xstar = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u:
    u = stats.norm.rvs(0, 1, size=n)

    # draw a sample of y:
    y = beta0 + beta1 * xstar + u

    # measurement error and mismeasured x:
    e1 = stats.norm.rvs(0, 1, size=n)
    x = xstar + e1
    df = pd.DataFrame({'y': y, 'xstar': xstar, 'x': x})

    # regress y on xstar and store slope estimate at position i:
    reg_star = smf.ols(formula='y ~ xstar', data=df)
    results_star = reg_star.fit()
    b1[i] = results_star.params['xstar']

    # regress y on x and store slope estimate at position i:
    reg_me = smf.ols(formula='y ~ x', data=df)
    results_me = reg_me.fit()
    b1_me[i] = results_me.params['x']

# mean with and without ME:
b1_mean = np.mean(b1)
b1_me_mean = np.mean(b1_me)
print(f'b1_mean: {b1_mean}\n')
print(f'b1_me_mean: {b1_me_mean}\n')

# variance with and without ME:
b1_var = np.var(b1, ddof=1)
b1_me_var = np.var(b1_me, ddof=1)
print(f'b1_var: {b1_var}\n')
print(f'b1_me_var: {b1_me_var}\n')
```

Output of Script 9.5: Sim-ME-Explan.py
```
b1_mean: 0.5002159846382418

b1_me_mean: 0.2445467197788616

b1_var: 0.0010335543409510668

b1_me_var: 0.0005435611029837354
```

Since in this simulation, $\text{Var}(x^*) = \text{Var}(e_1) = 1$, Equation 9.5 implies that $\text{plim}\hat{\beta}_1 = \frac{1}{2}\beta_1 = 0.25$. This is confirmed by the simulation results in Output 9.5 (`Sim-ME-Explan.py`). While the mean of the estimates in **b1** using the correct regressor again is around 0.5, the mean parameter estimate using the mismeasured regressor is about 0.25.

9.3. Missing Data and Nonrandom Samples

In many data sets, we fail to observe all variables for each observational unit. An important case is survey data where the respondents refuse or fail to answer some questions. We use **numpy** to account for missing data by using its special value **nan** (not a number). It indicates that we do not have the information or the value is not defined. The latter is usually the result of operations like $\frac{0}{0}$ or the logarithm of a negative number.

The function **isnan(value)** returns **True** if **value** is **nan** and **False** otherwise. Note that operations resulting in $\pm\infty$ like $\log(0)$ or $\frac{1}{0}$ are not coded as **nan** but as **inf** or **-inf**. Script 9.6 (`NA-NaN-Inf.py`) gives some examples.

Script 9.6: NA-NaN-Inf.py
```
import numpy as np
import pandas as pd
import scipy.stats as stats

# nan and inf handling in numpy:
x = np.array([-1, 0, 1, np.nan, np.inf, -np.inf])
logx = np.log(x)
invx = np.array(1 / x)
ncdf = np.array(stats.norm.cdf(x))
isnanx = np.isnan(x)

results = pd.DataFrame({'x': x, 'logx': logx, 'invx': invx,
                        'logx': logx, 'ncdf': ncdf, 'isnanx': isnanx})
print(f'results: \n{results}\n')
```

Output of Script 9.6: NA-NaN-Inf.py
```
results:
      x  logx  invx      ncdf  isnanx
0 -1.0   NaN  -1.0  0.158655   False
1  0.0  -inf   inf  0.500000   False
2  1.0   0.0   1.0  0.841345   False
3  NaN   NaN   NaN       NaN    True
4  inf   inf   0.0  1.000000   False
5 -inf   NaN  -0.0  0.000000   False
```

Depending on the data source, real-world data sets can have different rules for indicating missing information. Sometimes, impossible numeric values are used. For example, a survey including the number of years of education as a variable **educ** might have a value like "9999" to indicate missing information. For any software package, it is highly recommended to change these to proper missing-value codes early in the data-handling process. Otherwise, we take the risk that some statistical method interprets those values as "this person went to school for 9999 years" producing highly nonsensical results. For the education example, if the variable **educ** is in the data frame **mydata** this can be done with

```
mydata.loc[mydata['educ'] == 9999, 'educ'] = np.nan
```

We can also create Boolean variables indicating missing values using the **pandas** method **isna**. For example **mydata['educ'].isna()** will generate a Boolean variable of the same length which is **True** whenever **mydata['educ']** is **np.nan**. It can also be used on data frames. The command **mydata.isna()** will return another data frame with the same dimensions and variable names but full of Boolean variables for missing observations. It is useful to count the missings for each variable in a data frame with

```
missings = mydata.isna()
missings.sum(axis=0)
```

The argument **axis=0** makes sure that summing over observations is done for each variable, and since an observation in this case is **True** (treated as **1** by **sum**) or **False** (treated as **0** by **sum**) this gives the total amount of missing values per variable. Following the same idea, **axis=1** can be used to identify observations with no missing variables. Script 9.7 (`Missings.py`) demonstrates these commands for the data set `LAWSCH85` which contains data on law schools. Of the 156 schools, 6 do not report median LSAT scores. Looking at all variables, the most missings are found for the **age** of the school – we don't know it for 45 schools. For only 90 of the 156 schools, we have the full set of variables, for the other 66, one or more variable is missing.

Script 9.7: Missings.py

```
import wooldridge as woo
import pandas as pd

lawsch85 = woo.dataWoo('lawsch85')
lsat_pd = lawsch85['LSAT']

# create boolean indicator for missings:
missLSAT = lsat_pd.isna()

# LSAT and indicator for Schools No. 120-129:
preview = pd.DataFrame({'lsat_pd': lsat_pd[119:129],
                        'missLSAT': missLSAT[119:129]})
print(f'preview: \n{preview}\n')

# frequencies of indicator:
freq_missLSAT = pd.crosstab(missLSAT, columns='count')
print(f'freq_missLSAT: \n{freq_missLSAT}\n')

# missings for all variables in data frame (counts):
miss_all = lawsch85.isna()
colsums = miss_all.sum(axis=0)
print(f'colsums: \n{colsums}\n')

# computing amount of complete cases:
complete_cases = (miss_all.sum(axis=1) == 0)
freq_complete_cases = pd.crosstab(complete_cases, columns='count')
print(f'freq_complete_cases: \n{freq_complete_cases}\n')
```

Output of Script 9.7: `Missings.py`

```
preview:
     lsat_pd  missLSAT
119    156.0     False
120    159.0     False
121    157.0     False
122    167.0     False
123      NaN      True
124    158.0     False
125    155.0     False
126    157.0     False
127      NaN      True
128    163.0     False

freq_missLSAT:
col_0  count
LSAT
False    150
True       6

colsums:
rank        0
salary      8
cost        6
LSAT        6
GPA         7
libvol      1
faculty     4
age        45
clsize      3
north       0
south       0
east        0
west        0
lsalary     8
studfac     6
top10       0
r11_25      0
r26_40      0
r41_60      0
llibvol     1
lcost       6
dtype: int64

freq_complete_cases:
col_0  count
row_0
False     66
True      90
```

The question how to deal with missing values is not trivial and depends on many things. Modules in *Python* offer different strategies. A very strict approach is used for **numpy** data types. For basic functions such as **numpy**'s **mean** function, we cannot calculate the average, if at least one value of a provided **numpy** array is missing. Instead we have to use the function **nanmean**.

However, using the same **mean** function on **pandas** data types removes the observations with missing values and does the calculations for the remaining ones.[1] This shows that you have to check the behavior of each module in the presence of missing data to avoid errors.

The regression command **ols** removes missings by default and informs you just about the total number of complete observations used in the regression (also available in the output of **summary**). Script 9.8 (`Missings-Analyses.py`) gives examples of these features.

Script 9.8: Missings-Analyses.py

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

lawsch85 = woo.dataWoo('lawsch85')

# missings in numpy:
x_np = np.array(lawsch85['LSAT'])
x_np_bar1 = np.mean(x_np)
x_np_bar2 = np.nanmean(x_np)
print(f'x_np_bar1: {x_np_bar1}\n')
print(f'x_np_bar2: {x_np_bar2}\n')

# missings in pandas:
x_pd = lawsch85['LSAT']
x_pd_bar1 = np.mean(x_pd)
x_pd_bar2 = np.nanmean(x_pd)
print(f'x_pd_bar1: {x_pd_bar1}\n')
print(f'x_pd_bar2: {x_pd_bar2}\n')

# observations and variables:
print(f'lawsch85.shape: {lawsch85.shape}\n')

# regression (missings are taken care of by default):
reg = smf.ols(formula='np.log(salary) ~ LSAT + cost + age', data=lawsch85)
results = reg.fit()
print(f'results.nobs: {results.nobs}\n')
```

Output of Script 9.8: Missings-Analyses.py

```
x_np_bar1: nan

x_np_bar2: 158.29333333333332

x_pd_bar1: 158.29333333333332

x_pd_bar2: 158.29333333333332

lawsch85.shape: (156, 21)

results.nobs: 95.0
```

[1] This is also true for the **mean** method in **pandas**.

9.4. Outlying Observations

Wooldridge (2019, Section 9.5) offers a very useful discussion of outlying observations. One of the important messages from the discussion is that dealing with outliers is a tricky business. The module **statsmodels** offers a method **get_influence()** to automatically calculate all studentized residuals discussed there. These residuals become available under the attribute **resid_studentized_external** in the resulting object. For the R&D example from Wooldridge (2019), Script 9.9 (`Outliers.py`) calculates them and reports the highest and the lowest number. It also generates the histogram with overlayed density plot in Figure 9.1. Especially the highest value of 4.55 appears to be an extremely outlying value.

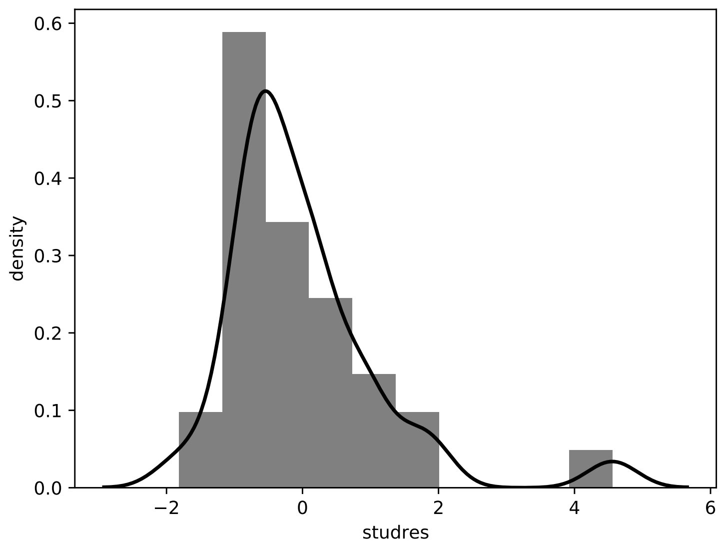
Script 9.9: Outliers.py

```
import wooldridge as woo
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

rdchem = woo.dataWoo('rdchem')

# OLS regression:
reg = smf.ols(formula='rdintens ~ sales + profmarg', data=rdchem)
results = reg.fit()

# studentized residuals for all observations:
studres = results.get_influence().resid_studentized_external

# display extreme values:
studres_max = np.max(studres)
studres_min = np.min(studres)
print(f'studres_max: {studres_max}\n')
print(f'studres_min: {studres_min}\n')

# histogram (and overlayed density plot):
kde = sm.nonparametric.KDEUnivariate(studres)
kde.fit()

plt.hist(studres, color='grey', density=True)
plt.plot(kde.support, kde.density, color='black', linewidth=2)
plt.ylabel('density')
plt.xlabel('studres')
plt.savefig('PyGraphs/Outliers.pdf')
```

Output of Script 9.9: Outliers.py

```
studres_max: 4.555033421514247

studres_min: -1.8180393952811718
```

Figure 9.1. Outliers: Distribution of Studentized Residuals



9.5. Least Absolute Deviations (LAD) Estimation

As an alternative to OLS, the least absolute deviations (LAD) estimator is less sensitive to outliers. Instead of minimizing the sum of *squared* residuals, it minimizes the sum of the *absolute values* of the residuals.

Wooldridge (2019, Section 9.6) explains that the LAD estimator attempts to estimate the parameters of the conditional median $\text{Med}(y|x_1,\ldots,x_k)$ instead of the conditional mean $\text{E}(y|x_1,\ldots,x_k)$. This makes LAD a special case of quantile regression which studies general quantiles of which the median (=0.5 quantile) is just a special case. In **statsmodels**, general quantile regression (and LAD as the special case) can easily be implemented with the command **quantreg**. It works very similar to **ols** for OLS estimation.

Script 9.10 (`LAD.py`) demonstrates its application using the example from Wooldridge (2019, Example 9.8) and Script 9.9. Note that LAD inferences are only valid asymptotically, so the results in this example with $n = 32$ should be taken with a grain of salt.

Script 9.10: LAD.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

rdchem = woo.dataWoo('rdchem')

# OLS regression:
reg_ols = smf.ols(formula='rdintens ~ I(sales/1000) + profmarg', data=rdchem)
results_ols = reg_ols.fit()

table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# LAD regression:
reg_lad = smf.quantreg(formula='rdintens ~ I(sales/1000) + profmarg', data=rdchem)
results_lad = reg_lad.fit(q=.5)

table_lad = pd.DataFrame({'b': round(results_lad.params, 4),
                          'se': round(results_lad.bse, 4),
                          't': round(results_lad.tvalues, 4),
                          'pval': round(results_lad.pvalues, 4)})
print(f'table_lad: \n{table_lad}\n')
```

Output of Script 9.10: LAD.py

```
table_ols:
                        b      se       t    pval
Intercept          2.6253  0.5855  4.4835  0.0001
I(sales / 1000)    0.0534  0.0441  1.2111  0.2356
profmarg           0.0446  0.0462  0.9661  0.3420

table_lad:
                        b      se       t    pval
Intercept          1.6231  0.7012  2.3148  0.0279
I(sales / 1000)    0.0186  0.0528  0.3529  0.7267
profmarg           0.1179  0.0553  2.1320  0.0416
```

Part II.

Regression Analysis with Time Series Data

10. Basic Regression Analysis with Time Series Data

Time series differ from cross-sectional data in that each observation (i.e. row in a data frame) corresponds to one point or period in time. Section 10.1 introduces the most basic static time series models. In Section 10.2, we look into more technical details how to deal with time series data in *Python*. Other aspects of time series models such as dynamics, trends, and seasonal effects are treated in Section 10.3.

10.1. Static Time Series Models

Static time series regression models describe the contemporaneous relation between the dependent variable y and the regressors $z_1, \ldots, z_k$. For each observation $t = 1, \ldots, n$, a static equation has the form

$$y_t = \beta_0 + \beta_1 z_{1t} + \cdots + \beta_k z_{kt} + u_t. \tag{10.1}$$

For the estimation of these models, the fact that we have time series does not make any practical difference. We can still use **`ols`** from **`statsmodels`** to estimate the parameters and the other tools for statistical inference. We only have to be aware that the assumptions needed for unbiased estimation and valid inference differ somewhat. Important differences to cross-sectional data are that we have to assume *strict* exogeneity (Assumption TS.3) for unbiasedness and no serial correlation (Assumption TS.5) for the usual variance-covariance formula to be valid, see Wooldridge (2019, Section 10.3).

Wooldridge, Example 10.2: Effects of Inflation and Deficits on Interest Rates

The data set `INTDEF` contains yearly information on interest rates and related time series between 1948 and 2003. Script 10.1 (`Example-10-2.py`) estimates a static model explaining the interest rate **`i3`** with the inflation rate **`inf`** and the federal budget deficit **`def`**. There is nothing different in the implementation than for cross-sectional data. Both regressors are found to have a statistically significant relation to the interest rate.
The example also demonstrates a practical problem: the variable names **`inf`** and **`def`** correspond to *Python* keywords that have a predefined meaning and syntax. Because we are interested in the variable and not in keywords, we have to use the **`Q`** function within the formula.

Script 10.1: `Example-10-2.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

intdef = woo.dataWoo('intdef')

# linear regression of static model (Q function avoids conflicts with keywords):
reg = smf.ols(formula='i3 ~ Q("inf") + Q("def")', data=intdef)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 10.1: `Example-10-2.py`

```
table:
                b      se       t    pval
Intercept  1.7333  0.4320  4.0125  0.0002
Q("inf")   0.6059  0.0821  7.3765  0.0000
Q("def")   0.5131  0.1184  4.3338  0.0001
```

10.2. Time Series Data Types in *Python*

For calculations specific to times series such as lags, trends, and seasonal effects, we will have to explicitly define the structure of our data. We will use **pandas** variable types specific to time series data. The most important distinction is whether or not the data are equispaced. The observations of **equispaced** time series are collected at regular points in time. Typical examples are monthly, quarterly, or yearly data.

Observations of **irregular** time series have varying distances. An important example are daily financial data which are unavailable on weekends and bank holidays. Another example is financial tick data which contain a record each time a trade is completed which obviously does not happen at regular points in time. Although we will mostly work with equispaced data, we will briefly introduce these types in Section 10.2.2.

10.2.1. Equispaced Time Series in *Python*

A convenient way to deal with equispaced time series in **pandas** is to store them as a data frame (i.e. the type **DataFrame**). To capture the time dimension, you assign an appropriate index. With equispaced time series this is especially convenient in **pandas** with the function **date_range**. It has the four important arguments **start**, **end**, **periods** and **freq** that describe the time structure of the data:

- **start** / **end**: Left/ right bound of first/ last observation is accepted in different formats. All examples create the same starting/ ending bound:
  - **start='1978-02'**
  - **start='1978-02-01'**
  - **start='02/01/1978'**

  - **start='2/1/1978'**
- **periods**: Number of equispaced points in time you need to generate.
- **freq**: Number of observations per time unit. Examples:
  - **freq='Y'**: Yearly data (at the end of a year)
  - **freq='QS'**: Quarterly data (at the beginning of a quarter)
  - **freq='M'**: Monthly data (at the end of a month)

Because the data are equispaced, you have to specify three arguments and the remaining one is implied. Obviously, this procedure only works, if two consecutive rows represent two consecutive points in time in an ascending order.

As an example, consider the data set named `BARIUM`. It contains monthly data on imports of barium chloride from China between February 1978 and December 1988. Wooldridge (2019, Example 10.5) explains the data and background. Script 10.2 (`Example-Barium.py`) demonstrates the use of **date_range** and how Figure 10.1 was generated. The time axis is automatically formatted appropriately.

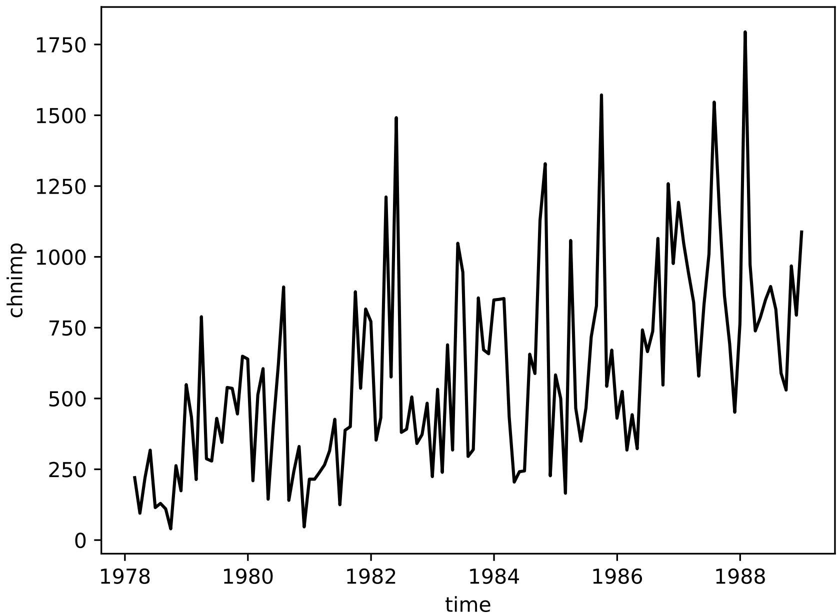
Script 10.2: Example-Barium.py

```
import wooldridge as woo
import pandas as pd
import matplotlib.pyplot as plt

barium = woo.dataWoo('barium')
T = len(barium)

# monthly time series starting Feb. 1978:
barium.index = pd.date_range(start='1978-02', periods=T, freq='M')
print(f'barium["chnimp"].head(): \n{barium["chnimp"].head()}\n')

# plot chnimp (default: index on the x-axis):
plt.plot('chnimp', data=barium, color='black', linestyle='-')
plt.ylabel('chnimp')
plt.xlabel('time')
plt.savefig('PyGraphs/Example-Barium.pdf')
```

Output of Script 10.2: Example-Barium.py

```
barium["chnimp"].head():
1978-02-28    220.462006
1978-03-31     94.797997
1978-04-30    219.357498
1978-05-31    317.421509
1978-06-30    114.639000
Freq: M, Name: chnimp, dtype: float64
```

Figure 10.1. Time Series Plot: Imports of Barium Chloride from China



10.2.2. Irregular Time Series in *Python*

For the remainder of this book, we will work with equispaced time series. But since irregular time series are important for example in finance, we will briefly introduce them here. The only thing changing is that you cannot use **date_range** to generate time stamps. Instead, these are provided in your data and you can assign them to the index of your **pandas** data frame.

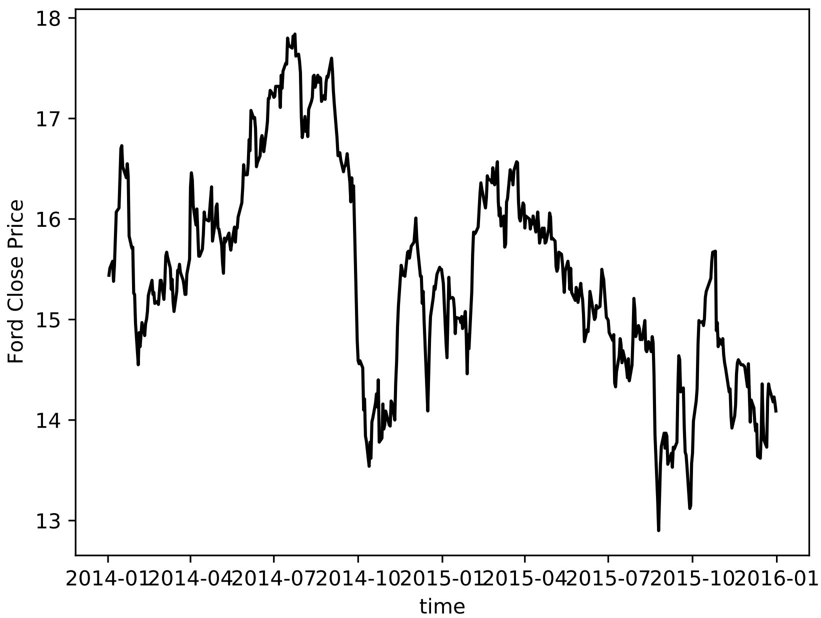
Daily financial data sets are important examples of irregular time series. Because of weekends and bank holidays, these data are not equispaced and each data point contains a time stamp - usually the date. To demonstrate this, we will briefly look at the module **pandas_datareader** introduced in Section 1.3.3. It can automatically download financial data from Yahoo Finance and other sources. In order to do so, we must know the ticker symbol of the stock or whatever we are interested in. It can be looked up at https://finance.yahoo.com/lookup.

For example, the symbol for the Dow Jones Industrial Average is ^DJI, Apple stocks have the symbol AAPL and the Ford Motor Company is simply abbreviated as F. Script 10.3 (Example-StockData.py) demonstrates the import and the format of the imported data. They include information on opening, closing, high, and low prices as well as the trading volume and the adjusted (for events like stock splits and dividend payments) closing prices. We also print the first and last 5 rows of data, and plot the adjusted closing prices over time.

Script 10.3: **Example-StockData.py**

```
import pandas_datareader as pdr
import matplotlib.pyplot as plt

# download data for 'F' (= Ford Motor Company) and define start and end:
tickers = ['F']
start_date = '2014-01-01'
end_date = '2015-12-31'

# use pandas_datareader for the import:
F_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# look at imported data:
print(f'F_data.head(): \n{F_data.head()}\n')
print(f'F_data.tail(): \n{F_data.tail()}\n')

# time series plot of adjusted closing prices:
plt.plot('Close', data=F_data, color='black', linestyle='-')
plt.ylabel('Ford Close Price')
plt.xlabel('time')
plt.savefig('PyGraphs/Example-StockData.pdf')
```

Figure 10.2. Time Series Plot: Stock Prices of Ford Motor Company



Output of Script 10.3: `Example-StockData.py`

```
F_data.head():
Attributes  Adj Close  Close   High    Low   Open      Volume
Symbols             F      F      F      F      F           F
Date
2014-01-02  11.349146  15.44  15.45  15.28  15.42  31528500.0
2014-01-03  11.400599  15.51  15.64  15.30  15.52  46122300.0
2014-01-06  11.452051  15.58  15.76  15.52  15.72  42657600.0
2014-01-07  11.305044  15.38  15.74  15.35  15.73  54476300.0
2014-01-08  11.422651  15.54  15.71  15.51  15.60  48448300.0

F_data.tail():
Attributes  Adj Close  Close   High    Low   Open      Volume
Symbols             F      F      F      F      F           F
Date
2015-12-24  11.299311  14.31  14.37  14.25  14.35   9000100.0
2015-12-28  11.196661  14.18  14.34  14.16  14.28  13697500.0
2015-12-29  11.236141  14.23  14.30  14.15  14.28  18867800.0
2015-12-30  11.188764  14.17  14.26  14.12  14.23  13800300.0
2015-12-31  11.125596  14.09  14.16  14.04  14.14  19881000.0
```

10.3. Other Time Series Models

10.3.1. Finite Distributed Lag Models

Finite distributed lag (FDL) models allow past values of regressors to affect the dependent variable. A FDL model of order q with an independent variable z can be written as

$$y_t = \alpha_0 + \delta_0 z_t + \delta_1 z_{t-1} + \cdots + \delta_q z_{t-q} + u_t. \tag{10.2}$$

Wooldridge (2019, Section 10.2) discusses the specification and interpretation of such models. For the implementation, we generate the q additional variables that reflect the lagged values $z_{t-1}, \ldots, z_{t-q}$ and include them in the model formula of **ols**. The method **shift(k)** allows to generate the lagged variable z_{t-k}. Be aware that this only works if rows are sorted in an ascending order by the time variable. If your data frame **df** looks different and **time** is the time variable, you have to run **df.sort_values(by=['time'])** first.

Wooldridge, Example 10.4: Effects of Personal Exemption on Fertility Rates

The data set FERTIL3 contains yearly information on the general fertility rate **gfr** and the personal tax exemption **pe** for the years 1913 through 1984. Dummy variables for the second world war **ww2** and the availability of the birth control pill **pill** are also included. Script 10.4 (Example-10-4.py) shows the distributed lag model including contemporaneous **pe** and two lags. All **pe** coefficients are insignificantly different from zero according to the respective t tests. In Script 10.5 (Example-10-4-cont.py) a usual F test implemented with **f_test** reveals that they are jointly significantly different from zero at a significance level of $\alpha = 5\%$ with a p value of 0.012 (see **ftest1**). As Wooldridge (2019) discusses, this points to a multicollinearity problem.

Script 10.4: **Example-10-4.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

fertil3 = woo.dataWoo('fertil3')
T = len(fertil3)

# define yearly time series beginning in 1913:
fertil3.index = pd.date_range(start='1913', periods=T, freq='Y').year

# add all lags of 'pe' up to order 2:
fertil3['pe_lag1'] = fertil3['pe'].shift(1)
fertil3['pe_lag2'] = fertil3['pe'].shift(2)

# linear regression of model with lags:
reg = smf.ols(formula='gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill', data=fertil3)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 10.4: Example-10-4.py

```
table:
                  b        se         t    pval
Intercept   95.8705    3.2820   29.2114  0.0000
pe           0.0727    0.1255    0.5789  0.5647
pe_lag1     -0.0058    0.1557   -0.0371  0.9705
pe_lag2      0.0338    0.1263    0.2679  0.7896
ww2        -22.1265   10.7320   -2.0617  0.0433
pill       -31.3050    3.9816   -7.8625  0.0000
```

The long-run propensity (LRP) of FDL models measures the cumulative effect of a change in the independent variable z on the dependent variable y over time and is simply equal to the sum of the respective parameters

$$\text{LRP} = \delta_0 + \delta_1 + \cdots + \delta_q.$$

We can calculate it directly from the estimated regression model. For testing whether it is different from zero, we can again use the convenient **f_test** command.

Wooldridge, Example 10.4: (continued)

Script 10.5 (Example-10-4-cont.py) calculates the estimated LRP to be around 0.1. According to an F test, it is significantly different from zero with a p value of around 0.001.

Script 10.5: Example-10-4-cont.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

fertil3 = woo.dataWoo('fertil3')
T = len(fertil3)

# define yearly time series beginning in 1913:
fertil3.index = pd.date_range(start='1913', periods=T, freq='Y').year

# add all lags of 'pe' up to order 2:
fertil3['pe_lag1'] = fertil3['pe'].shift(1)
fertil3['pe_lag2'] = fertil3['pe'].shift(2)

# linear regression of model with lags:
reg = smf.ols(formula='gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill', data=fertil3)
results = reg.fit()

# F test (H0: all pe coefficients are=0):
hypotheses1 = ['pe = 0', 'pe_lag1 = 0', 'pe_lag2 = 0']
ftest1 = results.f_test(hypotheses1)
fstat1 = ftest1.statistic[0][0]
fpval1 = ftest1.pvalue

print(f'fstat1: {fstat1}\n')
print(f'fpval1: {fpval1}\n')

# calculating the LRP:
b = results.params
b_pe_tot = b['pe'] + b['pe_lag1'] + b['pe_lag2']
print(f'b_pe_tot: {b_pe_tot}\n')
```

```
# F test (H0: LRP=0):
hypotheses2 = ['pe + pe_lag1 + pe_lag2 = 0']
ftest2 = results.f_test(hypotheses2)
fstat2 = ftest2.statistic[0][0]
fpval2 = ftest2.pvalue

print(f'fstat2: {fstat2}\n')
print(f'fpval2: {fpval2}\n')
```

Output of Script 10.5: `Example-10-4-cont.py`

```
fstat1: 3.9729640469785394

fpval1: 0.011652005303126536

b_pe_tot: 0.10071909027975469

fstat2: 11.421238467853682

fpval2: 0.0012408438602970466
```

10.3.2. Trends

As pointed out by Wooldridge (2019, Section 10.5), deterministic linear (and exponential) time trends are accounted for by adding the time measure as another independent variable.

Wooldridge, Example 10.7: Housing Investment and Prices

The data set `HSEINV` provides annual observations on housing investments **`invpc`** and housing prices **`price`** for the years 1947 through 1988. Using a double-logarithmic specification, Script 10.6 (`Example-10-7.py`) estimates a regression model with and without a linear trend. The variable **`t`** is used to capture the time trend in the second regression. Forgetting to add the trend leads to the spurious finding that investments and prices are related.
Because of the logarithmic dependent variable, the trend in **`invpc`** (as opposed to log **`invpc`**) is exponential. The estimated coefficient implies a 1% yearly increase in investments.

Script 10.6: `Example-10-7.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

hseinv = woo.dataWoo('hseinv')

# linear regression without time trend:
reg_wot = smf.ols(formula='np.log(invpc) ~ np.log(price)', data=hseinv)
results_wot = reg_wot.fit()

# print regression table:
table_wot = pd.DataFrame({'b': round(results_wot.params, 4),
                          'se': round(results_wot.bse, 4),
                          't': round(results_wot.tvalues, 4),
                          'pval': round(results_wot.pvalues, 4)})
print(f'table_wot: \n{table_wot}\n')
```

```
# linear regression with time trend (data set includes a time variable t):
reg_wt = smf.ols(formula='np.log(invpc) ~ np.log(price) + t', data=hseinv)
results_wt = reg_wt.fit()

# print regression table:
table_wt = pd.DataFrame({'b': round(results_wt.params, 4),
                         'se': round(results_wt.bse, 4),
                         't': round(results_wt.tvalues, 4),
                         'pval': round(results_wt.pvalues, 4)})
print(f'table_wt: \n{table_wt}\n')
```

Output of Script 10.6: Example-10-7.py

```
table_wot:
                    b      se        t    pval
Intercept     -0.5502  0.0430 -12.7882  0.0000
np.log(price)  1.2409  0.3824   3.2450  0.0024

table_wt:
                    b      se       t    pval
Intercept     -0.9131  0.1356 -6.7328  0.0000
np.log(price) -0.3810  0.6788 -0.5612  0.5779
t              0.0098  0.0035  2.7984  0.0079
```

10.3.3. Seasonality

To account for seasonal effects, we add dummy variables for all but one (the reference) "season". So with monthly data, we can include eleven dummies, see Chapter 7 for a detailed discussion.

Wooldridge, Example 10.11: Effects of Antidumping Filings

The data in BARIUM were used in an antidumping case. They are monthly data on barium chloride imports from China between February 1978 and December 1988. Wooldridge (2019, Example 10.5) explains the data and background. When we estimate a model with monthly dummies, they do not have significant coefficients except the dummy for April which is marginally significant. An F test which is not reported reveals no joint significance.

Script 10.7: `Example-10-11.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

barium = woo.dataWoo('barium')

# linear regression with seasonal effects:
reg = smf.ols(formula='np.log(chnimp) ~ np.log(chempi) + np.log(gas) +'
                      'np.log(rtwex) + befile6 + affile6 + afdec6 +'
                      'feb + mar + apr + may + jun + jul +'
                      'aug + sep + oct + nov + dec',
              data=barium)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 10.7: `Example-10-11.py`

```
table:
                      b       se       t    pval
Intercept       16.7792  32.4286  0.5174  0.6059
np.log(chempi)   3.2651   0.4929  6.6238  0.0000
np.log(gas)     -1.2781   1.3890 -0.9202  0.3594
np.log(rtwex)    0.6630   0.4713  1.4068  0.1622
befile6          0.1397   0.2668  0.5236  0.6016
affile6          0.0126   0.2787  0.0453  0.9639
afdec6          -0.5213   0.3019 -1.7264  0.0870
feb             -0.4177   0.3044 -1.3720  0.1728
mar              0.0591   0.2647  0.2231  0.8239
apr             -0.4515   0.2684 -1.6822  0.0953
may              0.0333   0.2692  0.1237  0.9018
jun             -0.2063   0.2693 -0.7663  0.4451
jul              0.0038   0.2788  0.0138  0.9890
aug             -0.1571   0.2780 -0.5650  0.5732
sep             -0.1342   0.2677 -0.5012  0.6172
oct              0.0517   0.2669  0.1937  0.8467
nov             -0.2463   0.2628 -0.9370  0.3508
dec              0.1328   0.2714  0.4894  0.6255
```

11. Further Issues in Using OLS with Time Series Data

This chapter introduces important concepts for time series analyses. Section 11.1 discusses the general conditions under which asymptotic analyses work with time series data. An important requirement will be that the time series exhibit weak dependence. In Section 11.2, we study highly persistent time series and present some simulation excercises. One solution to this problem is first differencing as demonstrated in Section 11.3. How this can be done in the regression framework is the topic of Section 11.4.

11.1. Asymptotics with Time Series

As Wooldridge (2019, Section 11.2) discusses, asymptotic arguments also work with time series data under certain conditions. Importantly, we have to assume that the data are stationary and weakly dependent (Assumption TS.1). On the other hand, we can relax the strict exogeneity assumption TS.3 and only have to assume contemporaneous exogeneity (Assumption TS.3'). Under the appropriate set of assumptions, we can use standard OLS estimation and inference.

Wooldridge, Example 11.4: Efficient Markets Hypothesis

The efficient markets hypothesis claims that we cannot predict stock returns from past returns. In a simple AR(1) model in which returns are regressed on lagged returns, this would imply a population slope coefficient of zero. The data set `NYSE` contains data on weekly stock returns.
Script 11.1 (`Example-11-4.py`) shows the analyses. Regression 1 is the AR(1) model also discussed by Wooldridge (2019). Models 2 and 3 add second and third lags to estimate higher-order AR(p) models. In all models, no lagged value has a significant coefficient and also the F tests for joint significance (not included in the script) do not reject the efficient markets hypothesis.

Script 11.1: Example-11-4.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

nyse = woo.dataWoo('nyse')
nyse['ret'] = nyse['return']

# add all lags up to order 3:
nyse['ret_lag1'] = nyse['ret'].shift(1)
nyse['ret_lag2'] = nyse['ret'].shift(2)
nyse['ret_lag3'] = nyse['ret'].shift(3)

# linear regression of model with lags:
reg1 = smf.ols(formula='ret ~ ret_lag1', data=nyse)
reg2 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2', data=nyse)
reg3 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2 + ret_lag3', data=nyse)
results1 = reg1.fit()
results2 = reg2.fit()
results3 = reg3.fit()

# print regression tables:
table1 = pd.DataFrame({'b': round(results1.params, 4),
                       'se': round(results1.bse, 4),
                       't': round(results1.tvalues, 4),
                       'pval': round(results1.pvalues, 4)})
print(f'table1: \n{table1}\n')

table2 = pd.DataFrame({'b': round(results2.params, 4),
                       'se': round(results2.bse, 4),
                       't': round(results2.tvalues, 4),
                       'pval': round(results2.pvalues, 4)})
print(f'table2: \n{table2}\n')

table3 = pd.DataFrame({'b': round(results3.params, 4),
                       'se': round(results3.bse, 4),
                       't': round(results3.tvalues, 4),
                       'pval': round(results3.pvalues, 4)})
print(f'table3: \n{table3}\n')
```

Output of Script 11.1: Example-11-4.py

```
table1:
                b      se       t    pval
Intercept  0.1796  0.0807  2.2248  0.0264
ret_lag1   0.0589  0.0380  1.5490  0.1218

table2:
                b      se       t    pval
Intercept  0.1857  0.0812  2.2889  0.0224
ret_lag1   0.0603  0.0382  1.5799  0.1146
ret_lag2  -0.0381  0.0381 -0.9982  0.3185

table3:
                b      se       t    pval
Intercept  0.1794  0.0816  2.1990  0.0282
ret_lag1   0.0614  0.0382  1.6056  0.1088
ret_lag2  -0.0403  0.0383 -1.0519  0.2932
ret_lag3   0.0307  0.0382  0.8038  0.4218
```

We can do a similar analysis for daily data. The module **pandas_datareader** introduced in Section 1.3.3 allows us to directly download daily stock prices from Yahoo Finance. Script 11.2 (`Example-EffMkts.py`) downloads daily stock prices of Apple (ticker symbol `AAPL`) and stores them as a **DataFrame** object. From the prices p_t, daily returns r_t are calculated using the standard formula

$$r_t = \log(p_t) - \log(p_{t-1}) \approx \frac{p_t - p_{t-1}}{p_{t-1}}.$$

Note that in the script, we calculate the difference using the method **diff**. It calculates the difference from trading day to trading day, ignoring the fact that some of them are separated by weekends or holidays. Obviously, this procedure only works, if two consecutive rows represent two consecutive points in time. Figure 11.1 plots the returns of the Apple stock. Even though we now have $n = 2267$ observations of daily returns, we cannot find any relation between current and past returns which supports (this version of) the efficient markets hypothesis.

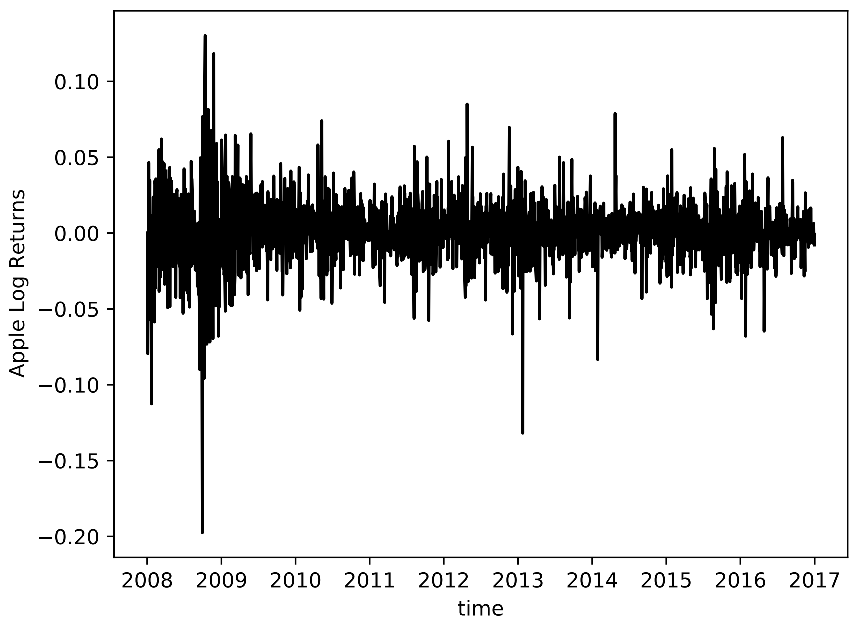
Script 11.2: `Example-EffMkts.py`

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

# download data for 'AAPL' (= Apple) and define start and end:
tickers = ['AAPL']
start_date = '2007-12-31'
end_date = '2016-12-31'

# use pandas_datareader for the import:
AAPL_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# drop ticker symbol from column name:
AAPL_data.columns = AAPL_data.columns.droplevel(level=1)

# calculate return as the log difference:
AAPL_data['ret'] = np.log(AAPL_data['Adj Close']).diff()

# time series plot of adjusted closing prices:
plt.plot('ret', data=AAPL_data, color='black', linestyle='-')
plt.ylabel('Apple Log Returns')
plt.xlabel('time')
plt.savefig('PyGraphs/Example-EffMkts.pdf')

# linear regression of models with lags:
AAPL_data['ret_lag1'] = AAPL_data['ret'].shift(1)
AAPL_data['ret_lag2'] = AAPL_data['ret'].shift(2)
AAPL_data['ret_lag3'] = AAPL_data['ret'].shift(3)

reg1 = smf.ols(formula='ret ~ ret_lag1', data=AAPL_data)
reg2 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2', data=AAPL_data)
reg3 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2 + ret_lag3', data=AAPL_data)
results1 = reg1.fit()
results2 = reg2.fit()
results3 = reg3.fit()

# print regression tables:
table1 = pd.DataFrame({'b': round(results1.params, 4),
                       'se': round(results1.bse, 4),
                       't': round(results1.tvalues, 4),
                       'pval': round(results1.pvalues, 4)})
print(f'table1: \n{table1}\n')

table2 = pd.DataFrame({'b': round(results2.params, 4),
                       'se': round(results2.bse, 4),
                       't': round(results2.tvalues, 4),
                       'pval': round(results2.pvalues, 4)})
print(f'table2: \n{table2}\n')

table3 = pd.DataFrame({'b': round(results3.params, 4),
                       'se': round(results3.bse, 4),
                       't': round(results3.tvalues, 4),
                       'pval': round(results3.pvalues, 4)})
print(f'table3: \n{table3}\n')
```

Output of Script 11.2: `Example-EffMkts.py`

```
table1:
                 b      se       t    pval
Intercept   0.0007  0.0004  1.5667  0.1173
ret_lag1   -0.0034  0.0210 -0.1628  0.8707

table2:
                 b      se       t    pval
Intercept   0.0007  0.0004  1.6107  0.1074
ret_lag1   -0.0035  0.0210 -0.1677  0.8668
ret_lag2   -0.0288  0.0210 -1.3722  0.1701

table3:
                 b      se       t    pval
Intercept   0.0007  0.0004  1.6909  0.0910
ret_lag1   -0.0034  0.0210 -0.1618  0.8715
ret_lag2   -0.0303  0.0210 -1.4451  0.1486
ret_lag3    0.0054  0.0210  0.2569  0.7973
```

Figure 11.1. Time Series Plot: Daily Stock Returns 2008–2016, Apple Inc.



11.2. The Nature of Highly Persistent Time Series

The simplest model for highly persistent time series is a random walk. It can be written as

$$y_t = y_{t-1} + e_t \tag{11.1}$$
$$= y_0 + e_1 + e_2 + \cdots + e_{t-1} + e_t \tag{11.2}$$

where the shocks $e_1, \ldots, e_t$ are i.i.d with a zero mean. It is a special case of a unit root process. Random walk processes are strongly dependent and nonstationary, violating assumption TS1′ required for the consistency of OLS parameter estimates. As Wooldridge (2019, Section 11.3) shows, the variance of y_t (conditional on y_0) increases linearly with t:

$$\text{Var}(y_t|y_0) = \sigma_e^2 \cdot t. \tag{11.3}$$

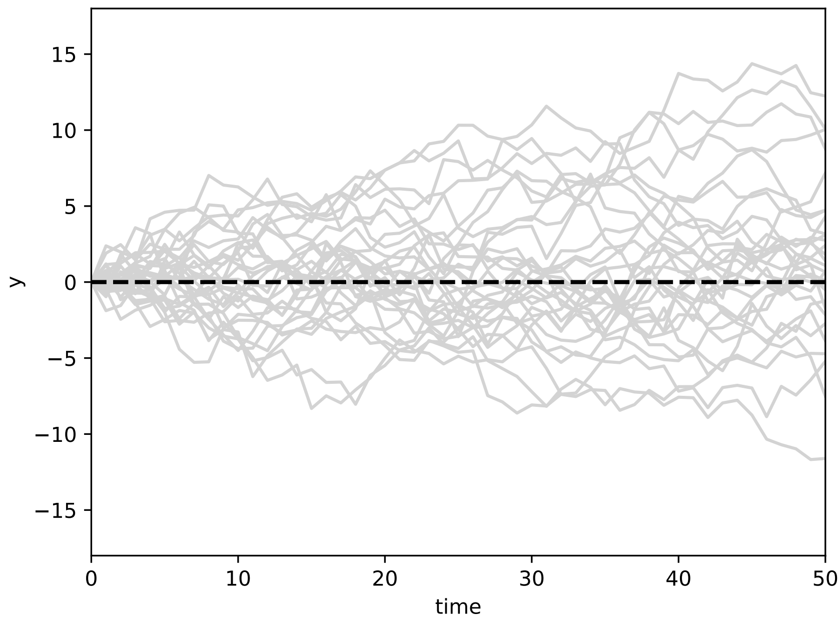
This can be easily seen in a simulation exercise. Script 11.3 (`Simulate-RandomWalk.py`) draws 30 realizations from a random walk process with i.i.d. standard normal shocks e_t. After initializing the random number generator, an empty figure with the right dimensions is produced. Then, the realizations of the time series are drawn in a loop.[1] In each of the 30 draws, we first obtain a sample of the $n = 50$ shocks $e_1, \ldots, e_{50}$. The random walk is generated as the cumulative sum of the shocks according to Equation 11.2 with an initial value of $y_0 = 0$. The respective time series are then added to the plot. In the resulting Figure 11.2, the increasing variance can be seen easily.

Script 11.3: `Simulate-RandomWalk.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# initialize plot:
x_range = np.linspace(0, 50, num=51)
plt.ylim([-18, 18])
plt.xlim([0, 50])

# loop over draws:
for r in range(0, 30):
    # i.i.d. standard normal shock:
    e = stats.norm.rvs(0, 1, size=51)

    # set first entry to 0 (gives y_0 = 0):
    e[0] = 0

    # random walk as cumulative sum of shocks:
    y = np.cumsum(e)

    # add line to graph:
    plt.plot(x_range, y, color='lightgrey', linestyle='-')

plt.axhline(linewidth=2, linestyle='--', color='black')
plt.ylabel('y')
plt.xlabel('time')
plt.savefig('PyGraphs/Simulate-RandomWalk.pdf')
```

[1] For a review of random number generation, see Section 1.6.4.

Figure 11.2. Simulations of a Random Walk Process



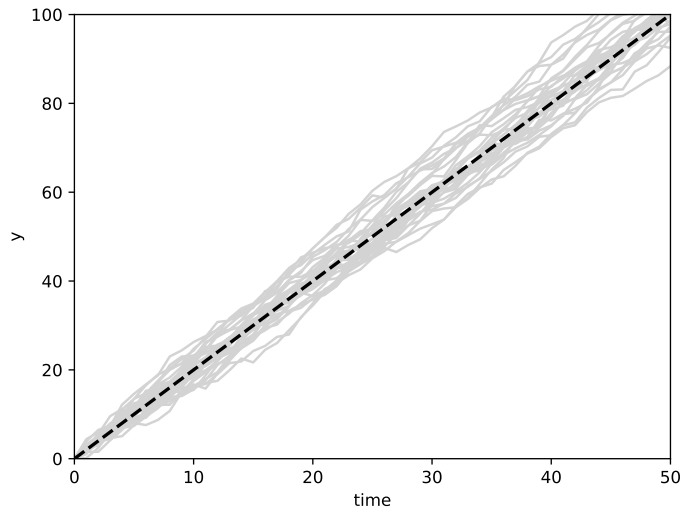
A simple generalization is a random walk with drift:

$$y_t = \alpha_0 + y_{t-1} + e_t \tag{11.4}$$

$$= y_0 + \alpha_0 \cdot t + e_1 + e_2 + \cdots + e_{t-1} + e_t. \tag{11.5}$$

Script 11.4 (`Simulate-RandomWalkDrift.py`) simulates such a process with $\alpha_0 = 2$ and i.i.d. standard normal shocks e_t. The resulting time series are plotted in Figure 11.3. The values fluctuate around the expected value $\alpha_0 \cdot t$. But unlike weakly dependent processes, they do not tend towards their mean, so the variance increases like for a simple random walk process.

Figure 11.3. Simulations of a Random Walk Process with Drift



Script 11.4: `Simulate-RandomWalkDrift.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# initialize plot:
x_range = np.linspace(0, 50, num=51)
plt.ylim([0, 100])
plt.xlim([0, 50])

# loop over draws:
for r in range(0, 30):
    # i.i.d. standard normal shock:
    e = stats.norm.rvs(0, 1, size=51)

    # set first entry to 0 (gives y_0 = 0):
    e[0] = 0

    # random walk as cumulative sum of shocks plus drift:
    y = np.cumsum(e) + 2 * x_range

    # add line to graph:
    plt.plot(x_range, y, color='lightgrey', linestyle='-')

plt.plot(x_range, 2 * x_range, linewidth=2, linestyle='--', color='black')
plt.ylabel('y')
plt.xlabel('time')
plt.savefig('PyGraphs/Simulate-RandomWalkDrift.pdf')
```

An obvious question is whether a given sample is from a unit root process such as a random walk. We will cover tests for unit roots in Section 18.2.

11.3. Differences of Highly Persistent Time Series

The simplest way to deal with highly persistent time series is to work with their differences rather than their levels. The first difference of the random walk with drift is

$$y_t = \alpha_0 + y_{t-1} + e_t \tag{11.6}$$

$$\Delta y_t = y_t - y_{t-1} = \alpha_0 + e_t \tag{11.7}$$

This is an i.i.d. process with mean α_0. Script 11.5 (`Simulate-RandomWalkDrift-Diff.py`) repeats the same simulation as Script 11.4 (`Simulate-RandomWalkDrift.py`) but calculates the differences using **`y[1:51] - y[0:50]`**. From now on, we will use the more convenient method **`diff`** for the same task. The resulting series are shown in Figure 11.4. They have a constant mean of 2, a constant variance of $\sigma_e^2 = 1$, and are independent over time.

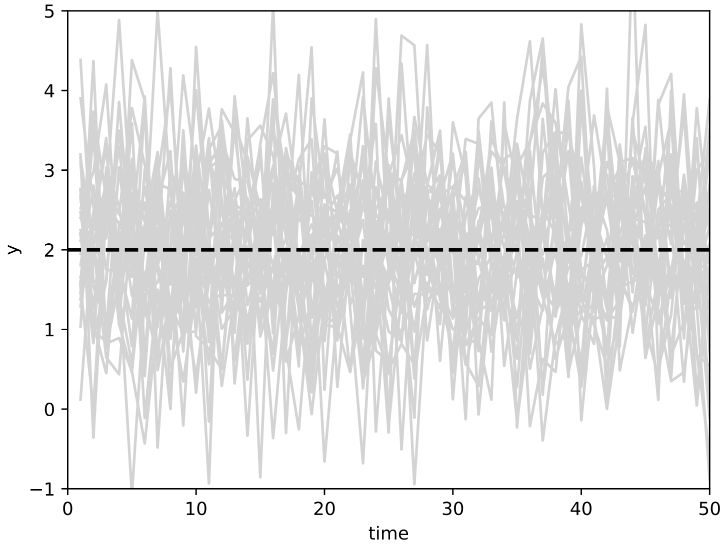
Script 11.5: `Simulate-RandomWalkDrift-Diff.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# initialize plot:
x_range = np.linspace(1, 50, num=50)
plt.ylim([-1, 5])
plt.xlim([0, 50])

# loop over draws:
for r in range(0, 30):
    # i.i.d. standard normal shock and cumulative sum of shocks:
    e = stats.norm.rvs(0, 1, size=51)
    e[0] = 0
    y = np.cumsum(2 + e)

    # first difference:
    Dy = y[1:51] - y[0:50]

    # add line to graph:
    plt.plot(x_range, Dy, color='lightgrey', linestyle='-')

plt.axhline(y=2, linewidth=2, linestyle='--', color='black')
plt.ylabel('y')
plt.xlabel('time')
plt.savefig('PyGraphs/Simulate-RandomWalkDrift-Diff.pdf')
```

11.4. Regression with First Differences

Adding first differences to regression models is straightforward. You have to add the dependent or independent variable **`var`** as a first difference to your data before starting the usual **`ols`** command. The same holds, if you want to combine differences with lags in your specifications. This is demonstrated in Example 11.6.

Figure 11.4. Simulations of a Random Walk Process with Drift: First Differences



As already mentioned, the methods **shift** and **diff** are helpful, but they require that consecutive rows represent two consecutive points in time. These commands do not use any time stamp you may have provided before.

Wooldridge, Example 11.6: Fertility Equation

We continue Example 10.4 and specify the fertility equation in first differences. Script 11.6 (`Example-11-6.py`) shows the analyses. While the first difference of the tax exemptions has no significant effect, its second lag has a significantly positive coefficient in the second model. This is consistent with fertility reacting two years after a change of the tax code.

Script 11.6: `Example-11-6.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

fertil3 = woo.dataWoo('fertil3')
T = len(fertil3)

# define time series (years only) beginning in 1913:
fertil3.index = pd.date_range(start='1913', periods=T, freq='Y').year

# compute first differences:
fertil3['gfr_diff1'] = fertil3['gfr'].diff()
fertil3['pe_diff1'] = fertil3['pe'].diff()
print(f'fertil3.head(): \n{fertil3.head()}\n')

# linear regression of model with first differences:
reg1 = smf.ols(formula='gfr_diff1 ~ pe_diff1', data=fertil3)
results1 = reg1.fit()
```

```
# print regression table:
table1 = pd.DataFrame({'b': round(results1.params, 4),
                       'se': round(results1.bse, 4),
                       't': round(results1.tvalues, 4),
                       'pval': round(results1.pvalues, 4)})
print(f'table1: \n{table1}\n')

# linear regression of model with lagged differences:
fertil3['pe_diff1_lag1'] = fertil3['pe_diff1'].shift(1)
fertil3['pe_diff1_lag2'] = fertil3['pe_diff1'].shift(2)

reg2 = smf.ols(formula='gfr_diff1 ~ pe_diff1 + pe_diff1_lag1 + pe_diff1_lag2',
               data=fertil3)
results2 = reg2.fit()

# print regression table:
table2 = pd.DataFrame({'b': round(results2.params, 4),
                       'se': round(results2.bse, 4),
                       't': round(results2.tvalues, 4),
                       'pval': round(results2.pvalues, 4)})
print(f'table2: \n{table2}\n')
```

Output of Script 11.6: `Example-11-6.py`

```
fertil3.head():
             gfr     pe  year  t  ...  cgfr_4       gfr_2  gfr_diff1  pe_diff1
1913  124.699997   0.00  1913  1  ...     NaN         NaN        NaN       NaN
1914  126.599998   0.00  1914  2  ...     NaN         NaN   1.900002      0.00
1915  125.000000   0.00  1915  3  ...     NaN  124.699997  -1.599998      0.00
1916  123.400002   0.00  1916  4  ...     NaN  126.599998  -1.599998      0.00
1917  121.000000  19.27  1917  5  ...     NaN  125.000000  -2.400002     19.27

[5 rows x 26 columns]

table1:
                b      se       t    pval
Intercept -0.7848  0.5020 -1.5632  0.1226
pe_diff1  -0.0427  0.0284 -1.5045  0.1370

table2:
                    b      se       t    pval
Intercept     -0.9637  0.4678 -2.0602  0.0434
pe_diff1      -0.0362  0.0268 -1.3522  0.1810
pe_diff1_lag1 -0.0140  0.0276 -0.5070  0.6139
pe_diff1_lag2  0.1100  0.0269  4.0919  0.0001
```

12. Serial Correlation and Heteroscedasticity in Time Series Regressions

In Chapter 8, we discussed the consequences of heteroscedasticity in cross-sectional regressions. In the time series setting, similar consequences and strategies apply to both heteroscedasticity (with some specific features) and serial correlation of the error term. Unbiasedness and consistency of the OLS estimators are unaffected. But the OLS estimators are inefficient and the usual standard errors and inferences are invalid.

We first discuss how to test for serial correlation in Section 12.1. Section 12.2 introduces efficient estimation using feasible GLS estimators. As an alternative, we can still use OLS and calculate standard errors that are valid under both heteroscedasticity and autocorrelation as discussed in Section 12.3. Finally, Section 12.4 covers heteroscedasticity and autoregressive conditional heteroscedasticity (ARCH) models.

12.1. Testing for Serial Correlation of the Error Term

Suppose we are worried that the error terms $u_1, u_2, \ldots$ in a regression model of the form

$$y_t = \beta_0 + \beta_1 x_{t1} + \beta_2 x_{t2} + \cdots + \beta_k x_{tk} + u_t \tag{12.1}$$

are serially correlated. A straightforward and intuitive testing approach is described by Wooldridge (2019, Section 12.3). It is based on the fitted residuals $\hat{u}_t = y_t - \hat{\beta}_0 - \hat{\beta}_1 x_{t1} - \cdots - \hat{\beta}_k x_{tk}$ which can be obtained in **`statsmodels`** with the attribute **`resid`**, see Section 2.2.

To test for AR(1) serial correlation under strict exogeneity, we regress $\hat{u}_t$ on their lagged values $\hat{u}_{t-1}$. If the regressors are not necessarily strictly exogenous, we can adjust the test by adding the original regressors $x_{t1}, \ldots, x_{tk}$ to this regression. Then we perform the usual t test on the coefficient of $\hat{u}_{t-1}$.

For testing for higher order serial correlation, we add higher order lags $\hat{u}_{t-2}, \hat{u}_{t-3}, \ldots$ as explanatory variables and test the joint hypothesis that they are all equal to zero using either an F test or a Lagrange multiplier (LM) test. Especially the latter version is often called Breusch-Godfrey test.

Wooldridge, Example 12.2: Testing for AR(1) Serial Correlation

We use this example to demonstrate the "pedestrian" way to test for autocorrelation which is actually straightforward and instructive. We estimate two versions of the Phillips curve: a static model

$$\texttt{inf}_t = \beta_0 + \beta_1 \texttt{unem}_t + u_t$$

and an expectation-augmented Phillips curve

$$\Delta\texttt{inf}_t = \beta_0 + \beta_1 \texttt{unem}_t + u_t.$$

Scripts 12.1 (`Example-12-2-Static.py`) and 12.2 (`Example-12-2-ExpAug.py`) show the analyses. After the estimation, the residuals are extracted with **`resid`** and regressed on their lagged values. We report standard errors and t statistics. While there is strong evidence for autocorrelation in the static equation with a t statistic of $\frac{0.573}{0.116} \approx 4.93$, the null hypothesis of no autocorrelation cannot be rejected in the second model with a t statistic of $\frac{-0.036}{0.124} \approx -0.29$.

Script 12.1: `Example-12-2-Static.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

phillips = woo.dataWoo('phillips')
T = len(phillips)

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=T, freq='Y')
phillips.index = date_range.year

# estimation of static Phillips curve:
yt96 = (phillips['year'] <= 1996)
reg_s = smf.ols(formula='Q("inf") ~ unem', data=phillips, subset=yt96)
results_s = reg_s.fit()

# residuals and AR(1) test:
phillips['resid_s'] = results_s.resid
phillips['resid_s_lag1'] = phillips['resid_s'].shift(1)
reg = smf.ols(formula='resid_s ~ resid_s_lag1', data=phillips, subset=yt96)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 12.1: `Example-12-2-Static.py`

```
table:
                   b      se        t    pval
Intercept    -0.1134  0.3594 -0.3155  0.7538
resid_s_lag1  0.5730  0.1161  4.9337  0.0000
```

Script 12.2: **Example-12-2-ExpAug.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

phillips = woo.dataWoo('phillips')
T = len(phillips)

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=T, freq='Y')
phillips.index = date_range.year

# estimation of expectations-augmented Phillips curve:
yt96 = (phillips['year'] <= 1996)
phillips['inf_diff1'] = phillips['inf'].diff()
reg_ea = smf.ols(formula='inf_diff1 ~ unem', data=phillips, subset=yt96)
results_ea = reg_ea.fit()

phillips['resid_ea'] = results_ea.resid
phillips['resid_ea_lag1'] = phillips['resid_ea'].shift(1)
reg = smf.ols(formula='resid_ea ~ resid_ea_lag1', data=phillips, subset=yt96)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 12.2: **Example-12-2-ExpAug.py**

```
table:
                    b      se       t    pval
Intercept      0.1942  0.3004  0.6464  0.5213
resid_ea_lag1 -0.0356  0.1239 -0.2873  0.7752
```

This class of tests can also be performed automatically in **statsmodels**. Given the regression results are stored in a variable **results**, the LM and F tests of AR(**q**) serial correlation can simply be tested using

```
stats.diagnostic.acorr_breusch_godfrey(results, nlags=q)
```

Wooldridge, Example 12.4: Testing for AR(3) Serial Correlation

We already used the monthly data set `BARIUM` and estimated a model for barium chloride imports in Example 10.11. Script 12.3 (`Example-12-4.py`) estimates the model and tests for AR(3) serial correlation using the manual regression approach and the command **acorr_breusch_godfrey**. The manual approach gives exactly the results reported by Wooldridge (2019) while the built-in command differs very slightly because of a different implementation (for details, see the module documentation).

Script 12.3: `Example-12-4.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

barium = woo.dataWoo('barium')
T = len(barium)

# monthly time series starting Feb. 1978:
barium.index = pd.date_range(start='1978-02', periods=T, freq='M')

reg = smf.ols(formula='np.log(chnimp) ~ np.log(chempi) + np.log(gas) +'
                      'np.log(rtwex) + befile6 + affile6 + afdec6',
              data=barium)
results = reg.fit()

# automatic test:
bg_result = sm.stats.diagnostic.acorr_breusch_godfrey(results, nlags=3)
fstat_auto = bg_result[2]
fpval_auto = bg_result[3]
print(f'fstat_auto: {fstat_auto}\n')
print(f'fpval_auto: {fpval_auto}\n')

# pedestrian test:
barium['resid'] = results.resid
barium['resid_lag1'] = barium['resid'].shift(1)
barium['resid_lag2'] = barium['resid'].shift(2)
barium['resid_lag3'] = barium['resid'].shift(3)

reg_manual = smf.ols(formula='resid ~ resid_lag1 + resid_lag2 + resid_lag3 +'
                             'np.log(chempi) + np.log(gas) + np.log(rtwex) +'
                             'befile6 + affile6 + afdec6', data=barium)
results_manual = reg_manual.fit()

hypotheses = ['resid_lag1 = 0', 'resid_lag2 = 0', 'resid_lag3 = 0']
ftest_manual = results_manual.f_test(hypotheses)
fstat_manual = ftest_manual.statistic[0][0]
fpval_manual = ftest_manual.pvalue
print(f'fstat_manual: {fstat_manual}\n')
print(f'fpval_manual: {fpval_manual}\n')
```

Output of Script 12.3: `Example-12-4.py`

```
fstat_auto: 5.124662239772493

fpval_auto: 0.0022637197671316277

fstat_manual: 5.122907054069368

fpval_manual: 0.0022898028329663344
```

Another popular test is the Durbin-Watson test for AR(1) serial correlation. While the test statistic is pretty straightforward to compute, its distribution is non-standard and depends on the data. **`statsmodels`** includes the test statistic in the output of the **`summary`** command or offers the command **`durbin_watson`**. The test statistic ranges from 0 to 4, where 2 represents the case of no serial correlation. A value towards 0 indicates positive serial correlation, a value towards 4 negative serial correlation. Given the CLM assumptions, p values can be calculated but they are not included in the output of this function. Instead we use the critical values reported in Wooldridge (2019) to perform the hypothesis tests.

Script 12.4 (`Example-DWtest.py`) repeats Example 12.2 but conducts DW tests instead of the t tests. The conclusions are the same: For the static model, no serial correlation can be rejected at a 1% level with a test statistic of $DW = 0.8027$, because it is below the critical value of $d_L = 1.32$. For the expectation augmented Phillips curve, the null hypothesis cannot be rejected at a 5% level because $DW = 1.7696$ is greater than $d_U = 1.59$.

Script 12.4: `Example-DWtest.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

phillips = woo.dataWoo('phillips')
T = len(phillips)

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=T, freq='Y')
phillips.index = date_range.year

# estimation of both Phillips curve models:
yt96 = (phillips['year'] <= 1996)
phillips['inf_diff1'] = phillips['inf'].diff()
reg_s = smf.ols(formula='Q("inf") ~ unem', data=phillips, subset=yt96)
reg_ea = smf.ols(formula='inf_diff1 ~ unem', data=phillips, subset=yt96)
results_s = reg_s.fit()
results_ea = reg_ea.fit()

# DW tests:
DW_s = sm.stats.stattools.durbin_watson(results_s.resid)
DW_ea = sm.stats.stattools.durbin_watson(results_ea.resid)
print(f'DW_s: {DW_s}\n')
print(f'DW_ea: {DW_ea}\n')
```

Output of Script 12.4: `Example-DWtest.py`

```
DW_s: 0.802700467848626

DW_ea: 1.7696478574549568
```

12.2. FGLS Estimation

There are several ways to implement the FGLS methods for serially correlated error terms in *Python*. A simple way is provided by the module **statsmodels** with its command **GLSAR**. It expects matrices of dependent and independent variables and reports the Cochrane-Orcutt estimator as demonstrated in Example 12.5.

Wooldridge, Example 12.5: Cochrane-Orcutt Estimation

We once again use the monthly data set BARIUM and the same model as before. Script 12.5 (Example-12-5.py) estimates the model with OLS and then calls **GLSAR**. As expected, the results are very close to the Prais-Winsten estimates reported by Wooldridge (2019).

Script 12.5: Example-12-5.py

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.api as sm
import patsy as pt

barium = woo.dataWoo('barium')
T = len(barium)

# monthly time series starting Feb. 1978:
barium.index = pd.date_range(start='1978-02', periods=T, freq='M')

# perform the Cochrane-Orcutt estimation (iterative procedure):
y, X = pt.dmatrices('np.log(chnimp) ~ np.log(chempi) + np.log(gas) +'
                    'np.log(rtwex) + befile6 + affile6 + afdec6',
                    data=barium, return_type='dataframe')
reg = sm.GLSAR(y, X)
CORC_results = reg.iterative_fit(maxiter=100)
table = pd.DataFrame({'b_CORC': CORC_results.params,
                      'se_CORC': CORC_results.bse})
print(f'reg.rho: {reg.rho}\n')
print(f'table: \n{table}\n')
```

Output of Script 12.5: Example-12-5.py

```
reg.rho: [0.29585313]

table:
                   b_CORC    se_CORC
Intercept      -37.512978  23.239015
np.log(chempi)   2.945448   0.647696
np.log(gas)      1.063321   0.991558
np.log(rtwex)    1.138404   0.514910
befile6         -0.017314   0.321390
affile6         -0.033108   0.323806
afdec6          -0.577328   0.344075
```

12.3. Serial Correlation-Robust Inference with OLS

Unbiasedness and consistency of OLS are not affected by heteroscedasticity or serial correlation, but the standard errors are. Similar to the heteroscedasticity-robust standard errors discussed in Section 8.1, we can use a formula for the variance-covariance matrix, often referred to as Newey-West standard errors. The module **statsmodels** provides the formula in the method **fit** as the option **cov_type = 'HAC'**. The argument **cov_kwds** specifies further details like the order of considered serial correlation (labeled g in Wooldridge (2019)). After that, reported standard errors, t statistics and their p values are based on the robust variance-covariance matrix.

Wooldridge, Example 12.1: The Puerto Rican Minimum Wage

Script 12.6 (Example-12-1.py) estimates a model for the employment rate depending on the minimum wage as well as the GNP in Puerto Rico and the US. After the model has been fitted by OLS, we provide regression coefficients and standard errors using the usual variance-covariance formula. With the option **cov_type = 'HAC'** and **cov_kwds = {'maxlags': 2}**, we get the results for the HAC variance-covariance formula. Both results imply a significantly negative relation between the minimum wage and employment.

Script 12.6: Example-12-1.py

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf

prminwge = woo.dataWoo('prminwge')
T = len(prminwge)
prminwge['time'] = prminwge['year'] - 1949
prminwge.index = pd.date_range(start='1950', periods=T, freq='Y').year

# OLS regression:
reg = smf.ols(formula='np.log(prepop) ~ np.log(mincov) + np.log(prgnp) +'
                      'np.log(usgnp) + time', data=prminwge)

# results with regular SE:
results_regu = reg.fit()

# print regression table:
table_regu = pd.DataFrame({'b': round(results_regu.params, 4),
                           'se': round(results_regu.bse, 4),
                           't': round(results_regu.tvalues, 4),
                           'pval': round(results_regu.pvalues, 4)})
print(f'table_regu: \n{table_regu}\n')

# results with HAC SE:
results_hac = reg.fit(cov_type='HAC', cov_kwds={'maxlags': 2})

# print regression table:
table_hac = pd.DataFrame({'b': round(results_hac.params, 4),
                          'se': round(results_hac.bse, 4),
                          't': round(results_hac.tvalues, 4),
                          'pval': round(results_hac.pvalues, 4)})
print(f'table_hac: \n{table_hac}\n')
```

Output of Script 12.6: `Example-12-1.py`

```
table_regu:
                      b      se       t    pval
Intercept       -6.6634  1.2578 -5.2976  0.0000
np.log(mincov) -0.2123  0.0402 -5.2864  0.0000
np.log(prgnp)   0.2852  0.0805  3.5437  0.0012
np.log(usgnp)   0.4860  0.2220  2.1896  0.0357
time            -0.0267  0.0046 -5.7629  0.0000

table_hac:
                      b      se       t    pval
Intercept       -6.6634  1.4318 -4.6539  0.0000
np.log(mincov) -0.2123  0.0426 -4.9821  0.0000
np.log(prgnp)   0.2852  0.0928  3.0720  0.0021
np.log(usgnp)   0.4860  0.2601  1.8687  0.0617
time            -0.0267  0.0054 -4.9710  0.0000
```

12.4. Autoregressive Conditional Heteroscedasticity

In time series, especially in financial data, a specific form of heteroscedasticity is often present. Autoregressive conditional heteroscedasticity (ARCH) and related models try to capture these effects.

Consider a basic linear time series equation

$$y_t = \beta_0 + \beta_1 x_{t1} + \beta_2 x_{t2} + \cdots + \beta_k x_{tk} + u_t. \tag{12.2}$$

The error term u follows a ARCH process if

$$\mathrm{E}(u_t^2 | u_{t-1}, u_{t-2}, \ldots) = \alpha_0 + \alpha_1 u_{t-1}^2. \tag{12.3}$$

As the equation suggests, we can estimate α_0 and α_1 by an OLS regression of the residuals $\hat{u}_t^2$ on $\hat{u}_{t-1}^2$.

Wooldridge, Example 12.9: ARCH in Stock Returns

Script 12.7 (`Example-12-9.py`) estimates a simple AR(1) model for weekly NYSE stock returns, already studied in Example 11.4. After the squared residuals are obtained, they are regressed on their lagged values. The coefficients from this regression are estimates for α_0 and α_1.

Script 12.7: Example-12-9.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

nyse = woo.dataWoo('nyse')
nyse['ret'] = nyse['return']
nyse['ret_lag1'] = nyse['ret'].shift(1)

# linear regression of model:
reg = smf.ols(formula='ret ~ ret_lag1', data=nyse)
results = reg.fit()

# squared residuals:
nyse['resid_sq'] = results.resid ** 2
nyse['resid_sq_lag1'] = nyse['resid_sq'].shift(1)

# model for squared residuals:
ARCHreg = smf.ols(formula='resid_sq ~ resid_sq_lag1', data=nyse)
results_ARCH = ARCHreg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results_ARCH.params, 4),
                      'se': round(results_ARCH.bse, 4),
                      't': round(results_ARCH.tvalues, 4),
                      'pval': round(results_ARCH.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 12.7: Example-12-9.py

```
table:
                    b      se       t  pval
Intercept      2.9474  0.4402  6.6951   0.0
resid_sq_lag1  0.3371  0.0359  9.3767   0.0
```

As a second example, let us reconsider the daily stock returns from Script 11.2 (`Example-EffMkts.py`). We again download the daily Apple stock prices from Yahoo Finance and calculate their returns. Figure 11.1 on page 209 plots them. They show a very typical pattern for an ARCH-type of model: there are periods with high (such as fall 2008) and other periods with low volatility (fall 2010). In Script 12.8 (`Example-ARCH.py`), we estimate an AR(1) process for the squared residuals. The t statistic is larger than 8, so there is very strong evidence for autoregressive conditional heteroscedasticity.

Script 12.8: **Example-ARCH.py**

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.formula.api as smf

# download data for 'AAPL' (= Apple) and define start and end:
tickers = ['AAPL']
start_date = '2007-12-31'
end_date = '2016-12-31'

# use pandas_datareader for the import:
AAPL_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# drop ticker symbol from column name:
AAPL_data.columns = AAPL_data.columns.droplevel(level=1)

# calculate return as the difference of logged prices:
AAPL_data['ret'] = np.log(AAPL_data['Adj Close']).diff()
AAPL_data['ret_lag1'] = AAPL_data['ret'].shift(1)

# AR(1) model for returns:
reg = smf.ols(formula='ret ~ ret_lag1', data=AAPL_data)
results = reg.fit()

# squared residuals:
AAPL_data['resid_sq'] = results.resid ** 2
AAPL_data['resid_sq_lag1'] = AAPL_data['resid_sq'].shift(1)

# model for squared residuals:
ARCHreg = smf.ols(formula='resid_sq ~ resid_sq_lag1', data=AAPL_data)
results_ARCH = ARCHreg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results_ARCH.params, 4),
                      'se': round(results_ARCH.bse, 4),
                      't': round(results_ARCH.tvalues, 4),
                      'pval': round(results_ARCH.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 12.8: **Example-ARCH.py**

```
table:
                    b      se        t  pval
Intercept      0.0003  0.0000  12.1550   0.0
resid_sq_lag1  0.1722  0.0207   8.3182   0.0
```

Part III.

Advanced Topics

13. Pooling Cross-Sections Across Time: Simple Panel Data Methods

Pooled cross sections consist of random samples from the same population at different points in time. Section 13.1 introduces this type of data set and how to use it for estimating changes over time. Section 13.2 covers difference-in-differences estimators, an important application of pooled cross-sections for identifying causal effects.

Panel data resemble pooled cross sectional data in that we have observations at different points in time. The key difference is that we observe the *same* cross-sectional units, for example individuals or firms. Panel data methods require the data to be organized in a systematic way, as discussed in Section 13.3. Section 13.4 introduces the first panel data method, first differenced estimation.

13.1. Pooled Cross-Sections

If we have random samples at different points in time, this does not only increase the overall sample size and thereby the statistical precision of our analyses. It also allows to study changes over time and shed additional light on relationships between variables.

Wooldridge, Example 13.2: Changes to the Return to Education and the Gender Wage Gap

The data set `cps78_85` includes two pooled cross-sections for the years 1978 and 1985. The dummy variable `y85` is equal to one for observations in 1985 and to zero for 1978. We estimate a model for the log wage `lwage` of the form

$$\texttt{lwage} = \beta_0 + \delta_0 \texttt{y85} + \beta_1 \texttt{educ} + \delta_1(\texttt{y85} \cdot \texttt{educ}) + \beta_2 \texttt{exper} + \beta_3 \frac{\texttt{exper}^2}{100} + \beta_4 \texttt{union} + \beta_5 \texttt{female} + \delta_5(\texttt{y85} \cdot \texttt{female}) + u.$$

Note that we divide $\texttt{exper}^2$ by 100 and thereby multiply β_3 by 100 compared to the results reported in Wooldridge (2019). The parameter β_1 measures the return to education in 1978 and δ_1 is the *difference* of the return to education in 1985 relative to 1978. Likewise, β_5 is the gender wage gap in 1978 and δ_5 is the change of the wage gap.
Script 13.1 (`Example-13-2.py`) estimates the model. The return to education is estimated to have increased by $\hat{\delta}_1 = 0.0185$ and the gender wage gap decreased in absolute value from $\hat{\beta}_5 = -0.3167$ to $\hat{\beta}_5 + \hat{\delta}_5 = -0.2316$, even though this change is only marginally significant. The interpretation and implementation of interactions were covered in more detail in Section 6.1.6.

Script 13.1: `Example-13-2.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

cps78_85 = woo.dataWoo('cps78_85')

# OLS results including interaction terms:
reg = smf.ols(formula='lwage ~ y85*(educ+female) + exper +'
                      'I((exper**2)/100) + union',
              data=cps78_85)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 13.1: `Example-13-2.py`

```
table:
                          b      se        t    pval
Intercept            0.4589  0.0934   4.9111  0.0000
y85                  0.1178  0.1238   0.9517  0.3415
educ                 0.0747  0.0067  11.1917  0.0000
female              -0.3167  0.0366  -8.6482  0.0000
y85:educ             0.0185  0.0094   1.9735  0.0487
y85:female           0.0851  0.0513   1.6576  0.0977
exper                0.0296  0.0036   8.2932  0.0000
I((exper ** 2) / 100) -0.0399  0.0078  -5.1513  0.0000
union                0.2021  0.0303   6.6722  0.0000
```

13.2. Difference-in-Differences

Wooldridge (2019, Section 13.2) discusses an important type of application for pooled cross-sections. Difference-in-differences (DiD) estimators estimate the effect of a policy intervention (in the broadest sense) by comparing the change over time of an outcome of interest between an affected and an unaffected group of observations.

In a regression framework, we regress the outcome of interest on a dummy variable for the affected ("treatment") group, a dummy indicating observations after the treatment and an interaction term between both. The coefficient of this interaction term can then be a good estimator for the effect of interest, controlling for initial differences between the groups and contemporaneous changes over time.

Wooldridge, Example 13.3: Effect of a Garbage Incinerator's Location on Housing Prices

We are interested in whether and how much the construction of a new garbage incinerator affected the value of nearby houses. Script 13.2 (`Example-13-3-1.py`) uses the data set `KIELMC`. We first estimate separate models for 1978 (before there were any rumors about the new incinerator) and 1981 (when the construction began). In 1981, the houses close to the construction site were cheaper by an average of $30,688.27. But this was not only due to the new incinerator since even

in 1978, nearby houses were cheaper by an average of $\$18,824.37$. The difference of these differences $\hat{\delta} = \$30,688.27 - \$18,824.37 = \$11,863.90$ is the DiD estimator and is arguably a better indicator of the actual effect.
The DiD estimator can be obtained more conveniently using a joint regression model with the interaction term as described above. The estimator $\hat{\delta} = \$11,863.90$ can be directly seen as the coefficient of the interaction term. Conveniently, standard regression tables include t tests of the hypothesis that the actual effect is equal to zero. For a one-sided test, the p value is $\frac{1}{2} \cdot 0.113 = 0.056$, so there is some statistical evidence of a negative impact.
The DiD estimator can be improved. A logarithmic specification is more plausible since it implies a constant *percentage* effect on the house values. We can also add additional regressors to control for incidental changes in the composition of the houses traded. Script 13.3 (`Example-13-3-2.py`) implements both improvements. The model including features of the houses implies an estimated decrease in the house values of about 13.2%. This effect is also significantly different from zero.

Script 13.2: `Example-13-3-1.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

kielmc = woo.dataWoo('kielmc')

# separate regressions for 1978 and 1981:
y78 = (kielmc['year'] == 1978)
reg78 = smf.ols(formula='rprice ~ nearinc', data=kielmc, subset=y78)
results78 = reg78.fit()

y81 = (kielmc['year'] == 1981)
reg81 = smf.ols(formula='rprice ~ nearinc', data=kielmc, subset=y81)
results81 = reg81.fit()

# joint regression including an interaction term:
reg_joint = smf.ols(formula='rprice ~ nearinc * C(year)', data=kielmc)
results_joint = reg_joint.fit()

# print regression tables:
table_78 = pd.DataFrame({'b': round(results78.params, 4),
                         'se': round(results78.bse, 4),
                         't': round(results78.tvalues, 4),
                         'pval': round(results78.pvalues, 4)})
print(f'table_78: \n{table_78}\n')

table_81 = pd.DataFrame({'b': round(results81.params, 4),
                         'se': round(results81.bse, 4),
                         't': round(results81.tvalues, 4),
                         'pval': round(results81.pvalues, 4)})
print(f'table_81: \n{table_81}\n')

table_joint = pd.DataFrame({'b': round(results_joint.params, 4),
                            'se': round(results_joint.bse, 4),
                            't': round(results_joint.tvalues, 4),
                            'pval': round(results_joint.pvalues, 4)})
print(f'table_joint: \n{table_joint}\n')
```

Output of Script 13.2: Example-13-3-1.py

```
table_78:
                     b        se        t    pval
Intercept  82517.2276  2653.790  31.0941  0.0000
nearinc   -18824.3705  4744.594  -3.9675  0.0001

table_81:
                     b         se        t  pval
Intercept  101307.5136  3093.0267  32.7535   0.0
nearinc    -30688.2738  5827.7088  -5.2659   0.0

table_joint:
                                   b         se        t    pval
Intercept                 82517.2276  2726.9101  30.2603  0.0000
C(year)[T.1981]           18790.2860  4050.0650   4.6395  0.0000
nearinc                  -18824.3705  4875.3221  -3.8612  0.0001
nearinc:C(year)[T.1981] -11863.9033  7456.6462  -1.5911  0.1126
```

Script 13.3: Example-13-3-2.py

```python
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

kielmc = woo.dataWoo('kielmc')

# difference in difference (DiD):
reg_did = smf.ols(formula='np.log(rprice) ~ nearinc*C(year)', data=kielmc)
results_did = reg_did.fit()

# print regression table:
table_did = pd.DataFrame({'b': round(results_did.params, 4),
                          'se': round(results_did.bse, 4),
                          't': round(results_did.tvalues, 4),
                          'pval': round(results_did.pvalues, 4)})
print(f'table_did: \n{table_did}\n')

# DiD with control variables:
reg_didC = smf.ols(formula='np.log(rprice) ~ nearinc*C(year) + age +'
                           'I(age**2) + np.log(intst) + np.log(land) +'
                           'np.log(area) + rooms + baths',
                   data=kielmc)
results_didC = reg_didC.fit()

# print regression table:
table_didC = pd.DataFrame({'b': round(results_didC.params, 4),
                           'se': round(results_didC.bse, 4),
                           't': round(results_didC.tvalues, 4),
                           'pval': round(results_didC.pvalues, 4)})
print(f'table_didC: \n{table_didC}\n')
```

Output of Script 13.3: `Example-13-3-2.py`

```
table_did:
                              b      se         t    pval
Intercept               11.2854  0.0305  369.8386  0.0000
C(year)[T.1981]          0.1931  0.0453    4.2606  0.0000
nearinc                 -0.3399  0.0546   -6.2308  0.0000
nearinc:C(year)[T.1981] -0.0626  0.0834   -0.7508  0.4533

table_didC:
                             b      se        t    pval
Intercept               7.6517  0.4159  18.3986  0.0000
C(year)[T.1981]         0.1621  0.0285   5.6868  0.0000
nearinc                 0.0322  0.0475   0.6789  0.4977
nearinc:C(year)[T.1981] -0.1315  0.0520  -2.5305  0.0119
age                    -0.0084  0.0014  -5.9236  0.0000
I(age ** 2)             0.0000  0.0000   4.3415  0.0000
np.log(intst)          -0.0614  0.0315  -1.9500  0.0521
np.log(land)            0.0998  0.0245   4.0766  0.0001
np.log(area)            0.3508  0.0515   6.8129  0.0000
rooms                   0.0473  0.0173   2.7317  0.0067
baths                   0.0943  0.0277   3.4003  0.0008
```

13.3. Organizing Panel Data

A panel data set includes several observations at different points in time t for the same (or at least an overlapping) set of cross-sectional units i. A simple "pooled" regression model could look like

$$y_{it} = \beta_0 + \beta_1 x_{it1} + \beta_2 x_{it2} + \cdots + \beta_k x_{itk} + v_{it}; \qquad t = 1, \ldots, T; \qquad i = 1, \ldots, n, \tag{13.1}$$

where the double subscript now indicates values for individual (or other cross-sectional unit) i at time t. We could estimate this model by OLS, essentially ignoring the panel structure. But at least the assumption that the error terms are unrelated is very hard to justify since they contain unobserved individual traits that are likely to be constant or at least correlated over time. Therefore, we need specific methods for panel data.

For the calculations used by panel data methods, we have to make sure that the data set is systematically organized and the estimation routines understand its structure. Usually, a panel data set comes in a "long" form where each row of data corresponds to one combination of i and t. We have to define which observations belong together by introducing an index variable for the cross-sectional units i and preferably also the time index t.

The module **`linearmodels`** is a comprehensive collection of commands dealing with panel data. It is not part of the Anaconda distribution and you have to install it as explained in Section 1.1.3. When working with panel data in **`linearmodels`**, our first line of code always is:

```
import linearmodels as plm
```

The routines require a **`pandas`** data frame with a two-dimensional index that describe the individual and time dimensions. Suppose we have our data in a standard data frame named **`mydf`**. It includes a variable **`ivar`** indicating the cross-sectional units and a variable **`tvar`** indicating the time. To work with **`linearmodels`** we create a data frame with the command

```
mydf = mydf.set_index(['ivar', 'tvar'])
```

Let's apply this to the data set `CRIME2` discussed by Wooldridge (2019, Section 13.3). It is a balanced panel of 46 cities, properly sorted. Script 13.4 (`Example-FD.py`) imports the data set and sets the indices correctly.

Once we use routines from **`linearmodels`**, it will report the number of cross-sectional units n, the number of time units T, and the total number of observations N. For an example, look at the first part of the output in Script 13.5 (`Example-13-9.py`).

13.4. First Differenced Estimator

Wooldridge (2019, Sections 13.3 – 13.5) discusses basic unobserved effects models and their estimation by first-differencing (FD). Consider the model

$$y_{it} = \beta_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + a_i + u_{it}; \qquad t = 1, \ldots, T; \qquad i = 1, \ldots, n, \tag{13.2}$$

which differs from Equation 13.1 in that it explicitly involves an unobserved effect a_i that is constant over time (since it has no t subscript). If it is correlated with one or more of the regressors $x_{it1}, \ldots, x_{itk}$, we cannot simply ignore a_i, leave it in the composite error term $v_{it} = a_i + u_{it}$ and estimate the equation by OLS. The error term v_{it} would be related to the regressors, violating assumption MLR.4 (and MLR.4') and creating biases and inconsistencies. Note that this problem is not unique to panel data, but possible solutions are.

The first differenced (FD) estimator is based on the first difference of the whole equation:

$$\begin{aligned} \Delta y_{it} &\equiv y_{it} - y_{it-1} \\ &= \beta_1 \Delta x_{it1} + \cdots + \beta_k \Delta x_{itk} + \Delta u_{it}; \qquad t = 2, \ldots, T; \qquad i = 1, \ldots, n. \end{aligned} \tag{13.3}$$

Note that we cannot evaluate this equation for the first observation $t = 1$ for any i since the lagged values are unknown for them. The trick is that a_i drops out of the equation by differencing since it does not change over time. No matter how badly it is correlated with the regressors, it cannot hurt the estimation anymore. This estimating equation is then analyzed by OLS. We simply regress the differenced dependent variable Δy_{it} on the differenced independent variables $\Delta x_{it1}, \ldots, \Delta x_{itk}$.

Script 13.4 (`Example-FD.py`) opens the data set `CRIME2` already described above. We describe the cumbersome data preparation required for the manual estimation. Before we can use the method **`diff`** to calculate first differences of the dependent variable crime rate (**`crmrte`**) and the independent variable unemployment rate (**`unem`**), we have to make sure with **`groupby('id')`** that these calculations are performed per individual.

A list of the first five observations reveals that the differences are unavailable (**`NaN`**) for the first year of each city. The other differences are also calculated as expected. For example the change of the crime rate for city 1 is $70.11729 - 74.65756 = -4.540268$ and the change of the unemployment rate for city 2 is $5.4 - 8.1 = -2.7$. The FD estimator can now be calculated by simply applying OLS to these differenced values. The observations for the first year with missing information are automatically dropped from the estimation sample. The results show a significantly positive relation between unemployment and crime.

Script 13.4: `Example-FD.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import linearmodels as plm

crime2 = woo.dataWoo('crime2')

# create time variable dummy by converting a Boolean variable to an integer:
crime2['t'] = (crime2['year'] == 87).astype(int)  # False=0, True=1

# create an index in this balanced data set by combining two arrays:
id_tmp = np.linspace(1, 46, num=46)
crime2['id'] = np.sort(np.concatenate([id_tmp, id_tmp]))

# manually calculate first differences per entity for crmrte and unem:
crime2['crmrte_diff1'] = \
    crime2.sort_values(['id', 'year']).groupby('id')['crmrte'].diff()
crime2['unem_diff1'] = \
    crime2.sort_values(['id', 'year']).groupby('id')['unem'].diff()
var_selection = ['id', 't', 'crimes', 'unem', 'crmrte_diff1', 'unem_diff1']
print(f'crime2[var_selection].head(): \n{crime2[var_selection].head()}\n')

# estimate FD model with statmodels on differenced data:
reg_sm = smf.ols(formula='crmrte_diff1 ~ unem_diff1', data=crime2)
results_sm = reg_sm.fit()

# print results:
table_sm = pd.DataFrame({'b': round(results_sm.params, 4),
                         'se': round(results_sm.bse, 4),
                         't': round(results_sm.tvalues, 4),
                         'pval': round(results_sm.pvalues, 4)})
print(f'table_sm: \n{table_sm}\n')

# estimate FD model with linearmodels:
crime2 = crime2.set_index(['id', 'year'])
reg_plm = plm.FirstDifferenceOLS.from_formula(formula='crmrte ~ t + unem',
                                              data=crime2)
results_plm = reg_plm.fit()

# print results:
table_plm = pd.DataFrame({'b': round(results_plm.params, 4),
                          'se': round(results_plm.std_errors, 4),
                          't': round(results_plm.tstats, 4),
                          'pval': round(results_plm.pvalues, 4)})
print(f'table_plm: \n{table_plm}\n')
```

Output of Script 13.4: Example-FD.py

```
crime2[var_selection].head():
    id  t   crimes  unem  crmrte_diff1  unem_diff1
0  1.0  0  17136.0   8.2           NaN         NaN
1  1.0  1  17306.0   3.7     -4.540268        -4.5
2  2.0  0  75654.0   8.1           NaN         NaN
3  2.0  1  83960.0   5.4     -2.962654        -2.7
4  3.0  0  31352.0   9.0           NaN         NaN

table_sm:
                 b      se       t    pval
Intercept  15.4022  4.7021  3.2756  0.0021
unem_diff1  2.2180  0.8779  2.5266  0.0152

table_plm:
            b      se       t    pval
t     15.4022  4.7021  3.2756  0.0021
unem   2.2180  0.8779  2.5266  0.0152
```

Generating the differenced values and using **ols** on them is actually unnecessary. The command **FirstDifferenceOLS** shows that many lines of code can be saved by using the canned routine in **linearmodels**. All the necessary calculations are done internally. As the output of Script 13.4 (Example-FD.py) shows, the parameter estimates are therefore exactly the same as our pedestrian calculations.[1]

Wooldridge, Example 13.9: County Crime Rates in North Carolina

Script 13.5 (Example-13-9.py) analyzes the data CRIME4. We estimate the model in first differences using **linearmodels**.
Note that in this specification, all variables are automatically differenced, so they have the intuitive interpretation in the level equation. In the results reported by Wooldridge (2019), the year dummies are not differenced which only makes a difference for the interpretation of the year coefficients. We will repeat this example with "robust" standard errors in Section 14.4.

Script 13.5: Example-13-9.py

```
import wooldridge as woo
import numpy as np
import linearmodels as plm

crime4 = woo.dataWoo('crime4')
crime4 = crime4.set_index(['county', 'year'], drop=False)

# estimate FD model:
reg = plm.FirstDifferenceOLS.from_formula(
    formula='np.log(crmrte) ~ year + d83 + d84 + d85 + d86 + d87 +'
            'lprbarr + lprbconv + lprbpris + lavgsen + lpolpc',
    data=crime4)
results = reg.fit()
print(f'results: \n{results}\n')
```

[1] Note that in **linearmodels** standard errors are accessible by the attribute **std_errors** instead of **bse** in **statsmodels**.

Output of Script 13.5: `Example-13-9.py`

```
results:
                     FirstDifferenceOLS Estimation Summary
================================================================================
Dep. Variable:          np.log(crmrte)   R-squared:                        0.4326
Estimator:          FirstDifferenceOLS   R-squared (Between):              0.6003
No. Observations:                  540   R-squared (Within):               0.4281
Date:                 Wed, May 13 2020   R-squared (Overall):              0.6000
Time:                         13:04:51   Log-likelihood                    248.48
Cov. Estimator:             Unadjusted
                                         F-statistic:                      36.661
Entities:                           90   P-value                           0.0000
Avg Obs:                        7.0000   Distribution:                   F(11,529)
Min Obs:                        7.0000
Max Obs:                        7.0000   F-statistic (robust):             36.661
                                         P-value                           0.0000
Time periods:                        7   Distribution:                   F(11,529)
Avg Obs:                        90.000
Min Obs:                        90.000
Max Obs:                        90.000

                             Parameter Estimates
==============================================================================
            Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
------------------------------------------------------------------------------
year           0.0077     0.0171     0.4522     0.6513     -0.0258      0.0412
d83           -0.0999     0.0239    -4.1793     0.0000     -0.1468     -0.0529
d84           -0.1478     0.0413    -3.5806     0.0004     -0.2289     -0.0667
d85           -0.1524     0.0584    -2.6098     0.0093     -0.2671     -0.0377
d86           -0.1249     0.0760    -1.6433     0.1009     -0.2742      0.0244
d87           -0.0841     0.0940    -0.8944     0.3715     -0.2687      0.1006
lprbarr       -0.3275     0.0300    -10.924     0.0000     -0.3864     -0.2686
lprbconv      -0.2381     0.0182    -13.058     0.0000     -0.2739     -0.2023
lprbpris      -0.1650     0.0260    -6.3555     0.0000     -0.2161     -0.1140
lavgsen       -0.0218     0.0221    -0.9850     0.3251     -0.0652      0.0216
lpolpc         0.3984     0.0269     14.821     0.0000      0.3456      0.4512
==============================================================================
```

14. Advanced Panel Data Methods

In this chapter, we look into additional panel data models and methods. We start with the widely used fixed effects (FE) estimator in Section 14.1, followed by random effects (RE) in Section 14.2. The dummy variable regression and correlated random effects approaches presented in Section 14.3 can be used as alternatives and generalizations of FE. Finally, we cover robust formulas for the variance-covariance matrix and the implied "clustered" standard errors in Section 14.4. We will come back to panel data in combination with instrumental variables in Section 15.6.

14.1. Fixed Effects Estimation

We start from the same basic unobserved effects models as Equation 13.2. Instead of first differencing, we get rid of the unobserved individual effect a_i using the within transformation:

$$\begin{aligned} y_{it} &= \beta_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + a_i + u_{it}; \qquad t = 1, \ldots, T; \qquad i = 1, \ldots, n, \\ \bar{y}_i &= \beta_0 + \beta_1 \bar{x}_{i1} + \cdots + \beta_k \bar{x}_{ik} + a_i + \bar{u}_i \\ \ddot{y}_{it} = y_{it} - \bar{y}_i &= \beta_1 \ddot{x}_{it1} + \cdots + \beta_k \ddot{x}_{itk} + \ddot{u}_{it}, \end{aligned} \tag{14.1}$$

where $\bar{y}_i$ is the average of y_{it} over time for cross-sectional unit i and for the other variables accordingly. The within transformation subtracts these individual averages from the respective observations y_{it}.

The fixed effects (FE) estimator simply estimates the demeaned Equation 14.1 using pooled OLS. Instead of applying the within transformation to all variables and running **`ols`**, we can simply use **`PanelOLS`** in the module **`linearmodels`**. Demeaning is considered by adding the word **`EntityEffects`** to the formula. This has the additional advantage that the degrees of freedom are adjusted to the demeaning and the variance-covariance matrix and standard errors are adjusted accordingly.[1] We will come back to different ways to get the same estimates in Section 14.3. This is shown in Script 14.1 (`Example-14-2.py`).

Wooldridge, Example 14.2: Has the Return to Education Changed over Time?

We estimate the change of the return to education over time using a fixed effects estimator. Script 14.1 (`Example-14-2.py`) shows the implementation. The data set `WAGEPAN` is a balanced panel for $n = 545$ individuals over $T = 8$ years. It includes the index variables **`nr`** and **`year`** for individuals and years, respectively. Since **`educ`** does not change over time, we cannot estimate its overall impact and have to use **`drop_absorbed=True`** in the estimation. However, we can interact it with time dummies to see how the impact changes over time.

---

[1] The default behavior of **`linearmodels`** is to exclude the constant, because β_0 drops out of the demeaned equation. In cases you need one, you can explicitly add it by using "**`1`**" in the formula.

Script 14.1: `Example-14-2.py`

```
import wooldridge as woo
import pandas as pd
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan = wagepan.set_index(['nr', 'year'], drop=False)

# FE model estimation:
reg = plm.PanelOLS.from_formula(
    formula='lwage ~ married + union + C(year)*educ + EntityEffects',
    data=wagepan, drop_absorbed=True)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.std_errors, 4),
                      't': round(results.tstats, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 14.1: `Example-14-2.py`

```
table:
                          b      se        t    pval
C(year)[1980]        1.3625  0.0162  83.9031  0.0000
C(year)[1981]        1.3400  0.1452   9.2307  0.0000
C(year)[1982]        1.3567  0.1451   9.3481  0.0000
C(year)[1983]        1.3729  0.1452   9.4561  0.0000
C(year)[1984]        1.4468  0.1452   9.9617  0.0000
C(year)[1985]        1.4122  0.1451   9.7315  0.0000
C(year)[1986]        1.4281  0.1451   9.8404  0.0000
C(year)[1987]        1.4529  0.1452  10.0061  0.0000
married              0.0548  0.0184   2.9773  0.0029
union                0.0830  0.0194   4.2671  0.0000
C(year)[T.1981]:educ  0.0116  0.0123   0.9448  0.3448
C(year)[T.1982]:educ  0.0148  0.0123   1.2061  0.2279
C(year)[T.1983]:educ  0.0171  0.0123   1.3959  0.1628
C(year)[T.1984]:educ  0.0166  0.0123   1.3521  0.1764
C(year)[T.1985]:educ  0.0237  0.0123   1.9316  0.0535
C(year)[T.1986]:educ  0.0274  0.0123   2.2334  0.0256
C(year)[T.1987]:educ  0.0304  0.0123   2.4798  0.0132
```

14.2. Random Effects Models

We again base our analysis on the basic unobserved effects model in Equation 13.2. The random effects (RE) model assumes that the unobserved effects a_i are independent of (or at least uncorrelated with) the regressors x_{itj} for all t and $j = 1, \ldots, k$. Therefore, our main motivation for using FD or FE disappears: OLS consistently estimates the model parameters under this additional assumption.

However, like the situation with heteroscedasticity (see Section 8.3) and autocorrelation (see Section 12.2), we can obtain more efficient estimates if we take into account the structure of the variances and covariances of the error term. Wooldridge (2019, Section 14.2) shows that the GLS transformation that takes care of their special structure implied by the RE model leads to a quasi-demeaned specification

$$\mathring{y}_{it} = y_{it} - \theta\bar{y}_i = \beta_0(1-\theta) + \beta_1\mathring{x}_{it1} + \cdots + \beta_k\mathring{x}_{itk} + \mathring{v}_{it}, \tag{14.2}$$

where $\mathring{y}_{it}$ is similar to the demeaned $\ddot{y}_{it}$ from Equation 14.1 but subtracts only a fraction θ of the individual averages. The same holds for the regressors x_{itj} and the composite error term $v_{it} = a_i + u_{it}$.

The parameter $\theta = 1 - \sqrt{\frac{\sigma_u^2}{\sigma_u^2 + T\sigma_a^2}}$ depends on the variances of u_{it} and a_i and the length of the time series dimension T. It is unknown and has to be estimated. Given our experience with FD and FE estimation, it should not come as a surprise that we can estimate the RE model parameters in **linearmodels** using the command **RandomEffects**. Different versions of estimating the random effects parameter θ can be implemented and one version is saved as the attribute **theta** in the results object (see the module documentation for more details).

Unlike with FD and FE estimators, we can include variables in our model that are constant over time for each cross-sectional unit. We can use **pandas** methods to provide a list of these variables as well as of those that do not vary within each point in time.

Wooldridge, Example 14.4: A Wage Equation Using Panel Data

The data set `WAGEPAN` was already used in Example 14.2. Script 14.2 (`Example-14-4-1.py`) loads the data set and defines the panel structure. Then, we check the panel dimensions and get a list of time-constant variables using **pandas**. Therefore we calculated grouped variances and used the fact that they are zero over time or individual. With these preparations, we get estimates using OLS, RE, and FE estimators in Script 14.3 (`Example-14-4-2.py`). We use **PooledOLS**, **RandomEffects** and **PanelOLS** (with the option **EntityEffects**), respectively.

Script 14.2: Example-14-4-1.py

```
import wooldridge as woo

wagepan = woo.dataWoo('wagepan')

# print relevant dimensions for panel:
N = wagepan.shape[0]
T = wagepan['year'].drop_duplicates().shape[0]
n = wagepan['nr'].drop_duplicates().shape[0]
print(f'N: {N}\n')
print(f'T: {T}\n')
print(f'n: {n}\n')

# check non-varying variables

# (I) across time and within individuals by calculating individual
# specific variances for each variable:
isv_nr = (wagepan.groupby('nr').var() == 0)  # True, if variance is zero
# choose variables where all grouped variances are zero:
noVar_nr = isv_nr.all(axis=0)  # which cols are completely True
print(f'isv_nr.columns[noVar_nr]: \n{isv_nr.columns[noVar_nr]}\n')

# (II) across individuals within one point in time for each variable:
isv_t = (wagepan.groupby('year').var() == 0)
noVar_t = isv_t.all(axis=0)
print(f'isv_t.columns[noVar_t]: \n{isv_t.columns[noVar_t]}\n')
```

Output of Script 14.2: Example-14-4-1.py

```
N: 4360

T: 8

n: 545

isv_nr.columns[noVar_nr]:
Index(['black', 'hisp', 'educ'], dtype='object')

isv_t.columns[noVar_t]:
Index(['d81', 'd82', 'd83', 'd84', 'd85', 'd86', 'd87'], dtype='object')
```

Script 14.3: Example-14-4-2.py

```
import wooldridge as woo
import pandas as pd
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')

# estimate different models:
wagepan = wagepan.set_index(['nr', 'year'], drop=False)

reg_ols = plm.PooledOLS.from_formula(
    formula='lwage ~ educ + black + hisp + exper + I(exper**2) +'
            'married + union + C(year)', data=wagepan)
results_ols = reg_ols.fit()

reg_re = plm.RandomEffects.from_formula(
    formula='lwage ~ educ + black + hisp + exper + I(exper**2) +'
            'married + union + C(year)', data=wagepan)
results_re = reg_re.fit()

reg_fe = plm.PanelOLS.from_formula(
    formula='lwage ~ I(exper**2) + married + union +'
            'C(year) + EntityEffects', data=wagepan)
results_fe = reg_fe.fit()

# print results:
theta_hat = results_re.theta.iloc[0, 0]
print(f'theta_hat: {theta_hat}\n')

table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.std_errors, 4),
                          't': round(results_ols.tstats, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

table_re = pd.DataFrame({'b': round(results_re.params, 4),
                         'se': round(results_re.std_errors, 4),
                         't': round(results_re.tstats, 4),
                         'pval': round(results_re.pvalues, 4)})
print(f'table_re: \n{table_re}\n')
```

```
table_fe = pd.DataFrame({'b': round(results_fe.params, 4),
                         'se': round(results_fe.std_errors, 4),
                         't': round(results_fe.tstats, 4),
                         'pval': round(results_fe.pvalues, 4)})
print(f'table_fe: \n{table_fe}\n')
```

Output of Script 14.3: Example-14-4-2.py

```
theta_hat: 0.6450593029243452

table_ols:
                    b      se        t    pval
C(year)[1980]  0.0921  0.0783   1.1761  0.2396
C(year)[1981]  0.1504  0.0838   1.7935  0.0730
C(year)[1982]  0.1548  0.0893   1.7335  0.0831
C(year)[1983]  0.1541  0.0944   1.6323  0.1027
C(year)[1984]  0.1825  0.0990   1.8437  0.0653
C(year)[1985]  0.2013  0.1031   1.9523  0.0510
C(year)[1986]  0.2340  0.1068   2.1920  0.0284
C(year)[1987]  0.2659  0.1100   2.4166  0.0157
educ           0.0913  0.0052  17.4419  0.0000
black         -0.1392  0.0236  -5.9049  0.0000
hisp           0.0160  0.0208   0.7703  0.4412
exper          0.0672  0.0137   4.9095  0.0000
I(exper ** 2) -0.0024  0.0008  -2.9413  0.0033
married        0.1083  0.0157   6.8997  0.0000
union          0.1825  0.0172  10.6349  0.0000

table_re:
                    b      se       t    pval
C(year)[1980]  0.0234  0.1514  0.1546  0.8771
C(year)[1981]  0.0638  0.1601  0.3988  0.6901
C(year)[1982]  0.0543  0.1690  0.3211  0.7481
C(year)[1983]  0.0436  0.1780  0.2450  0.8065
C(year)[1984]  0.0664  0.1871  0.3551  0.7225
C(year)[1985]  0.0811  0.1961  0.4136  0.6792
C(year)[1986]  0.1152  0.2052  0.5617  0.5744
C(year)[1987]  0.1583  0.2143  0.7386  0.4602
educ           0.0919  0.0107  8.5744  0.0000
black         -0.1394  0.0480 -2.9054  0.0037
hisp           0.0217  0.0428  0.5078  0.6116
exper          0.1058  0.0154  6.8706  0.0000
I(exper ** 2) -0.0047  0.0007 -6.8623  0.0000
married        0.0638  0.0168  3.8035  0.0001
union          0.1059  0.0179  5.9289  0.0000

table_fe:
                    b      se        t    pval
C(year)[1980]  1.4260  0.0183  77.7484  0.0000
C(year)[1981]  1.5772  0.0216  72.9656  0.0000
C(year)[1982]  1.6790  0.0265  63.2583  0.0000
C(year)[1983]  1.7805  0.0333  53.4392  0.0000
C(year)[1984]  1.9161  0.0417  45.9816  0.0000
C(year)[1985]  2.0435  0.0515  39.6460  0.0000
C(year)[1986]  2.1915  0.0630  34.7714  0.0000
C(year)[1987]  2.3510  0.0762  30.8669  0.0000
I(exper ** 2) -0.0052  0.0007  -7.3612  0.0000
married        0.0467  0.0183   2.5494  0.0108
union          0.0800  0.0193   4.1430  0.0000
```

The RE estimator needs stronger assumptions to be consistent than the FE estimator. On the other hand, it is more efficient if these assumptions hold and we can include time constant regressors. A widely used test of this additional assumption is the Hausman test. It is based on the comparison between the FE and RE parameter estimates. We include an example as Script 14.4 (`Example-HausmTest.py`) in Appendix IV (p. 387), which uses the FE and RE estimates and implements a Hausman test as shown in Wooldridge (2010) (Section 10.7.3). The null hypothesis that the RE model is consistent is clearly rejected with sensible significance levels like $\alpha = 5\%$ or $\alpha = 1\%$. It also demonstrates that implementing a test on your own is a lot more cumbersome than relying completely on a module's routines.

14.3. Dummy Variable Regression and Correlated Random Effects

It turns out that we can get the FE parameter estimates in two other ways than the within transformation we used in Section 14.1. The dummy variable regression uses OLS on the original variables in Equation 13.2 instead of the transformed ones. But it adds $n-1$ dummy variables (or n dummies and removes the constant), one for each cross-sectional unit $i = 1, \ldots, n$. The simplest (although not the computationally most efficient) way to implement this in *Python* is to use the cross-sectional index as another categorical variable.

The third way to get the same results is the correlated random effects (CRE) approach. Instead of assuming that the individual effects a_i are independent of the regressors x_{itj}, we assume that they only depend on the averages over time $\bar{x}_{ij} = \frac{1}{T}\sum_{t=1}^{T} x_{itj}$:

$$a_i = \gamma_0 + \gamma_1 \bar{x}_{i1} + \cdots + \gamma_k \bar{x}_{ik} + r_i \tag{14.3}$$

$$\begin{aligned} y_{it} &= \beta_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + a_i + u_{it} \\ &= \beta_0 + \gamma_0 + \beta_1 x_{it1} + \cdots + \beta_k x_{itk} + \gamma_1 \bar{x}_{i1} + \cdots + \gamma_k \bar{x}_{ik} + r_i + u_{it}. \end{aligned} \tag{14.4}$$

If r_i is uncorrelated with the regressors, we can consistently estimate the parameters of this model using the RE estimator. In addition to the original regressors, we include their averages over time.

Script 14.5 (`Example-Dummy-CRE-1.py`) uses `WAGEPAN` again. We estimate the FE parameters using the within transformation (**`reg_we`**), the dummy variable approach (**`reg_dum`**), and the CRE approach (**`reg_cre`**). We also estimate the RE version of this model (**`reg_re`**). The results confirm that the first three methods deliver exactly the same parameter estimates, while the RE estimates differ.

Script 14.5: **Example-Dummy-CRE-1.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan['t'] = wagepan['year']
wagepan['entity'] = wagepan['nr']
wagepan = wagepan.set_index(['nr'])

# include group specific means:
wagepan['married_b'] = wagepan.groupby('nr').mean()['married']
wagepan['union_b'] = wagepan.groupby('nr').mean()['union']
wagepan = wagepan.set_index(['year'], append=True)

# estimate FE parameters in 3 different ways:
reg_we = plm.PanelOLS.from_formula(
    formula='lwage ~ married + union + C(t)*educ + EntityEffects',
    drop_absorbed=True, data=wagepan)
results_we = reg_we.fit()

reg_dum = smf.ols(
    formula='lwage ~ married + union + C(t)*educ + C(entity)',
    data=wagepan)
results_dum = reg_dum.fit()

reg_cre = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + C(t)*educ + married_b + union_b',
    data=wagepan)
results_cre = reg_cre.fit()

# compare to RE estimates:
reg_re = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + C(t)*educ',
    data=wagepan)
results_re = reg_re.fit()

var_selection = ['married', 'union', 'C(t)[T.1982]:educ']

# print results:
table = pd.DataFrame({'b_we': round(results_we.params[var_selection], 4),
                      'b_dum': round(results_dum.params[var_selection], 4),
                      'b_cre': round(results_cre.params[var_selection], 4),
                      'b_re': round(results_re.params[var_selection], 4)})
print(f'table: \n{table}\n')
```

Output of Script 14.5: **Example-Dummy-CRE-1.py**

```
table:
                      b_we   b_dum   b_cre    b_re
married             0.0548  0.0548  0.0548  0.0773
union               0.0830  0.0830  0.0830  0.1075
C(t)[T.1982]:educ   0.0148  0.0148  0.0148  0.0143
```

Given we have estimated the CRE model, it is easy to test the null hypothesis that the RE estimator is consistent. The additional assumptions needed are $\gamma_1 = \cdots = \gamma_k = 0$. They can easily be tested using an F test or the very similar Wald test as demonstrated in Script 14.6 (`Example-CRE-test-RE.py`). As you see, **linearmodels** conveniently provides the routines for these tests. Like the Hausman test, we clearly reject the null hypothesis that the RE model is appropriate with a tiny p value of about 0.0001.

Script 14.6: `Example-CRE-test-RE.py`

```
import wooldridge as woo
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan['t'] = wagepan['year']
wagepan['entity'] = wagepan['nr']
wagepan = wagepan.set_index(['nr'])

# include group specific means:
wagepan['married_b'] = wagepan.groupby('nr').mean()['married']
wagepan['union_b'] = wagepan.groupby('nr').mean()['union']
wagepan = wagepan.set_index(['year'], append=True)

# estimate CRE:
reg_cre = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + C(t)*educ  + married_b + union_b',
    data=wagepan)
results_cre = reg_cre.fit()

# RE test as an Wald test on the CRE specific coefficients:
wtest = results_cre.wald_test(formula='married_b = union_b = 0')
print(f'wtest: \n{wtest}\n')
```

Output of Script 14.6: `Example-CRE-test-RE.py`

```
wtest:
Linear Equality Hypothesis Test
H0: Linear equality constraint is valid
Statistic: 19.4058
P-value: 0.0001
Distributed: chi2(2)
```

Another advantage of the CRE approach is that we can add time-constant regressors to the model. Since we cannot control for average values $\bar{x}_{ij}$ for these variables, they have to be uncorrelated with a_i for consistent estimation of *their* coefficients. For the other coefficients of the time-varying variables, we still don't need these additional RE assumptions.

Script 14.7 (`Example-CRE-2.py`) estimates another version of the wage equation using the CRE approach. The variables **married** and **union** vary over time, so we can control for their between effects. The variables **educ**, **black**, and **hisp** do not vary. For a causal interpretation of *their* coefficients, we have to rely on uncorrelatedness with a_i. Given a_i includes intelligence and other labor market success factors, this uncorrelatedness is more plausible for some variables (like gender or race) than for other variables (like education).

Script 14.7: `Example-CRE-2.py`

```
import wooldridge as woo
import pandas as pd
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan['t'] = wagepan['year']
wagepan['entity'] = wagepan['nr']
wagepan = wagepan.set_index(['nr'])

# include group specific means:
wagepan['married_b'] = wagepan.groupby('nr').mean()['married']
wagepan['union_b'] = wagepan.groupby('nr').mean()['union']
wagepan = wagepan.set_index(['year'], append=True)

# estimate CRE paramters:
reg = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + educ +'
            'black + hisp + married_b + union_b',
    data=wagepan)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.std_errors, 4),
                      't': round(results.tstats, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 14.7: `Example-CRE-2.py`

```
table:
                b      se         t    pval
married    0.2417  0.0177   13.6772  0.0000
union      0.0700  0.0207    3.3804  0.0007
educ       0.1257  0.0023   55.4837  0.0000
black     -0.0892  0.0499   -1.7864  0.0741
hisp       0.0784  0.0426    1.8428  0.0654
married_b -0.0436  0.0450   -0.9685  0.3329
union_b    0.2105  0.0519    4.0576  0.0001
```

14.4. Robust (Clustered) Standard Errors

We argued above that under the RE assumptions, OLS is inefficient but consistent. Instead of using RE, we could simply use OLS but would have to adjust the standard errors for the fact that the composite error term $v_{it} = a_i + u_{it}$ is correlated over time because of the constant individual effect a_i. In fact, the variance-covariance matrix could be more complex than the RE assumption with i.i.d. u_{it} implies. These error terms could be serially correlated and/or heteroscedastic. This would invalidate the standard errors not only of OLS but also of FD, FE, RE, and CRE.

There is an elegant solution, especially in panels with a large cross-sectional dimension. Similar to standard errors that are robust with respect to heteroscedasticity in cross-sectional data (Section 8.1) and serial correlation in time series (Section 12.3), there are formulas for the variance-covariance matrix for panel data that are robust with respect to heteroscedasticity and *arbitrary* correlations of the error term within a cross-sectional unit (or "cluster").

These "clustered" standard errors are mentioned in Wooldridge (2019, Section 14.4 and Example 13.9). Different versions of the clustered variance-covariance matrix can be computed in **linearmodels**. Script 14.8 (`Example-13-9-ClSE.py`) repeats the FD regression from Example 13.9 and reports the adjusted standard errors. Similar to the heteroscedasticity-robust standard errors discussed in Section 8.1, there are different versions of formulas for clustered standard errors. We first use the default type (**results_default**), a clustered type without (**results_cluster**) and with a small sample correction (**results_css**). The latter uses **debiased=True** (default) to adjust the degrees of freedom when estimating the covariance.

Script 14.8: `Example-13-9-ClSE.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels as plm

crime4 = woo.dataWoo('crime4')
crime4 = crime4.set_index(['county', 'year'], drop=False)

# estimate FD model:
reg = plm.FirstDifferenceOLS.from_formula(
    formula='np.log(crmrte) ~ year + d83 + d84 + d85 + d86 + d87 +'
            'lprbarr + lprbconv + lprbpris + lavgsen + lpolpc',
    data=crime4)

# regression with standard SE:
results_default = reg.fit()

# regression with "clustered" SE:
results_cluster = reg.fit(cov_type='clustered', cluster_entity=True,
                          debiased=False)

# regression with "clustered" SE (small-sample correction):
results_css = reg.fit(cov_type='clustered', cluster_entity=True)

# print results:
table = pd.DataFrame({'b': round(results_default.params, 4),
                      'se_default': round(results_default.std_errors, 4),
                      'se_cluster': round(results_cluster.std_errors, 4),
                      'se_css': round(results_css.std_errors, 4)})
print(f'table: \n{table}\n')
```

Output of Script 14.8: `Example-13-9-ClSE.py`

```
table:
               b  se_default  se_cluster  se_css
year      0.0077      0.0171      0.0136  0.0137
d83      -0.0999      0.0239      0.0219  0.0222
d84      -0.1478      0.0413      0.0356  0.0359
d85      -0.1524      0.0584      0.0505  0.0511
d86      -0.1249      0.0760      0.0624  0.0630
d87      -0.0841      0.0940      0.0773  0.0781
lprbarr  -0.3275      0.0300      0.0556  0.0562
lprbconv -0.2381      0.0182      0.0390  0.0394
lprbpris -0.1650      0.0260      0.0451  0.0456
lavgsen  -0.0218      0.0221      0.0254  0.0257
lpolpc    0.3984      0.0269      0.1014  0.1025
```

15. Instrumental Variables Estimation and Two Stage Least Squares

Instrumental variables are potentially powerful tools for the identification and estimation of causal effects. We start the discussion in Section 15.1 with the simplest case of one endogenous regressor and one instrumental variable. Section 15.2 shows how to implement models with additional exogenous regressors. In Section 15.3, we will introduce two stage least squares which efficiently deals with several endogenous variables and several instruments.

Tests of the exogeneity of the regressors and instruments are presented in Sections 15.4 and 15.5, respectively. Finally, Section 15.6 shows how to conveniently combine panel data estimators with instrumental variables.

15.1. Instrumental Variables in Simple Regression Models

We start the discussion of instrumental variables (IV) regression with the most straightforward case of only one regressor and only one instrumental variable. Consider the simple linear regression model for cross-sectional data

$$y = \beta_0 + \beta_1 x + u. \tag{15.1}$$

The OLS estimator for the slope parameter is $\hat{\beta}_1^{\text{OLS}} = \frac{\text{Cov}(x,y)}{\text{Var}(x)}$, see Equation 2.3. Suppose the regressor x is correlated with the error term u, so OLS parameter estimators will be biased and inconsistent.

If we have a valid instrumental variable z, we can consistently estimate β_1 using the IV estimator

$$\hat{\beta}_1^{\text{IV}} = \frac{\text{Cov}(z,y)}{\text{Cov}(z,x)}. \tag{15.2}$$

A valid instrument is correlated with the regressor x ("relevant"), so the denominator of Equation 15.2 is nonzero. It is also uncorrelated with the error term u ("exogenous"). Wooldridge (2019, Section 15.1) provides more discussion and examples.

To implement IV regression in *Python*, the module **linearmodels** offers the command **IV2SLS** including the convenient formula syntax we know from **statsmodels**. When working with IV regression in **linearmodels**, our first line of code always is:

```
import linearmodels.iv as iv
```

In the formula specification, the endogenous regressor(s) **x_end** and instruments **z** are provided in the following way:

```
y ~  1 + [ x_end ~ z ]
```

Note that we can easily consider different assumptions about the error term by providing the argument **cov_type** to the **fit** method. If you use **cov_type='unadjusted'** error terms are assumed to be homoskedastic. In combination with **debiased=True** this is the right option if you want to replicate results in Wooldridge (2019). The argument **cov_type='robust'** is the default and implements a robust estimation. Also remember that constants in **linearmodels** must be explicitly included by adding "**1**" to the formula. For other options, see the module documentation.

Wooldridge, Example 15.1: Return to Education for Married Women

Script 15.1 (Example-15-1.py) uses data from MROZ. We only analyze women with non-missing wage, so we use the method **dropna** to extract them. We want to estimate the return to education (**educ**) for these women. As an instrumental variable for education, we use the education of her father (**fatheduc**).

First, we calculate the OLS and IV slope parameters according to Equations 2.3 and 15.2. Then, the full OLS and IV estimates are calculated using the boxed routines **ols** and **IV2SLS**, respectively. Not surprisingly, the slope parameters match the manual results.

Script 15.1: Example-15-1.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

cov_yz = np.cov(mroz['lwage'], mroz['fatheduc'])[1, 0]
cov_xy = np.cov(mroz['educ'], mroz['lwage'])[1, 0]
cov_xz = np.cov(mroz['educ'], mroz['fatheduc'])[1, 0]
var_x = np.var(mroz['educ'], ddof=1)
x_bar = np.mean(mroz['educ'])
y_bar = np.mean(mroz['lwage'])

# OLS slope parameter manually:
b_ols_man = cov_xy / var_x
print(f'b_ols_man: {b_ols_man}\n')

# IV slope parameter manually:
b_iv_man = cov_yz / cov_xz
print(f'b_iv_man: {b_iv_man}\n')

# OLS automatically:
reg_ols = smf.ols(formula='np.log(wage) ~ educ', data=mroz)
results_ols = reg_ols.fit()

# print regression table:
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# IV automatically:
reg_iv = iv.IV2SLS.from_formula(formula='np.log(wage) ~ 1 + [educ ~ fatheduc]',
                                data=mroz)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)
```

```
# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Output of Script 15.1: Example-15-1.py

```
b_ols_man: 0.10864865517467534

b_iv_man: 0.05917347999936601

table_ols:
                b      se       t   pval
Intercept -0.1852  0.1852 -0.9998  0.318
educ       0.1086  0.0144  7.5451  0.000

table_iv:
                b      se       t    pval
Intercept  0.4411  0.4461  0.9888  0.3233
educ       0.0592  0.0351  1.6839  0.0929
```

15.2. More Exogenous Regressors

The IV approach can easily be generalized to include additional exogenous regressors, i.e. regressors that are assumed to be unrelated to the error term. In the formula specification of **IV2SLS**, the exogenous regressor(s) **x_exg**, the endogenous regressor(s) **x_end** and instruments **z** are provided in the following way:

```
y ~  1 + x_exg + [ x_end ~ z ]
```

Wooldridge, Example 15.4: Using College Proximity as an IV for Education

In Script 15.2 (`Example-15-4.py`), we use CARD to estimate the return to education. Education is allowed to be endogenous and instrumented with the dummy variable **nearc4** which indicates whether the individual grew up close to a college. In addition, we control for experience, race, and regional information. These variables are assumed to be exogenous and act as their own instruments.
We first check for relevance by regressing the endogenous independent variable **educ** on all exogenous variables including the instrument **nearc4**. Its parameter is highly significantly different from zero, so relevance is supported. We then estimate the log wage equation with OLS and IV.

Script 15.2: **Example-15-4.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf

card = woo.dataWoo('card')

# checking for relevance with reduced form:
reg_redf = smf.ols(
    formula='educ ~ nearc4 + exper + I(exper**2) + black + smsa +'
    'south + smsa66 + reg662 + reg663 + reg664 + reg665 + reg666 +'
    'reg667 + reg668 + reg669', data=card)
results_redf = reg_redf.fit()

# print regression table:
table_redf = pd.DataFrame({'b': round(results_redf.params, 4),
                           'se': round(results_redf.bse, 4),
                           't': round(results_redf.tvalues, 4),
                           'pval': round(results_redf.pvalues, 4)})
print(f'table_redf: \n{table_redf}\n')

# OLS:
reg_ols = smf.ols(
    formula='np.log(wage) ~ educ + exper + I(exper**2) + black + smsa +'
    'south + smsa66 + reg662 + reg663 + reg664 + reg665 +'
    'reg666 + reg667 + reg668 + reg669', data=card)
results_ols = reg_ols.fit()

# print regression table:
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# IV automatically:
reg_iv = iv.IV2SLS.from_formula(
    formula='np.log(wage)~ 1 + exper + I(exper**2) + black + smsa + '
            'south + smsa66 + reg662 + reg663 + reg664 + reg665 +'
            'reg666 + reg667 + reg668 + reg669 + [educ ~ nearc4]',
    data=card)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Output of Script 15.2: `Example-15-4.py`

```
table_redf:
                    b      se         t    pval
Intercept     16.6383  0.2406   69.1446  0.0000
nearc4         0.3199  0.0879    3.6408  0.0003
exper         -0.4125  0.0337  -12.2415  0.0000
I(exper ** 2)   0.0009  0.0017    0.5263  0.5987
black         -0.9355  0.0937   -9.9806  0.0000
smsa           0.4022  0.1048    3.8372  0.0001
south         -0.0516  0.1354   -0.3811  0.7032
smsa66         0.0255  0.1058    0.2409  0.8096
reg662        -0.0786  0.1871   -0.4203  0.6743
reg663        -0.0279  0.1834   -0.1524  0.8789
reg664         0.1172  0.2173    0.5394  0.5897
reg665        -0.2726  0.2184   -1.2481  0.2121
reg666        -0.3028  0.2371   -1.2773  0.2016
reg667        -0.2168  0.2344   -0.9250  0.3550
reg668         0.5239  0.2675    1.9587  0.0502
reg669         0.2103  0.2025    1.0386  0.2991

table_ols:
                   b      se         t    pval
Intercept     4.6208  0.0742   62.2476  0.0000
educ          0.0747  0.0035   21.3510  0.0000
exper         0.0848  0.0066   12.8063  0.0000
I(exper ** 2) -0.0023  0.0003   -7.2232  0.0000
black        -0.1990  0.0182  -10.9058  0.0000
smsa          0.1364  0.0201    6.7851  0.0000
south        -0.1480  0.0260   -5.6950  0.0000
smsa66        0.0262  0.0194    1.3493  0.1773
reg662        0.0964  0.0359    2.6845  0.0073
reg663        0.1445  0.0351    4.1151  0.0000
reg664        0.0551  0.0417    1.3221  0.1862
reg665        0.1280  0.0418    3.0599  0.0022
reg666        0.1405  0.0452    3.1056  0.0019
reg667        0.1180  0.0448    2.6334  0.0085
reg668       -0.0564  0.0513   -1.1010  0.2710
reg669        0.1186  0.0388    3.0536  0.0023

table_iv:
                   b      se        t    pval
Intercept     3.6662  0.9248   3.9641  0.0001
exper         0.1083  0.0237   4.5764  0.0000
I(exper ** 2) -0.0023  0.0003  -7.0014  0.0000
black        -0.1468  0.0539  -2.7231  0.0065
smsa          0.1118  0.0317   3.5313  0.0004
south        -0.1447  0.0273  -5.3023  0.0000
smsa66        0.0185  0.0216   0.8576  0.3912
reg662        0.1008  0.0377   2.6739  0.0075
reg663        0.1483  0.0368   4.0272  0.0001
reg664        0.0499  0.0437   1.1408  0.2541
reg665        0.1463  0.0471   3.1079  0.0019
reg666        0.1629  0.0519   3.1382  0.0017
reg667        0.1346  0.0494   2.7240  0.0065
reg668       -0.0831  0.0593  -1.4002  0.1616
reg669        0.1078  0.0418   2.5784  0.0100
educ          0.1315  0.0550   2.3926  0.0168
```

15.3. Two Stage Least Squares

Two stage least squares (2SLS) is a general approach for IV estimation when we have one or more endogenous regressors and at least as many additional instrumental variables. Consider the regression model

$$y_1 = \beta_0 + \beta_1 y_2 + \beta_2 y_3 + \beta_3 z_1 + \beta_4 z_2 + \beta_5 z_3 + u_1. \tag{15.3}$$

The regressors y_2 and y_3 are potentially correlated with the error term u_1, the regressors z_1, z_2, and z_3 are assumed to be exogenous. Because we have two endogenous regressors, we need at least two additional instrumental variables, say z_4 and z_5.

The name of 2SLS comes from the fact that it can be performed in two stages of OLS regressions:

(1) Separately regress y_2 and y_3 on z_1 through z_5. Obtain fitted values $\hat{y}_2$ and $\hat{y}_3$.

(2) Regress y_1 on $\hat{y}_2$, $\hat{y}_3$, and z_1 through z_3.

If the instruments are valid, this will give consistent estimates of the parameters β_0 through β_5. Generalizing this to more endogenous regressors and instrumental variables is obvious.

This procedure can of course easily be implemented using **ols** in **statsmodels**, remembering that fitted values are saved in **fittedvalues**. One of the problems of this manual approach is that the resulting variance-covariance matrix and analyses based on them are invalid. Conveniently, **IV2SLS** will automatically do these calculations and calculate correct standard errors and the like.

Wooldridge, Example 15.5: Return to Education for Married Women

We continue Example 15.1 and still want to estimate the return to education for women using the data in `MROZ`. Now, we use both mother's and father's education as instruments for their own education.
In Script 15.3 (`Example-15-5.py`), we obtain 2SLS estimates in two ways: First, we do both stages manually, including fitted education as **educ_fitted** as a regressor in the second stage. **IV2SLS** does this automatically and delivers the same parameter estimates as the output table reveals. But the standard errors differ slightly because the manual two stage version did not correct them.

Script 15.3: Example-15-5.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 1st stage (reduced form):
reg_redf = smf.ols(formula='educ ~ exper + I(exper**2) + motheduc + fatheduc',
                   data=mroz)
results_redf = reg_redf.fit()
mroz['educ_fitted'] = results_redf.fittedvalues

# print regression table:
table_redf = pd.DataFrame({'b': round(results_redf.params, 4),
                           'se': round(results_redf.bse, 4),
                           't': round(results_redf.tvalues, 4),
                           'pval': round(results_redf.pvalues, 4)})
print(f'table_redf: \n{table_redf}\n')
```

```
# 2nd stage:
reg_secstg = smf.ols(formula='np.log(wage) ~ educ_fitted + exper + I(exper**2)',
                     data=mroz)
results_secstg = reg_secstg.fit()

# print regression table:
table_secstg = pd.DataFrame({'b': round(results_secstg.params, 4),
                             'se': round(results_secstg.bse, 4),
                             't': round(results_secstg.tvalues, 4),
                             'pval': round(results_secstg.pvalues, 4)})
print(f'table_secstg: \n{table_secstg}\n')

# IV automatically:
reg_iv = iv.IV2SLS.from_formula(
    formula='np.log(wage) ~ 1 + exper + I(exper**2) +'
            '[educ  ~ motheduc + fatheduc]',
    data=mroz)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Output of Script 15.3: Example-15-5.py

```
table_redf:
                     b      se        t    pval
Intercept       9.1026  0.4266  21.3396  0.0000
exper           0.0452  0.0403   1.1236  0.2618
I(exper ** 2) -0.0010  0.0012  -0.8386  0.4022
motheduc        0.1576  0.0359   4.3906  0.0000
fatheduc        0.1895  0.0338   5.6152  0.0000

table_secstg:
                     b      se       t    pval
Intercept       0.0481  0.4198  0.1146  0.9088
educ_fitted     0.0614  0.0330  1.8626  0.0632
exper           0.0442  0.0141  3.1361  0.0018
I(exper ** 2) -0.0009  0.0004 -2.1344  0.0334

table_iv:
                     b      se       t    pval
Intercept       0.0481  0.4003  0.1202  0.9044
exper           0.0442  0.0134  3.2883  0.0011
I(exper ** 2) -0.0009  0.0004 -2.2380  0.0257
educ            0.0614  0.0314  1.9530  0.0515
```

15.4. Testing for Exogeneity of the Regressors

There is another way to get the same IV parameter estimates as with 2SLS. In the same setup as above, this "control function approach" also consists of two stages:

(1) Like in 2SLS, regress y_2 and y_3 on z_1 through z_5. Obtain residuals $\hat{v}_2$ and $\hat{v}_3$ instead of fitted values $\hat{y}_2$ and $\hat{y}_3$.

(2) Regress y_1 on y_2, y_3, z_1, z_2, z_3, and the first stage residuals $\hat{v}_2$ and $\hat{v}_3$.

This approach is as simple to implement as 2SLS and will also result in the same parameter estimates and invalid OLS standard errors in the second stage (unless the dubious regressors y_2 and y_3 are in fact exogenous).

After this second stage regression, we can test for exogeneity in a simple way assuming the instruments are valid. We just need to do a t or F test of the null hypothesis that the parameters of the first-stage residuals are equal to zero. If we reject this hypothesis, this indicates endogeneity of y_2 and y_3.

Wooldridge, Example 15.7: Return to Education for Married Women

In Script 15.4 (`Example-15-7.py`), we continue Example 15.5 using the control function approach. Again, we use both mother's and father's education as instruments. The first stage regression is identical as in Script 15.3 (`Example-15-5.py`). The second stage adds the first stage residuals to the original list of regressors. The parameter estimates are identical to both the manual 2SLS and the automatic **`IV2SLS`** results. We can perform a t test based on the regression table as a test for exogeneity. Here, $t = \frac{0.058}{0.035} \approx 1.67$ with a two-sided p value of $p = 0.095$, indicating a marginally significant evidence for endogeneity.

Script 15.4: **`Example-15-7.py`**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 1st stage (reduced form):
reg_redf = smf.ols(formula='educ ~ exper + I(exper**2) + motheduc + fatheduc',
                   data=mroz)
results_redf = reg_redf.fit()
mroz['resid'] = results_redf.resid

# 2nd stage:
reg_secstg = smf.ols(formula='np.log(wage)~ resid + educ + exper + I(exper**2)',
                     data=mroz)
results_secstg = reg_secstg.fit()

# print regression table:
table_secstg = pd.DataFrame({'b': round(results_secstg.params, 4),
                             'se': round(results_secstg.bse, 4),
                             't': round(results_secstg.tvalues, 4),
                             'pval': round(results_secstg.pvalues, 4)})
print(f'table_secstg: \n{table_secstg}\n')
```

Output of Script 15.4: `Example-15-7.py`

```
table_secstg:
                     b      se       t    pval
Intercept       0.0481  0.3946  0.1219  0.9030
resid           0.0582  0.0348  1.6711  0.0954
educ            0.0614  0.0310  1.9815  0.0482
exper           0.0442  0.0132  3.3363  0.0009
I(exper ** 2) -0.0009  0.0004 -2.2706  0.0237
```

15.5. Testing Overidentifying Restrictions

If we have more instruments than endogenous variables, we can use either all or only some of them. If all are valid, using all improves the accuracy of the 2SLS estimator and reduces its standard errors. If the exogeneity of some is dubious, including them might cause inconsistency. It is therefore useful to test for the exogeneity of a set of dubious instruments if we have another (large enough) set that is undoubtedly exogenous. The procedure is described by Wooldridge (2019, Section 15.5):

(1) Estimate the model by 2SLS and obtain residuals $\hat{u}_1$.

(2) Regress $\hat{u}_1$ on all exogenous variables and calculate R_1^2.

(3) The test statistic nR_1^2 is asymptotically distributed as χ_q^2, where q is the number of *overidentifying* restrictions, i.e. number of instruments minus number of endogenous regressors.

Wooldridge, Example 15.8: Return to Education for Married Women

We will again use the data and model of Examples 15.5 and 15.7. Script 15.5 (`Example-15-8.py`) estimates the model using **`IV2SLS`**. The results are stored in variable **`results_iv`**. We then run the auxiliary regression and compute its R^2 as **`r2`**. The test statistic **`teststat`** is computed to be 0.378. We also compute the p value from the χ_1^2 distribution. We cannot reject exogeneity of the instruments using this test. But be aware of the fact that the underlying assumption that at least one instrument is valid might be violated here.

Script 15.5: **Example-15-8.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# IV regression:
reg_iv = iv.IV2SLS.from_formula(formula='np.log(wage) ~ 1 + exper + I(exper**2) +'
                                        '[educ ~ motheduc + fatheduc]', data=mroz)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')

# auxiliary regression:
mroz['resid_iv'] = results_iv.resids
reg_aux = smf.ols(formula='resid_iv ~ exper + I(exper**2) + motheduc + fatheduc',
                  data=mroz)
results_aux = reg_aux.fit()

# calculations for test:
r2 = results_aux.rsquared
n = results_aux.nobs
teststat = n * r2
pval = 1 - stats.chi2.cdf(teststat, 1)

print(f'r2: {r2}\n')
print(f'n: {n}\n')
print(f'teststat: {teststat}\n')
print(f'pval: {pval}\n')
```

Output of Script 15.5: **Example-15-8.py**

```
table_iv:
                     b      se       t    pval
Intercept       0.0481  0.4003  0.1202  0.9044
exper           0.0442  0.0134  3.2883  0.0011
I(exper ** 2) -0.0009  0.0004 -2.2380  0.0257
educ            0.0614  0.0314  1.9530  0.0515

r2: 0.0008833444088017783

n: 428.0

teststat: 0.3780714069671611

pval: 0.5386371981605377
```

15.6. Instrumental Variables with Panel Data

Instrumental variables can be used for panel data, too. In this way, we can get rid of time-constant individual heterogeneity by first differencing or within transformations and then fix remaining endogeneity problems with instrumental variables.

We know how to get panel data estimates using OLS on the transformed data, so we can easily use IV as before.

Wooldridge, Example 15.10: Job Training and Worker Productivity

We use the data set `JTRAIN` to estimate the effect of job training **hrsemp** on the scrap rate. In Script 15.6 (`Example-15-10.py`), we load the data, choose a subset of the years 1987 and 1988 with **loc** and store the data with correct index variables **fcode** and **year**, see Section 13.3. Then we estimate the parameters using first-differencing with the instrumental variable **grant**.

Script 15.6: Example-15-10.py

```
import wooldridge as woo
import pandas as pd
import linearmodels.iv as iv

jtrain = woo.dataWoo('jtrain')

# define panel data (for 1987 and 1988 only):
jtrain_87_88 = jtrain.loc[(jtrain['year'] == 1987) | (jtrain['year'] == 1988), :]
jtrain_87_88 = jtrain_87_88.set_index(['fcode', 'year'])

# manual computation of deviations of entity means:
jtrain_87_88['lscrap_diff1'] = \
    jtrain_87_88.sort_values(['fcode', 'year']).groupby('fcode')['lscrap'].diff()
jtrain_87_88['hrsemp_diff1'] = \
    jtrain_87_88.sort_values(['fcode', 'year']).groupby('fcode')['hrsemp'].diff()
jtrain_87_88['grant_diff1'] = \
    jtrain_87_88.sort_values(['fcode', 'year']).groupby('fcode')['grant'].diff()

# IV regression:
reg_iv = iv.IV2SLS.from_formula(
    formula='lscrap_diff1 ~ 1 + [hrsemp_diff1 ~ grant_diff1]',
    data=jtrain_87_88)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Output of Script 15.6: Example-15-10.py

```
table_iv:
                   b      se       t    pval
Intercept    -0.0327  0.1270 -0.2573  0.7982
hrsemp_diff1 -0.0142  0.0079 -1.7882  0.0808
```

16. Simultaneous Equations Models

In simultaneous equations models (SEM), both the dependent variable and at least one regressor are determined jointly. This leads to an endogeneity problem and inconsistent OLS parameter estimators. The main challenge for successfully using SEM is to specify a sensible model and make sure it is identified, see Wooldridge (2019, Sections 16.1–16.3). We briefly introduce a general model and the notation in Section 16.1.

As discussed in Chapter 15, 2SLS regression can solve endogeneity problems if there are enough exogenous instrumental variables. This also works in the setting of SEM, an example is given in Section 16.2. Using **`linearmodels`**, more advanced estimation commands are straightforward to implement. We will show this for three-stage-least-squares (3SLS) estimation in Section 16.3.

16.1. Setup and Notation

Consider the general SEM with q endogenous variables $y_1, \ldots, y_q$ and k exogenous variables $x_1, \ldots, x_k$. The system of equations is:

$$\begin{aligned}
y_1 &= \alpha_{12} y_2 + \alpha_{13} y_3 + \cdots + \alpha_{1q} y_q \qquad + \beta_{10} + \beta_{11} x_1 + \cdots + \beta_{1k} x_k + u_1 \\
y_2 &= \alpha_{21} y_1 + \alpha_{23} y_3 + \cdots + \alpha_{2q} y_q \qquad + \beta_{20} + \beta_{21} x_1 + \cdots + \beta_{2k} x_k + u_2 \\
&\vdots \\
y_q &= \alpha_{q1} y_1 + \alpha_{q2} y_2 + \cdots + \alpha_{qq-1} y_{q-1} + \beta_{q0} + \beta_{q1} x_1 + \cdots + \beta_{qk} x_k + u_q
\end{aligned}$$

As discussed in more detail in Wooldridge (2019, Section 16), this system is not identified without restrictions on the parameters. The order condition for identification of any equation is that if we have m included endogenous regressors (i.e. α parameters that are not restricted to 0), we need to exclude at least m exogenous regressors (i.e. restrict their β parameters to 0). They can then be used as instrumental variables.

Wooldridge, Example 16.3: Labor Supply of Married, Working Women

We have the two endogenous variables `hours` and `wage` which influence each other.

$$\begin{aligned}
\texttt{hours} &= \alpha_{12} \log(\texttt{wage}) + \beta_{10} + \beta_{11}\texttt{educ} + \beta_{12}\texttt{age} + \beta_{13}\texttt{kidslt6} + \beta_{14}\texttt{nwifeinc} \\
&\qquad + \beta_{15}\texttt{exper} + \beta_{16}\texttt{exper}^2 + u_1 \\
\log(\texttt{wage}) &= \alpha_{21}\texttt{hours} \qquad + \beta_{20} + \beta_{21}\texttt{educ} + \beta_{22}\texttt{age} + \beta_{23}\texttt{kidslt6} + \beta_{24}\texttt{nwifeinc} \\
&\qquad + \beta_{25}\texttt{exper} + \beta_{26}\texttt{exper}^2 + u_2
\end{aligned}$$

For both equations to be identified, we have to exclude at least one exogenous regressor from each equation. Wooldridge (2019) discusses a model in which we restrict $\beta_{15} = \beta_{16} = 0$ in the first and $\beta_{22} = \beta_{23} = \beta_{24} = 0$ in the second equation.

16.2. Estimation by 2SLS

Estimation of each equation separately by 2SLS is straightforward once we have set up the system and ensured identification. The excluded regressors in each equation serve as instrumental variables. As shown in Chapter 15, the command **IV2SLS** from the module **linearmodels** provides convenient 2SLS estimation.

Wooldridge, Example 16.5: Labor Supply of Married, Working Women

Script 16.1 (`Example-16-5-2SLS.py`) estimates the parameters of the two equations from Example 16.3 separately using **IV2SLS**.

Script 16.1: **Example-16-5-2SLS.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 2SLS regressions:
reg_iv1 = iv.IV2SLS.from_formula(
    'hours ~ 1 + educ +  age + kidslt6 + nwifeinc +'
    '[np.log(wage) ~ exper + I(exper**2)]', data=mroz)
results_iv1 = reg_iv1.fit(cov_type='unadjusted', debiased=True)

reg_iv2 = iv.IV2SLS.from_formula(
    'np.log(wage) ~ 1 + educ + exper + I(exper**2) +'
    '[hours ~ age + kidslt6 + nwifeinc]', data=mroz)
results_iv2 = reg_iv2.fit(cov_type='unadjusted', debiased=True)

# print results:
table_iv1 = pd.DataFrame({'b': round(results_iv1.params, 4),
                          'se': round(results_iv1.std_errors, 4),
                          't': round(results_iv1.tstats, 4),
                          'pval': round(results_iv1.pvalues, 4)})
print(f'table_iv1: \n{table_iv1}\n')

table_iv2 = pd.DataFrame({'b': round(results_iv2.params, 4),
                          'se': round(results_iv2.std_errors, 4),
                          't': round(results_iv2.tstats, 4),
                          'pval': round(results_iv2.pvalues, 4)})
print(f'table_iv2: \n{table_iv2}\n')

cor_u1u2 = np.corrcoef(results_iv1.resids, results_iv2.resids)[0, 1]
print(f'cor_u1u2: {cor_u1u2}\n')
```

Output of Script 16.1: Example-16-5-2SLS.py

```
table_iv1:
                       b        se       t    pval
Intercept      2225.6618  574.5641  3.8737  0.0001
educ           -183.7513   59.0998 -3.1092  0.0020
age              -7.8061    9.3780 -0.8324  0.4057
kidslt6        -198.1543  182.9291 -1.0832  0.2793
nwifeinc        -10.1696    6.6147 -1.5374  0.1249
np.log(wage)   1639.5556  470.5757  3.4841  0.0005

table_iv2:
                     b      se       t    pval
Intercept      -0.6557  0.3378 -1.9412  0.0529
educ            0.1103  0.0155  7.1069  0.0000
exper           0.0346  0.0195  1.7742  0.0767
I(exper ** 2)  -0.0007  0.0005 -1.5543  0.1209
hours           0.0001  0.0003  0.4945  0.6212

cor_u1u2: -0.903769419629963
```

16.3. Outlook: Estimation by 3SLS

An interesting piece of information in Script 16.1 (`Example-16-5-2SLS.py`) is the correlation between the residuals of the equations. In the example, it is reported to be a substantially negative -0.90. We can account for the correlation between the error terms to derive a potentially more efficient parameter estimator than 2SLS. Without going into details here, the three stage least squares (3SLS) estimator adds another stage to 2SLS by estimating the correlation and accounting for it using a FGLS approach. For a detailed discussion of this and related methods, see for example Wooldridge (2010, Chapter 8).

Using 3SLS in **linearmodels** is simple: The function **IV3SLS** is all we need as the output of Script 16.2 (`Example-16-5-3SLS.py`) shows.

Script 16.2: Example-16-5-3SLS.py

```
import wooldridge as woo
import numpy as np
import linearmodels.system as iv3

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 3SLS regressions:
formula = {'eq1': 'hours ~ 1 + educ + age + kidslt6 + nwifeinc +'
                  '[np.log(wage) ~ exper+I(exper**2)]',
           'eq2': 'np.log(wage) ~ 1 + educ + exper + I(exper**2) +'
                  '[hours ~ age + kidslt6 + nwifeinc]'}

reg_3sls = iv3.IV3SLS.from_formula(formula, data=mroz)

results_3sls = reg_3sls.fit(cov_type='unadjusted', debiased=True)
print(f'results_3sls: \n{results_3sls}\n')
```

Output of Script 16.2: Example-16-5-3SLS.py

```
results_3sls:
                           System GLS Estimation Summary
===================================================================================
Estimator:                        GLS   Overall R-squared:                  -2.3957
No. Equations.:                     2   McElroy's R-squared:                 0.7846
No. Observations:                 428   Judge's (OLS) R-squared:            -2.3957
Date:                Fri, May 08 2020   Berndt's R-squared:                  0.5181
Time:                        08:55:43   Dhrymes's R-squared:                -2.3957
                                        Cov. Estimator:                  unadjusted
                                        Num. Constraints:                      None
                    Equation: eq1, Dependent Variable: hours
================================================================================
              Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
--------------------------------------------------------------------------------
Intercept        2305.9     511.54     4.5077     0.0000      1300.4      3311.3
educ            -212.82     53.727    -3.9611     0.0001     -318.43     -107.21
age             -9.5150     7.9609    -1.1952     0.2327     -25.163      6.1331
kidslt6         -192.36     150.92    -1.2746     0.2032     -489.00      104.28
nwifeinc        -0.1770     3.5836    -0.0494     0.9606     -7.2210      6.8670
np.log(wage)     1781.9     439.88     4.0509     0.0001      917.30      2646.6

====================
    Instruments
--------------------
exper, I(exper ** 2)

                 Equation: eq2, Dependent Variable: np.log(wage)
================================================================================
               Parameter  Std. Err.     T-stat    P-value    Lower CI    Upper CI
--------------------------------------------------------------------------------
Intercept        -0.6939     0.3360    -2.0653     0.0395     -1.3543     -0.0335
educ              0.1127     0.0154     7.3355     0.0000      0.0825      0.1429
exper             0.0214     0.0154     1.3929     0.1644     -0.0088      0.0517
I(exper ** 2)    -0.0003     0.0003    -1.1303     0.2590     -0.0008      0.0002
hours             0.0002     0.0002     0.7707     0.4413     -0.0003      0.0007
======================
     Instruments
----------------------
age, kidslt6, nwifeinc
----------------------

Covariance Estimator:
Homoskedastic (Unadjusted) Covariance (Debiased: True, GLS: True)
```

17. Limited Dependent Variable Models and Sample Selection Corrections

A limited dependent variable (LDV) can only take a limited set of values. An extreme case are binary variables that can only take two values. We already used such dummy variables as regressors in Chapter 7. Section 17.1 discusses how to use them as dependent variables. Another example for LDV are counts that take only non-negative integers, they are covered in Section 17.2. Similarly, Tobit models discussed in Section 17.3 deal with dependent variables that can only take positive values (or are restricted in a similar way), but are otherwise continuous.

The Sections 17.4 and 17.5 are concerned with continuous dependent variables but are not perfectly observed. For some units of the censored, truncated, or selected observations we only know that they are above or below a certain threshold or we don't know anything about them.

17.1. Binary Responses

Binary dependent variables are frequently studied in applied econometrics. Because a dummy variable y can only take the values 0 and 1, its (conditional) expected value is equal to the (conditional) probability that $y = 1$:

$$\begin{aligned} \mathrm{E}(y|\mathbf{x}) &= 0 \cdot \mathrm{P}(y=0|\mathbf{x}) + 1 \cdot \mathrm{P}(y=1|\mathbf{x}) \\ &= \mathrm{P}(y=1|\mathbf{x}) \end{aligned} \tag{17.1}$$

So when we study the conditional mean, it makes sense to think about it as the probability of outcome $y = 1$. Likewise, the predicted value $\hat{y}$ should be thought of as a predicted probability.

17.1.1. Linear Probability Models

If a dummy variable is used as the dependent variable y, we can still use OLS to estimate its relation to the regressors $\mathbf{x}$. These linear probability models are covered by Wooldridge (2019) in Section 7.5. If we write the usual linear regression model

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k \tag{17.2}$$

and make the usual assumptions, especially MLR.4: $E(u|\mathbf{x}) = 0$, this implies for the conditional mean (which is the probability that $y = 1$) and the predicted probabilities:

$$\mathrm{P}(y=1|\mathbf{x}) = \mathrm{E}(y|\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k \tag{17.3}$$

$$\hat{\mathrm{P}}(y=1|\mathbf{x}) = \quad \hat{y} \quad = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k \tag{17.4}$$

The interpretation of the parameters is straightforward: β_j is a measure of the average change in probability of a "success" ($y = 1$) as x_j increases by one unit and the other determinants remain constant. Linear probability models automatically suffer from heteroscedasticity, so with OLS, we should use heteroscedasticity-robust inference, see Section 8.1.

Wooldridge, Example 17.1: Married Women's Labor Force Participation

We study the probability that a woman is in the labor force depending on socio-demographic characteristics. Script 17.1 (`Example-17-1-1.py`) estimates a linear probability model using the data set `mroz`. The estimated coefficient of **`educ`** can be interpreted as: an additional year of schooling increases the probability that a woman is in the labor force *ceteris paribus* by 0.038 on average. We used the refined version of White's robust variance-covariance matrix.

Script 17.1: `Example-17-1-1.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate linear probability model:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper +'
                          'I(exper**2) + age + kidslt6 + kidsge6',
                  data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

# print regression table:
table = pd.DataFrame({'b': round(results_lin.params, 4),
                      'se': round(results_lin.bse, 4),
                      't': round(results_lin.tvalues, 4),
                      'pval': round(results_lin.pvalues, 4)})
print(f'table: \n{table}\n')
```

Output of Script 17.1: `Example-17-1-1.py`

```
table: 
                     b      se       t    pval
Intercept       0.5855  0.1536  3.8125  0.0001
nwifeinc       -0.0034  0.0016 -2.1852  0.0289
educ            0.0380  0.0073  5.1766  0.0000
exper           0.0395  0.0060  6.6001  0.0000
I(exper ** 2)  -0.0006  0.0002 -2.9973  0.0027
age            -0.0161  0.0024 -6.6640  0.0000
kidslt6        -0.2618  0.0322 -8.1430  0.0000
kidsge6         0.0130  0.0137  0.9526  0.3408
```

One problem with linear probability models is that $\mathrm{P}(y = 1|\mathbf{x})$ is specified as a linear function of the regressors. By construction, there are (more or less realistic) combinations of regressor values that yield $\hat{y} < 0$ or $\hat{y} > 1$. Since these are probabilities, this does not really make sense.

As an example, Script 17.2 (`Example-17-1-2.py`) calculates the predicted values for two women (see Section 6.2 for how to **`predict`** after OLS estimation): Woman 1 is 20 years old, has no work experience, 5 years of education, two children below age 6 and has additional family income of 100,000 USD. Woman 2 is 52 years old, has 30 years of work experience, 17 years of education, no children and no other source of income. The predicted "probability" for woman 1 is -41%, the probability for woman 2 is 104% as can also be easily checked with a calculator.

Script 17.2: `Example-17-1-2.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate linear probability model:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper +'
                          'I(exper**2) + age + kidslt6 + kidsge6',
                  data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

# predictions for two "extreme" women:
X_new = pd.DataFrame(
    {'nwifeinc': [100, 0], 'educ': [5, 17],
     'exper': [0, 30], 'age': [20, 52],
     'kidslt6': [2, 0], 'kidsge6': [0, 0]})
predictions = results_lin.predict(X_new)

print(f'predictions: \n{predictions}\n')
```

Output of Script 17.2: `Example-17-1-2.py`

```
predictions:
0   -0.410458
1    1.042808
dtype: float64
```

17.1.2. Logit and Probit Models: Estimation

Specialized models for binary responses make sure that the implied probabilities are restricted between 0 and 1. An important class of models specifies the success probability as

$$\mathrm{P}(y=1|\mathbf{x}) = G(\beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k) = G(\mathbf{x}\boldsymbol{\beta}) \tag{17.5}$$

where the "link function" $G(z)$ always returns values between 0 and 1. In the statistics literature, this type of models is often called generalized linear model (GLM) because a linear part $\mathbf{x}\boldsymbol{\beta}$ shows up within the nonlinear function G.

For binary response models, by far the most widely used specifications for G are

- the **probit** model with $G(z) = \Phi(z)$, the standard normal CDF and
- the **logit** model with $G(z) = \Lambda(z) = \frac{\exp(z)}{1+\exp(z)}$, the CDF of the logistic distribution.

Wooldridge (2019, Section 17.1) provides useful discussions of the derivation and interpretation of these models. Here, we are concerned with the practical implementation. In **`statsmodels`**, many generalized linear models can be estimated with already implemented routines working similar as **`ols`**. In the following, we will use two of them frequently:

- **`logit`** for the logit model and
- **`probit`** for the probit model.

Maximum likelihood estimation (MLE) of the parameters is done automatically and the **`summary`** of the results contains the regression table and additional information. Scripts 17.3 (`Example-17-1-3.py`) and 17.4 (`Example-17-1-4.py`) implement the logit and probit model, respectively. The log likelihood value $\mathscr{L}(\hat{\boldsymbol{\beta}})$ is saved as the attribute **`llf`** and is also reported by

summary. The command also reports **LL-Null**, which is the log likelihood $\mathscr{L}_0$ of a model with an intercept only.

Scripts 17.3 (`Example-17-1-3.py`) and 17.4 (`Example-17-1-4.py`) demonstrate how to access the log likelihood and McFadden's pseudo R-squared that can be calculated as

$$\text{pseudo } R^2 = 1 - \frac{\mathscr{L}(\hat{\beta})}{\mathscr{L}_0}. \tag{17.6}$$

Script 17.3: Example-17-1-3.py

```
import wooldridge as woo
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate logit model:
reg_logit = smf.logit(formula='inlf ~ nwifeinc + educ + exper +'
                              'I(exper**2) + age + kidslt6 + kidsge6',
                      data=mroz)

# disp = 0 avoids printing out information during the estimation:
results_logit = reg_logit.fit(disp=0)
print(f'results_logit.summary(): \n{results_logit.summary()}\n')

# log likelihood value:
print(f'results_logit.llf: {results_logit.llf}\n')

# McFadden's pseudo R2:
print(f'results_logit.prsquared: {results_logit.prsquared}\n')
```

Output of Script 17.3: Example-17-1-3.py

```
results_logit.summary():
                           Logit Regression Results
==============================================================================
Dep. Variable:                   inlf   No. Observations:                  753
Model:                          Logit   Df Residuals:                      745
Method:                           MLE   Df Model:                            7
Date:                Thu, 14 May 2020   Pseudo R-squ.:                  0.2197
Time:                        12:36:00   Log-Likelihood:                -401.77
converged:                       True   LL-Null:                       -514.87
Covariance Type:            nonrobust   LLR p-value:                 3.159e-45
=================================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------
Intercept         0.4255      0.860      0.494      0.621      -1.261       2.112
nwifeinc         -0.0213      0.008     -2.535      0.011      -0.038      -0.005
educ              0.2212      0.043      5.091      0.000       0.136       0.306
exper             0.2059      0.032      6.422      0.000       0.143       0.269
I(exper ** 2)    -0.0032      0.001     -3.104      0.002      -0.005      -0.001
age              -0.0880      0.015     -6.040      0.000      -0.117      -0.059
kidslt6          -1.4434      0.204     -7.090      0.000      -1.842      -1.044
kidsge6           0.0601      0.075      0.804      0.422      -0.086       0.207
=================================================================================

results_logit.llf: -401.76515113438177

results_logit.prsquared: 0.21968137484058803
```

Script 17.4: Example-17-1-4.py

```
import wooldridge as woo
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate probit model:
reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper +'
                                'I(exper**2) + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)
print(f'results_probit.summary(): \n{results_probit.summary()}\n')

# log likelihood value:
print(f'results_probit.llf: {results_probit.llf}\n')

# McFadden's pseudo R2:
print(f'results_probit.prsquared: {results_probit.prsquared}\n')
```

Output of Script 17.4: Example-17-1-4.py

```
results_probit.summary():
                          Probit Regression Results
==============================================================================
Dep. Variable:                   inlf   No. Observations:                  753
Model:                         Probit   Df Residuals:                      745
Method:                           MLE   Df Model:                            7
Date:                Thu, 14 May 2020   Pseudo R-squ.:                  0.2206
Time:                        12:36:01   Log-Likelihood:                -401.30
converged:                       True   LL-Null:                       -514.87
Covariance Type:            nonrobust   LLR p-value:                 2.009e-45
=================================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------
Intercept         0.2701      0.509      0.531      0.595      -0.727       1.267
nwifeinc         -0.0120      0.005     -2.484      0.013      -0.022      -0.003
educ              0.1309      0.025      5.183      0.000       0.081       0.180
exper             0.1233      0.019      6.590      0.000       0.087       0.160
I(exper ** 2)    -0.0019      0.001     -3.145      0.002      -0.003      -0.001
age              -0.0529      0.008     -6.235      0.000      -0.069      -0.036
kidslt6          -0.8683      0.119     -7.326      0.000      -1.101      -0.636
kidsge6           0.0360      0.043      0.828      0.408      -0.049       0.121
=================================================================================

results_probit.llf: -401.30219317389515

results_probit.prsquared: 0.22058054372529368
```

17.1.3. Inference

The **summary** output of the **logit** or **probit** results contains a standard regression table with parameters and (asymptotic) standard errors. The next column is labeled **z** instead of **t** in the output of **ols**. The interpretation is the same. The difference is that the standard errors only have an asymptotic foundation and the distribution used for calculating p values is the standard normal distribution (which is equal to the t distribution with very large degrees of freedom). The bottom line is that tests for single parameters can be done as before, see Section 4.1.

For testing multiple hypotheses similar to the F test (see Section 4.3), the likelihood ratio test is popular. It is based on comparing the log likelihood values of the unrestricted and the restricted model. The test statistic is

$$LR = 2(\mathscr{L}_{ur} - \mathscr{L}_r) \tag{17.7}$$

where $\mathscr{L}_{ur}$ and $\mathscr{L}_r$ are the log likelihood values of the unrestricted and restricted model, respectively. Under H_0, the LR test statistic is asymptotically distributed as χ^2 with the degrees of freedom equal to the number of restrictions to be tested. The test of overall significance is a special case just like with F tests. The null hypothesis is that all parameters except the constant are equal to zero. With the notation above, the test statistic is

$$LR = 2\big(\mathscr{L}(\hat{\boldsymbol{\beta}}) - \mathscr{L}_0\big). \tag{17.8}$$

Translated to **statsmodels** with fitted model results stored in **results**, this corresponds to:

```
LR = 2 * (results.llf - results.llnull)
```

For other hypotheses, you can compute LR based on the log likelihood of a restricted model. Alternatively, **statsmodels** offers a Wald test with the function **wald_test** including the convenient calculation of p values. Script 17.5 (`Example-17-1-5.py`) implements the test of overall significance for the probit model using both manual and and automatic calculations. It also tests the joint null hypothesis that experience and age are irrelevant by first estimating the restricted model and then running the automated test.

Script 17.5: `Example-17-1-5.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# estimate probit model:
reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper +'
                                'I(exper**2) + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)

# test of overall significance (test statistic and pvalue):
llr1_manual = 2 * (results_probit.llf - results_probit.llnull)
print(f'llr1_manual: {llr1_manual}\n')
print(f'results_probit.llr: {results_probit.llr}\n')
print(f'results_probit.llr_pvalue: {results_probit.llr_pvalue}\n')
```

```
# automatic Wald test of H0 (experience and age are irrelevant):
hypotheses = ['exper=0', 'I(exper ** 2)=0', 'age=0']
waldstat = results_probit.wald_test(hypotheses)
teststat2_autom = waldstat.statistic
pval2_autom = waldstat.pvalue
print(f'teststat2_autom: {teststat2_autom}\n')
print(f'pval2_autom: {pval2_autom}\n')

# manual likelihood ratio statistic test
# of H0 (experience and age are irrelevant):
reg_probit_restr = smf.probit(formula='inlf ~ nwifeinc + educ +'
                                      'kidslt6 + kidsge6',
                              data=mroz)
results_probit_restr = reg_probit_restr.fit(disp=0)

llr2_manual = 2 * (results_probit.llf - results_probit_restr.llf)
pval2_manual = 1 - stats.chi2.cdf(llr2_manual, 3)
print(f'llr2_manual2: {llr2_manual}\n')
print(f'pval2_manual2: {pval2_manual}\n')
```

Output of Script 17.5: `Example-17-1-5.py`

```
llr1_manual: 227.14202283719214

results_probit.llr: 227.14202283719214

results_probit.llr_pvalue: 2.0086732957629427e-45

teststat2_autom: [[110.91852003]]

pval2_autom: 6.96073840669924e-24

llr2_manual2: 127.03401014418023

pval2_manual2: 0.0
```

17.1.4. Predictions

The command **`predict`** can calculate predicted values for the estimation sample ("fitted values") or arbitrary sets of regressor values also for binary response models estimated with **`logit`** or **`probit`**. Given the results of the **`fit`** method are stored in the variable **`results`**, we can calculate:

- $\mathbf{x}_i\hat{\boldsymbol{\beta}}$ for the estimation sample same as **`results.fittedvalues`**
- $\hat{y} = G(\mathbf{x}_i\hat{\boldsymbol{\beta}})$ for the estimation sample with **`results.predict()`**
- $\hat{y} = G(\mathbf{x}_i\hat{\boldsymbol{\beta}})$ for the regressor values stored in **`xpred`** with **`results.predict(xpred)`**

The predictions for the two hypothetical women introduced in Section 17.1.1 are repeated for the linear probability, logit, and probit models in Script 17.6 (`Example-17-1-6.py`). Unlike the linear probability model, the predicted probabilities from the logit and probit models remain between 0 and 1.

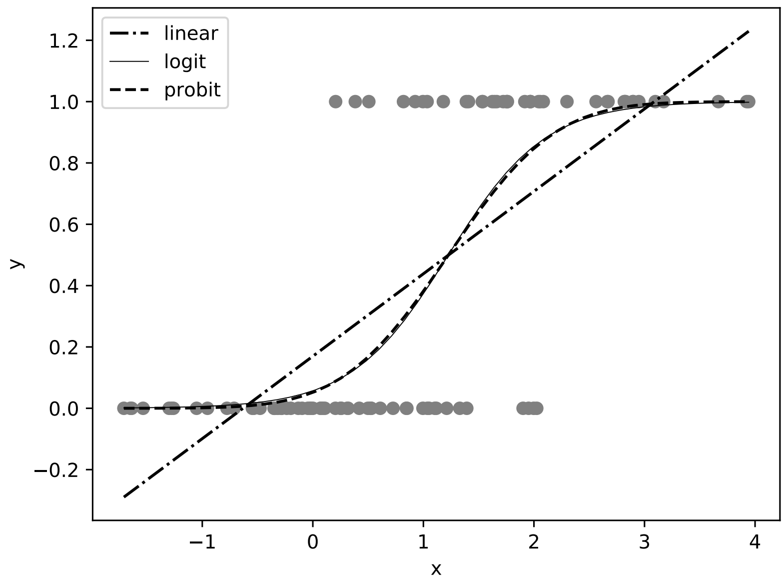
Script 17.6: `Example-17-1-6.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate models:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper +'
                          'I(exper**2) + age + kidslt6 + kidsge6',
                  data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

reg_logit = smf.logit(formula='inlf ~ nwifeinc + educ + exper +'
                              'I(exper**2) + age + kidslt6 + kidsge6',
                      data=mroz)
results_logit = reg_logit.fit(disp=0)

reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper +'
                                'I(exper**2) + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)

# predictions for two "extreme" women:
X_new = pd.DataFrame(
    {'nwifeinc': [100, 0], 'educ': [5, 17],
     'exper': [0, 30], 'age': [20, 52],
     'kidslt6': [2, 0], 'kidsge6': [0, 0]})
predictions_lin = results_lin.predict(X_new)
predictions_logit = results_logit.predict(X_new)
predictions_probit = results_probit.predict(X_new)

print(f'predictions_lin: \n{predictions_lin}\n')
print(f'predictions_logit: \n{predictions_logit}\n')
print(f'predictions_probit: \n{predictions_probit}\n')
```

Output of Script 17.6: `Example-17-1-6.py`

```
predictions_lin:
0   -0.410458
1    1.042808
dtype: float64

predictions_logit:
0    0.005218
1    0.950049
dtype: float64

predictions_probit:
0    0.001065
1    0.959870
dtype: float64
```

Figure 17.1. Predictions from Binary Response Models (Simulated Data)



If we only have one regressor, predicted values can nicely be plotted against it. Figure 17.1 shows such a figure for a simulated data set. For interested readers, the script used for generating the data and the figure is printed as Script 17.7 (`Binary-Predictions.py`) in Appendix IV (p. 399). In this example, the linear probability model clearly predicts probabilities outside of the "legal" area between 0 and 1. The logit and probit models yield almost identical predictions. This is a general finding that holds for most data sets.

17.1.5. Partial Effects

The parameters of linear regression models have straightforward interpretations: β_j measures the *ceteris paribus* effect of x_j on $\mathrm{E}(y|\mathbf{x})$. The parameters of nonlinear models like logit and probit have a less straightforward interpretation since the linear index $\mathbf{x}\boldsymbol{\beta}$ affects $\hat{y}$ through the link function G.

A useful measure of the influence is the partial effect (or marginal effect) which in a graph like Figure 17.1 is the slope and has the same interpretation as the parameters in the linear model. Because of the chain rule, it is

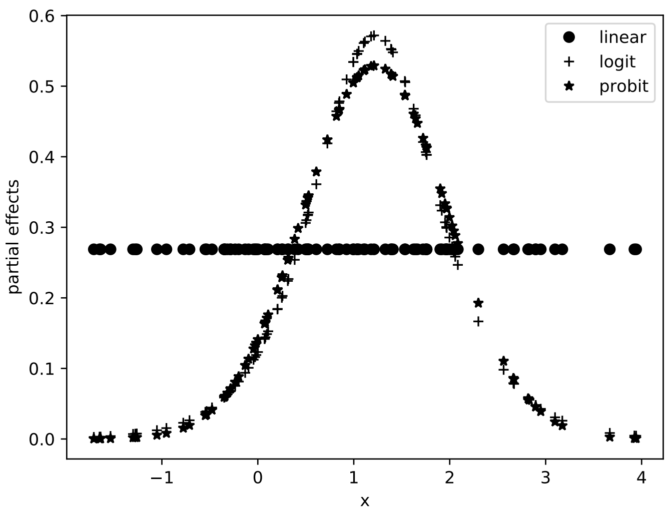
$$\frac{\partial \hat{y}}{\partial x_j} = \frac{\partial G(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k)}{\partial x_j} \tag{17.9}$$

$$= \hat{\beta}_j \cdot g(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k), \tag{17.10}$$

where $g(z)$ is the derivative of the link function $G(z)$. So

- for the probit model, the partial effect is
  $\frac{\partial \hat{y}}{\partial x_j} = \hat{\beta}_j \cdot \phi(\mathbf{x}\hat{\boldsymbol{\beta}})$

Figure 17.2. Partial Effects for Binary Response Models (Simulated Data)



- for the logit model, it is
  $\frac{\partial \hat{y}}{\partial x_j} = \hat{\beta}_j \cdot \lambda(\mathbf{x}\hat{\boldsymbol{\beta}})$

where $\phi(z)$ and $\lambda(z)$ are the PDFs of the standard normal and the logistic distribution, respectively.

The partial effect depends on the value of $\mathbf{x}\hat{\boldsymbol{\beta}}$. The PDFs have the famous bell-shape with highest values in the middle and values close to zero in the tails. This is already obvious from Figure 17.1. Depending on the value of x, the slope of the probability differs. For our simulated data set, Figure 17.2 shows the estimated partial effects for all 100 observed x values. Interested readers can see the complete code for this as Script 17.8 (`Binary-Margeff.py`) in Appendix IV (p. 399).

The fact that the partial effects differ by regressor values makes it harder to present the results in a concise and meaningful way. There are two common ways to aggregate the partial effects:

- Partial effects at the average: $PEA = \hat{\beta}_j \cdot g(\overline{\mathbf{x}}\hat{\boldsymbol{\beta}})$
- Average partial effects: $APE = \frac{1}{n}\sum_{i=1}^{n} \hat{\beta}_j \cdot g(\mathbf{x}_i\hat{\boldsymbol{\beta}}) = \hat{\beta}_j \cdot \overline{g(\mathbf{x}\hat{\boldsymbol{\beta}})}$

where $\overline{\mathbf{x}}$ is the vector of sample averages of the regressors and $\overline{g(\mathbf{x}\hat{\boldsymbol{\beta}})}$ is the sample average of g evaluated at the individual linear index $\mathbf{x}_i\hat{\boldsymbol{\beta}}$. Both measures multiply each coefficient $\hat{\beta}_j$ with a constant factor.

The first part of Script 17.9 (`Example-17-1-7.py`) implements the APE calculations for our labor force participation example using already known functions:

1. The linear indices $\mathbf{x}_i\hat{\boldsymbol{\beta}}$ are accessed using **`fittedvalues`**.
2. The factors $\overline{g(\mathbf{x}\hat{\boldsymbol{\beta}})}$ are calculated by using the PDF functions **`logistic.pdf`** and **`norm.pdf`** from the module **`scipy`** and then averaging over the sample with **`mean`**.
3. The APEs are calculated by multiplying the coefficients obtained with **`params`** with the corresponding factor. Note that for the linear probability model, the partial effects are constant and simply equal to the coefficients.

The second part of Script 17.9 (`Example-17-1-7.py`) shows how this can be done conveniently by using the method **`get_margeff()`**. All values (except the constant) are replicated. APEs for the constant are not part of the methods output since they do not have a direct meaningful interpretation. The APEs for the other variables don't differ too much between the models. As a general observation, as long as we are interested in APEs only and not in individual predictions or partial effects and as long as not too many probabilities are close to 0 or 1, the linear probability model often works well enough.

Script 17.9: `Example-17-1-7.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# estimate models:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper + I(exper**2) +'
                          'age + kidslt6 + kidsge6', data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

reg_logit = smf.logit(formula='inlf ~ nwifeinc + educ + exper + I(exper**2) +'
                              'age + kidslt6 + kidsge6', data=mroz)
results_logit = reg_logit.fit(disp=0)

reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper + I(exper**2) +'
                                'age + kidslt6 + kidsge6', data=mroz)
results_probit = reg_probit.fit(disp=0)

# manual average partial effects:
APE_lin = np.array(results_lin.params)

xb_logit = results_logit.fittedvalues
factor_logit = np.mean(stats.logistic.pdf(xb_logit))
APE_logit_manual = results_logit.params * factor_logit

xb_probit = results_probit.fittedvalues
factor_probit = np.mean(stats.norm.pdf(xb_probit))
APE_probit_manual = results_probit.params * factor_probit

table_manual = pd.DataFrame({'APE_lin': np.round(APE_lin, 4),
                             'APE_logit_manual': np.round(APE_logit_manual, 4),
                             'APE_probit_manual': np.round(APE_probit_manual, 4)})
print(f'table_manual: \n{table_manual}\n')

# automatic average partial effects:
coef_names = np.array(results_lin.model.exog_names)
coef_names = np.delete(coef_names, 0)  # drop Intercept

APE_logit_autom = results_logit.get_margeff().margeff
APE_probit_autom = results_probit.get_margeff().margeff

table_auto = pd.DataFrame({'coef_names': coef_names,
                           'APE_logit_autom': np.round(APE_logit_autom, 4),
                           'APE_probit_autom': np.round(APE_probit_autom, 4)})
print(f'table_auto: \n{table_auto}\n')
```

Output of Script 17.9: `Example-17-1-7.py`

```
table_manual:
                 APE_lin  APE_logit_manual  APE_probit_manual
Intercept         0.5855            0.0760             0.0812
nwifeinc         -0.0034           -0.0038            -0.0036
educ              0.0380            0.0395             0.0394
exper             0.0395            0.0368             0.0371
I(exper ** 2)    -0.0006           -0.0006            -0.0006
age              -0.0161           -0.0157            -0.0159
kidslt6          -0.2618           -0.2578            -0.2612
kidsge6           0.0130            0.0107             0.0108

table_auto:
      coef_names  APE_logit_autom  APE_probit_autom
0       nwifeinc          -0.0038           -0.0036
1           educ           0.0395            0.0394
2          exper           0.0368            0.0371
3  I(exper ** 2)          -0.0006           -0.0006
4            age          -0.0157           -0.0159
5        kidslt6          -0.2578           -0.2612
6        kidsge6           0.0107            0.0108
```

17.2. Count Data: The Poisson Regression Model

Instead of just 0/1-coded binary data, count data can take any non-negative integer $0, 1, 2, \ldots$. If they take very large numbers (like the number of students in a school), they can be approximated reasonably well as continuous variables in linear models and estimated using OLS. If the numbers are relatively small (like the number of children of a mother), this approximation might not work well. For example, predicted values can become negative.

The Poisson regression model is the most basic and convenient model explicitly designed for count data. The probability that y takes any value $h \in \{0, 1, 2, \ldots\}$ for this model can be written as

$$\mathrm{P}(y = h|\mathbf{x}) = \frac{e^{-e^{\mathbf{x}\boldsymbol{\beta}}} \cdot e^{h \cdot \mathbf{x}\boldsymbol{\beta}}}{h!}. \tag{17.11}$$

The parameters of the Poisson model are much easier to interpret than those of a probit or logit model. In this model, the conditional mean of y is

$$\mathrm{E}(y|\mathbf{x}) = e^{\mathbf{x}\boldsymbol{\beta}}, \tag{17.12}$$

so each slope parameter β_j has the interpretation of a semi elasticity:

$$\frac{\partial \mathrm{E}(y|\mathbf{x})}{\partial x_j} = \beta_j \cdot e^{\mathbf{x}\boldsymbol{\beta}} = \beta_j \cdot \mathrm{E}(y|\mathbf{x}) \tag{17.13}$$

$$\Leftrightarrow \beta_j = \frac{1}{\mathrm{E}(y|\mathbf{x})} \cdot \frac{\partial \mathrm{E}(y|\mathbf{x})}{\partial x_j}. \tag{17.14}$$

If x_j increases by one unit (and the other regressors remain the same), $\mathrm{E}(y|\mathbf{x})$ will increase roughly by $100 \cdot \beta_j$ percent (the exact value is once again $100 \cdot (e^{\beta_j} - 1)$).

A problem with the Poisson model is that it is quite restrictive. The Poisson distribution implicitly restricts the variance of y to be equal to its mean. If this assumption is violated but the conditional mean is still correctly specified, the Poisson parameter estimates are consistent, but the standard errors and all inferences based on them are invalid. A simple solution is to interpret the Poisson estimators as quasi-maximum likelihood estimators (QMLE). Similar to the heteroscedasticity-robust inference for OLS discussed in Section 8.1, the standard errors can be adjusted.

Estimating Poisson regression models in **statsmodels** is straightforward. They can be estimated using the convenient formula syntax and the command **poisson**. For the more robust QMLE standard errors, we use the command **glm** with **family=sm.families.Poisson()**.

Wooldridge, Example 17.3: Poisson Regression for Number of Arrests

We apply the Poisson regression model to study the number of arrests of young men in 1986. Script 17.10 (`Example-17-3.py`) imports the data and first estimates a linear regression model using OLS. Then, a Poisson model is estimated using **poisson**. Finally, we estimate the same model using the QMLE specification with **glm** to adjust the standard errors for a potential violation of the Poisson distribution. By construction, the parameter estimates are the same, but the standard errors are larger for the QMLE.

Script 17.10: **Example-17-3.py**

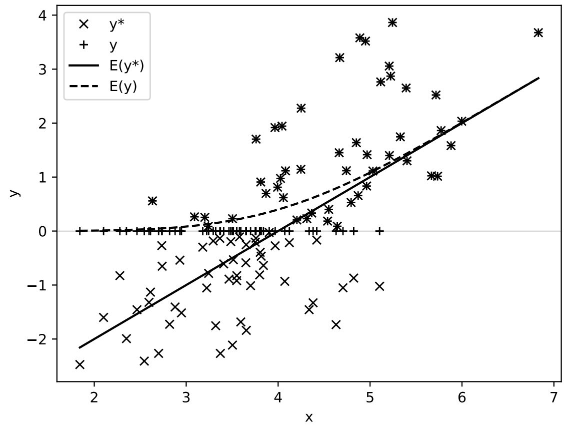
```
import wooldridge as woo
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

crime1 = woo.dataWoo('crime1')

# estimate linear model:
reg_lin = smf.ols(formula='narr86 ~ pcnv + avgsen + tottime + ptime86 +'
                          'qemp86 + inc86 + black + hispan + born60',
                  data=crime1)
results_lin = reg_lin.fit()

# print regression table:
table_lin = pd.DataFrame({'b': round(results_lin.params, 4),
                          'se': round(results_lin.bse, 4),
                          't': round(results_lin.tvalues, 4),
                          'pval': round(results_lin.pvalues, 4)})
print(f'table_lin: \n{table_lin}\n')

# estimate Poisson model:
reg_poisson = smf.poisson(formula='narr86 ~ pcnv + avgsen + tottime +'
                                  'ptime86 + qemp86 + inc86 + black +'
                                  'hispan + born60',
                          data=crime1)
results_poisson = reg_poisson.fit(disp=0)

# print regression table:
table_poisson = pd.DataFrame({'b': round(results_poisson.params, 4),
                              'se': round(results_poisson.bse, 4),
                              't': round(results_poisson.tvalues, 4),
                              'pval': round(results_poisson.pvalues, 4)})
print(f'table_poisson: \n{table_poisson}\n')
```

```
# estimate Quasi-Poisson model:
reg_qpoisson = smf.glm(formula='narr86 ~ pcnv + avgsen + tottime + ptime86 +'
                               'qemp86 + inc86 + black + hispan + born60',
                       family=sm.families.Poisson(),
                       data=crime1)
# the argument scale controls for the dispersion in exponential dispersion models,
# see the module documentation for more details:
results_qpoisson = reg_qpoisson.fit(scale='X2', disp=0)

# print regression table:
table_qpoisson = pd.DataFrame({'b': round(results_qpoisson.params, 4),
                               'se': round(results_qpoisson.bse, 4),
                               't': round(results_qpoisson.tvalues, 4),
                               'pval': round(results_qpoisson.pvalues, 4)})
print(f'table_qpoisson: \n{table_qpoisson}\n')
```

Output of Script 17.10: `Example-17-3.py`

```
table_lin:
                b      se        t    pval
Intercept  0.5766  0.0379  15.2150  0.0000
pcnv      -0.1319  0.0404  -3.2642  0.0011
avgsen    -0.0113  0.0122  -0.9257  0.3547
tottime    0.0121  0.0094   1.2790  0.2010
ptime86   -0.0409  0.0088  -4.6378  0.0000
qemp86    -0.0513  0.0145  -3.5420  0.0004
inc86     -0.0015  0.0003  -4.2613  0.0000
black      0.3270  0.0454   7.1987  0.0000
hispan     0.1938  0.0397   4.8799  0.0000
born60    -0.0225  0.0333  -0.6747  0.4999

table_poisson:
                b      se       t    pval
Intercept -0.5996  0.0673 -8.9158  0.0000
pcnv      -0.4016  0.0850 -4.7260  0.0000
avgsen    -0.0238  0.0199 -1.1918  0.2333
tottime    0.0245  0.0148  1.6603  0.0969
ptime86   -0.0986  0.0207 -4.7625  0.0000
qemp86    -0.0380  0.0290 -1.3099  0.1902
inc86     -0.0081  0.0010 -7.7624  0.0000
black      0.6608  0.0738  8.9503  0.0000
hispan     0.4998  0.0739  6.7609  0.0000
born60    -0.0510  0.0641 -0.7967  0.4256

table_qpoisson:
                b      se       t    pval
Intercept -0.5996  0.0828 -7.2393  0.0000
pcnv      -0.4016  0.1046 -3.8373  0.0001
avgsen    -0.0238  0.0246 -0.9677  0.3332
tottime    0.0245  0.0182  1.3481  0.1776
ptime86   -0.0986  0.0255 -3.8670  0.0001
qemp86    -0.0380  0.0357 -1.0636  0.2875
inc86     -0.0081  0.0013 -6.3028  0.0000
black      0.6608  0.0909  7.2673  0.0000
hispan     0.4998  0.0910  5.4896  0.0000
born60    -0.0510  0.0789 -0.6469  0.5177
```

Figure 17.3. Conditional Means for the Tobit Model



17.3. Corner Solution Responses: The Tobit Model

Corner solutions describe situations where the variable of interest is continuous but restricted in range. Typically, it cannot be negative. A significant share of people buy exactly zero amounts of alcohol, tobacco, or diapers. The Tobit model explicitly models dependent variables like this. It can be formulated in terms of a latent variable y^* that can take all real values. For it, the classical linear regression model assumptions MLR.1–MLR.6 are assumed to hold. If y^* is positive, we observe $y = y^*$. Otherwise, $y = 0$. Wooldridge (2019, Section 17.2) shows how to derive properties and the likelihood function for this model.

The problem of interpreting the parameters is similar to logit or probit models. While β_j measures the *ceteris paribus* effect of x_j on $\mathrm{E}(y^*|\mathbf{x})$, the interest is typically in y instead. The partial effect of interest can be written as

$$\frac{\partial \mathrm{E}(y|\mathbf{x})}{\partial x_j} = \beta_j \cdot \Phi\left(\frac{\mathbf{x}\boldsymbol{\beta}}{\sigma}\right) \tag{17.15}$$

and again depends on the regressor values $\mathbf{x}$. To aggregate them over the sample, we can either calculate the partial effects at the average (PEA) or the average partial effect (APE) just like with the binary variable models.

Figure 17.3 depicts these properties for a simulated data set with only one regressor. Whenever $y^* > 0$, $y = y^*$ and the symbols $\times$ and $+$ are on top of each other. If $y^* < 0$, then $y = 0$. Therefore, the slope of $\mathrm{E}(y|x)$ gets close to zero for very low x values. The code that generated the data set and the graph is hidden as Script 17.11 (`Tobit-CondMean.py`) in Appendix IV (p. 402).

We use **`statsmodels`** for the practical ML estimation, but not in the usual way. The reason is that there is no boxed routine to perform the estimation so we have to come up with our own definition of a log likelihood. Once we have done this, we let **`statsmodels`** do the rest. Before you have a look at Script 17.12 (`Example-17-2.py`) you might want to repeat Section 1.8.4. The basic idea is to inherit from the class **`GenericLikelihoodModel`** in **`statsmodels`**, i.e. we reuse its attributes and methods and call this new class **`Tobit`**. Now, we define the method **`nloglikeobs`**, which simply gives the code to obtain the negative log likelihood per observation for a given set of parameters (i.e. data and coefficients you want to estimate). Wooldridge (2019) provides details on the definition of the log likelihood we have implemented here. To keep things simple, we make no use of formula syntax and provide the data as matrices with the help of **`patsy`**. Because we inherited from **`GenericLikelihoodModel`** the new class **`Tobit`** also has the method **`fit`**, which internally calls **nloglikeobs** multiple times with different values for **params** to find an optimum of the provided log likelihood. We provide OLS results as a start solution for this optimization procedure. We finally use the (inherited) method **summary** to print out nicely formatted outputs with the estimated coefficients.

Wooldridge, Example 17.2: Married Women's Annual Labor Supply

We have already estimated labor supply models for the women in the data set `mroz`, ignoring the fact that the hours worked is necessarily non-negative. Script 17.12 (`Example-17-2.py`) estimates a Tobit model accounting for this fact.

Script 17.12: Example-17-2.py

```
import wooldridge as woo
import numpy as np
import patsy as pt
import scipy.stats as stats
import statsmodels.formula.api as smf
import statsmodels.base.model as smclass

mroz = woo.dataWoo('mroz')
y, X = pt.dmatrices('hours ~ nwifeinc + educ + exper +'
                    'I(exper**2)+ age + kidslt6 + kidsge6',
                    data=mroz, return_type='dataframe')

# generate starting solution:
reg_ols = smf.ols(formula='hours ~ nwifeinc + educ + exper + I(exper**2) +'
                          'age + kidslt6 + kidsge6', data=mroz)
results_ols = reg_ols.fit()
sigma_start = np.log(sum(results_ols.resid ** 2) / len(results_ols.resid))
params_start = np.concatenate((np.array(results_ols.params), sigma_start),
                              axis=None)

# extend statsmodels class by defining nloglikeobs:
class Tobit(smclass.GenericLikelihoodModel):
    # define a function that returns the negative log likelihood per observation
    # for a set of parameters that is provided by the argument "params":
    def nloglikeobs(self, params):
        # objects in "self" are defined in the parent class:
        X = self.exog
        y = self.endog
        p = X.shape[1]
        # for details on the implementation see Wooldridge (2019), formula 17.22:
        beta = params[0:p]
        sigma = np.exp(params[p])
        y_hat = np.dot(X, beta)
        y_eq = (y == 0)
        y_g = (y > 0)
        ll = np.empty(len(y))
        ll[y_eq] = np.log(stats.norm.cdf(-y_hat[y_eq] / sigma))
        ll[y_g] = np.log(stats.norm.pdf((y - y_hat)[y_g] / sigma)) - np.log(sigma)
        # return an array of log likelihoods for each observation:
        return -ll

# results of MLE:
reg_tobit = Tobit(endog=y, exog=X)
results_tobit = reg_tobit.fit(start_params=params_start, maxiter=10000, disp=0)
print(f'results_tobit.summary(): \n{results_tobit.summary()}\n')
```

Output of Script 17.12: `Example-17-2.py`

```
results_tobit.summary():
                              Tobit Results
==============================================================================
Dep. Variable:                  hours   Log-Likelihood:                -3819.1
Model:                          Tobit   AIC:                             7654.
Method:            Maximum Likelihood   BIC:                             7691.
Date:                Thu, 14 May 2020
Time:                        12:36:10
No. Observations:                 753
Df Residuals:                     745
Df Model:                           7
=================================================================================
                    coef    std err          z      P>|z|      [0.025      0.975]
---------------------------------------------------------------------------------
Intercept       965.3055    446.435      2.162      0.031      90.309    1840.302
nwifeinc         -8.8142      4.459     -1.977      0.048     -17.554      -0.075
educ             80.6456     21.583      3.736      0.000      38.343     122.948
exper           131.5643     17.279      7.614      0.000      97.697     165.431
I(exper ** 2)    -1.8642      0.538     -3.467      0.001      -2.918      -0.810
age             -54.4050      7.418     -7.334      0.000     -68.945     -39.865
kidslt6        -894.0217    111.878     -7.991      0.000   -1113.298    -674.745
kidsge6         -16.2180     38.640     -0.420      0.675     -91.952      59.516
par0              7.0229      0.037    189.514      0.000       6.950       7.096
=================================================================================
```

17.4. Censored and Truncated Regression Models

Censored regression models are closely related to Tobit models. In fact, their parameters can be estimated with nearly the same procedure discussed in the previous section. General censored regression models also start from a latent variable y^*. The observed dependent variable y is equal to y^* for some (the uncensored) observations. For the other observations, we only know an upper or lower bound for y^*. In the basic Tobit model, we observe $y = y^*$ in the "uncensored" cases with $y^* > 0$ and we only know that $y^* \leq 0$ if we observe $y = 0$. The censoring rules can be much more general. There could be censoring from above or the thresholds can vary from observation to observation.

The main difference between Tobit and censored regression models is the interpretation. In the former case, we are interested in the observed y, in the latter case, we are interested in the underlying y^*.[1] Censoring is merely a data problem that has to be accounted for instead of a logical feature of the dependent variable. We already know how to estimate Tobit models. With censored regression, we can use the same tools. The problem of calculating partial effects does not exist in this case since we are interested in the linear $\mathrm{E}(y^*|\mathbf{x})$ and the slope parameters are directly equal to the partial effects of interest.

[1] Wooldridge (2019, Section 17.4) uses the notation w instead of y and y instead of y^*.

Wooldridge, Example 17.4: Duration of Recidivism

We are interested in the criminal prognosis of individuals released from prison. We model the time it takes them to be arrested again. Explanatory variables include demographic characteristics as well as a dummy variable **workprg** indicating the participation in a work program during their time in prison. The 1445 former inmates observed in the data set recid were followed for a while.
During that time, 893 inmates were not arrested again. For them, we only know that their true duration y^* is at least **durat**, which for them is the time between the release and the end of the observation period, so we have right censoring. The threshold of censoring differs by individual depending on when they were released.
In Script 17.13 (Example-17-4.py) we inherit from **GenericLikelihoodModel** to create a class **CensReg**. Because of the more complicated selection rule, we have to update the **__init__** method by a parameter **cens**, which is a dummy variable indicating *censored* observations. Details on the foundation of the implementation for the log likelihood with right censored data in **nloglikeobs** is provided in Wooldridge (2019).
Estimates can directly be interpreted. Because of the logarithmic specification, they represent semi-elasticities. For example, do married individuals take around $100 \cdot \hat{\beta} = 34\%$ longer to be arrested again. (Actually, the accurate number is $100 \cdot (e^{\hat{\beta}} - 1) = 40\%$.) There is no significant effect of the work program.

Script 17.13: `Example-17-4.py`

```
import wooldridge as woo
import numpy as np
import patsy as pt
import scipy.stats as stats
import statsmodels.formula.api as smf
import statsmodels.base.model as smclass

recid = woo.dataWoo('recid')

# define dummy for censored observations:
censored = recid['cens'] != 0
y, X = pt.dmatrices('ldurat ~ workprg + priors + tserved + felon +'
                    'alcohol + drugs + black + married + educ + age',
                    data=recid, return_type='dataframe')

# generate starting solution:
reg_ols = smf.ols(formula='ldurat ~ workprg + priors + tserved + felon +'
                          'alcohol + drugs + black + married + educ + age',
                  data=recid)
results_ols = reg_ols.fit()
sigma_start = np.log(sum(results_ols.resid ** 2) / len(results_ols.resid))
params_start = np.concatenate((np.array(results_ols.params), sigma_start),
                              axis=None)


# extend statsmodels class by defining nloglikeobs:
class CensReg(smclass.GenericLikelihoodModel):
    def __init__(self, endog, cens, exog):
        self.cens = cens
        super(smclass.GenericLikelihoodModel, self).__init__(endog, exog,
                                                             missing='none')

    def nloglikeobs(self, params):
        X = self.exog
        y = self.endog
        cens = self.cens
        p = X.shape[1]
        beta = params[0:p]
        sigma = np.exp(params[p])
        y_hat = np.dot(X, beta)
        ll = np.empty(len(y))
        # uncensored:
        ll[~cens] = np.log(stats.norm.pdf((y - y_hat)[~cens] /
                                          sigma)) - np.log(sigma)
        # censored:
        ll[cens] = np.log(stats.norm.cdf(-(y - y_hat)[cens] / sigma))
        return -ll


# results of MLE:
reg_censReg = CensReg(endog=y, exog=X, cens=censored)
results_censReg = reg_censReg.fit(start_params=params_start,
                                  maxiter=10000, method='BFGS', disp=0)
print(f'results_censReg.summary(): \n{results_censReg.summary()}\n')
```

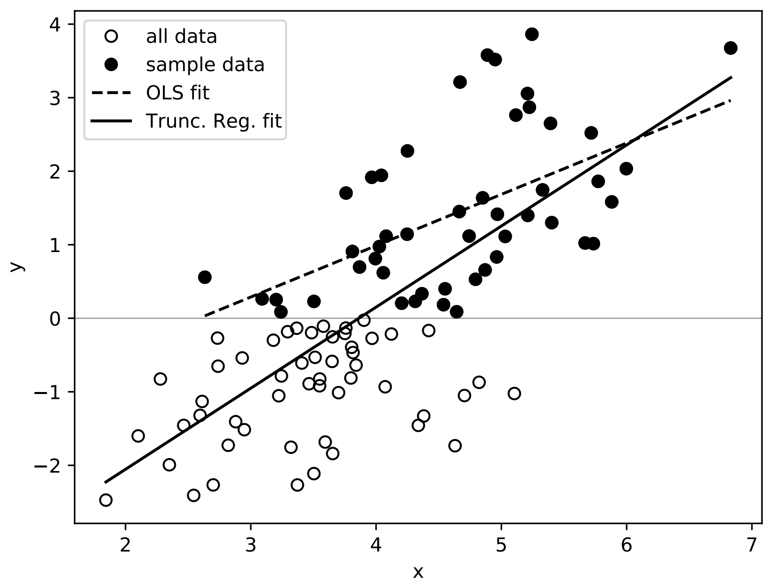
Output of Script 17.13: `Example-17-4.py`

```
results_censReg.summary():
                              CensReg Results
==============================================================================
Dep. Variable:                 ldurat   Log-Likelihood:                -1597.1
Model:                        CensReg   AIC:                             3216.
Method:            Maximum Likelihood   BIC:                             3274.
Date:                Thu, 14 May 2020
Time:                        12:36:12
No. Observations:                1445
Df Residuals:                    1434
Df Model:                          10
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      4.0994      0.348     11.796      0.000       3.418       4.781
workprg       -0.0626      0.120     -0.521      0.602      -0.298       0.173
priors        -0.1373      0.021     -6.396      0.000      -0.179      -0.095
tserved       -0.0193      0.003     -6.491      0.000      -0.025      -0.013
felon          0.4440      0.145      3.060      0.002       0.160       0.728
alcohol       -0.6349      0.144     -4.403      0.000      -0.918      -0.352
drugs         -0.2982      0.133     -2.246      0.025      -0.558      -0.038
black         -0.5427      0.117     -4.621      0.000      -0.773      -0.313
married        0.3407      0.140      2.436      0.015       0.067       0.615
educ           0.0229      0.025      0.902      0.367      -0.027       0.073
age            0.0039      0.001      6.450      0.000       0.003       0.005
par0           0.5936      0.034     17.249      0.000       0.526       0.661
==============================================================================
```

Truncation is a more serious problem than censoring since our observations are more severely affected. If the true latent variable y^* is above or below a certain threshold, the individual is not even sampled. We therefore do not even have any information. Classical truncated regression models rely on parametric and distributional assumptions to correct this problem. In **`statsmodels`** they can be implemented by providing an adjusted log likelihood just as discussed above. We will not go into details here, but Wooldridge (2019) describes how to implement the log likelihood.

Figure 17.4 shows results for a simulated data set. Because it is simulated, we actually know the values for everybody (hollow and solid dots). In our sample, we only observe those with $y > 0$ (solid dots). When applying OLS to this sample, we get a downward biased slope (dashed line). Truncated regression fixes this problem and gives a consistent slope estimator (solid line). Script 17.14 (`TruncReg-Simulation.py`) which generated the data set and the graph is shown in Appendix IV (p. 404).

Figure 17.4. Truncated Regression: Simulated Example



17.5. Sample Selection Corrections

Sample selection models are related to truncated regression models. We do have a random sample from the population of interest, but we do not observe the dependent variable y for a non-random sub-sample. The sample selection is not based on a threshold for y but on some other selection mechanism.

Heckman's selection model consists of a probit-like model for the binary fact whether y is observed and a linear regression-like model for y. Selection can be driven by the same determinants as y but should have at least one additional factor excluded from the equation for y. Wooldridge (2019, Section 17.5) discusses the specification and estimation of these models in more detail.

The classical Heckman selection model can be estimated either in two steps using software for probit and OLS as discussed by Wooldridge (2019) or by a specialized command using MLE. We will demonstrate the two step approach with **statsmodels**.

Wooldridge, Example 17.5: Wage offer Equation for Married Women

We once again look at the sample of women in the data set `MROZ`. Of the 753 women, 428 worked (`inlf=1`) and the rest did not work (`inlf=0`). For the latter, we do not observe the wage they would have gotten had they worked. Script 17.15 (`Example-17-5.py`) estimates the Heckman selection model using two formulas: one for the selection and one for the wage equation.

Script 17.15: `Example-17-5.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# step 1 (use all n observations to estimate a probit model of s_i on z_i):
reg_probit = smf.probit(formula='inlf ~ educ + exper + I(exper**2) +'
                                'nwifeinc + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)
pred_inlf = results_probit.fittedvalues
mroz['inv_mills'] = stats.norm.pdf(pred_inlf) / stats.norm.cdf(pred_inlf)

# step 2 (regress y_i on x_i and inv_mills in sample selection):
reg_heckit = smf.ols(formula='lwage ~ educ + exper + I(exper**2) + inv_mills',
                     subset=(mroz['inlf'] == 1), data=mroz)
results_heckit = reg_heckit.fit()

# print results:
print(f'results_heckit.summary(): \n{results_heckit.summary()}\n')
```

Output of Script 17.15: Example-17-5.py

```
results_heckit.summary():
                            OLS Regression Results
==============================================================================
Dep. Variable:                  lwage   R-squared:                       0.157
Model:                            OLS   Adj. R-squared:                  0.149
Method:                 Least Squares   F-statistic:                     19.69
Date:                Thu, 14 May 2020   Prob (F-statistic):           7.14e-15
Time:                        12:36:14   Log-Likelihood:                -431.57
No. Observations:                 428   AIC:                             873.1
Df Residuals:                     423   BIC:                             893.4
Df Model:                           4
Covariance Type:            nonrobust
=================================================================================
                    coef    std err          t      P>|t|      [0.025      0.975]
---------------------------------------------------------------------------------
Intercept        -0.5781      0.307     -1.885      0.060      -1.181       0.025
educ              0.1091      0.016      6.987      0.000       0.078       0.140
exper             0.0439      0.016      2.684      0.008       0.012       0.076
I(exper ** 2)    -0.0009      0.000     -1.946      0.052      -0.002    8.49e-06
inv_mills         0.0323      0.134      0.240      0.810      -0.232       0.296
==============================================================================
Omnibus:                       78.250   Durbin-Watson:                   1.958
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              299.801
Skew:                          -0.761   Prob(JB):                     7.93e-66
Kurtosis:                       6.807   Cond. No.                     3.61e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
[2] The condition number is large, 3.61e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

18. Advanced Time Series Topics

After we have introduced time series concepts in Chapters 10 – 12, this chapter touches on some more advanced topics in time series econometrics. Namely, we we look at infinite distributed lag models in Section 18.1, unit roots tests in Section 18.2, spurious regression in Section 18.3, cointegration in Section 18.4 and forecasting in Section 18.5.

18.1. Infinite Distributed Lag Models

We have covered finite distributed lag models in Section 10.3. We have estimated those and related models in *Python* using the module **`statsmodels`**. In *infinite* distributed lag models, shocks in the regressors z_t have an infinitely long impact on $y_t, y_{t+1}, \ldots$. The long-run propensity is the overall future effect of increasing z_t by one unit and keeping it at that level.

Without further restrictions, infinite distributed lag models cannot be estimated. Wooldridge (2019, Section 18.1) discusses two different models. The **geometric (or Koyck)** distributed lag model boils down to a linear regression equation in terms of lagged dependent variables

$$y_t = \alpha_0 + \gamma z_t + \rho y_{t-1} + v_t \tag{18.1}$$

and has a long-run propensity of

$$LRP = \frac{\gamma}{1-\rho}. \tag{18.2}$$

The **rational** distributed lag model can be written as a somewhat more general equation

$$y_t = \alpha_0 + \gamma_0 z_t + \rho y_{t-1} + \gamma_1 z_{t-1} + v_t \tag{18.3}$$

and has a long-run propensity of

$$LRP = \frac{\gamma_0 + \gamma_1}{1-\rho}. \tag{18.4}$$

In terms of the implementation of these models, there is nothing really new compared to Section 10.3. The only difference is that we include lagged dependent variables as regressors.

Wooldridge, Example 18.1: Housing Investment and Residential Price Inflation

Script 18.1 (`Example-18-1.py`) implements the geometric and the rational distributed lag models for the housing investment equation. The dependent variable is detrended by the method **`detrend`**, which simply uses the residual of a regression on a linear time trend. We store this detrended variable in the data frame.

The two models are estimated using **`statsmodels`** and a regression table very similar to Wooldridge (2019, Table 18.1) is produced. Finally, we estimate the LRP for both models using the formulas given above. We first extract the (named) coefficient and then do the calculations. For example, **`results_koyck.params["gprice"]`** is the coefficient with the label **`"gprice"`** which in our notation above corresponds to γ in the geometric distributed lag model.

Script 18.1: **Example-18-1.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm

hseinv = woo.dataWoo('hseinv')

# add lags and detrend:
hseinv['linvpc_det'] = sm.tsa.tsatools.detrend(hseinv['linvpc'])
hseinv['gprice_lag1'] = hseinv['gprice'].shift(1)
hseinv['linvpc_det_lag1'] = hseinv['linvpc_det'].shift(1)

# Koyck geometric d.l.:
reg_koyck = smf.ols(formula='linvpc_det ~ gprice + linvpc_det_lag1',
                    data=hseinv)
results_koyck = reg_koyck.fit()

# print regression table:
table_koyck = pd.DataFrame({'b': round(results_koyck.params, 4),
                            'se': round(results_koyck.bse, 4),
                            't': round(results_koyck.tvalues, 4),
                            'pval': round(results_koyck.pvalues, 4)})
print(f'table_koyck: \n{table_koyck}\n')

# rational d.l.:
reg_rational = smf.ols(formula='linvpc_det ~ gprice + linvpc_det_lag1 +'
                               'gprice_lag1',
                       data=hseinv)
results_rational = reg_rational.fit()

# print regression table:
table_rational = pd.DataFrame({'b': round(results_rational.params, 4),
                               'se': round(results_rational.bse, 4),
                               't': round(results_rational.tvalues, 4),
                               'pval': round(results_rational.pvalues, 4)})
print(f'table_rational: \n{table_rational}\n')

# LRP:
lrp_koyck = results_koyck.params['gprice'] / (
        1 - results_koyck.params['linvpc_det_lag1'])
print(f'lrp_koyck: {lrp_koyck}\n')

lrp_rational = (results_rational.params['gprice'] +
                results_rational.params['gprice_lag1']) / (
                       1 - results_rational.params['linvpc_det_lag1'])
print(f'lrp_rational: {lrp_rational}\n')
```

Output of Script 18.1: `Example-18-1.py`

```
table_koyck:
                       b      se        t    pval
Intercept        -0.0100  0.0179 -0.5561  0.5814
gprice            3.0948  0.9333  3.3159  0.0020
linvpc_det_lag1   0.3399  0.1316  2.5831  0.0138

table_rational:
                       b      se        t    pval
Intercept         0.0059  0.0169  0.3466  0.7309
gprice            3.2564  0.9703  3.3559  0.0019
linvpc_det_lag1   0.5472  0.1517  3.6076  0.0009
gprice_lag1      -2.9363  0.9732 -3.0172  0.0047

lrp_koyck: 4.688434194769012

lrp_rational: 0.7066808046888197
```

18.2. Testing for Unit Roots

We have covered strongly dependent unit root processes in Chapter 11 and promised to supply tests for unit roots later. There are several tests available. Conceptually, the Dickey-Fuller (DF) test is the simplest. If we want to test whether variable y has a unit root, we regress Δy_t on y_{t-1}. The test statistic is the usual t-test statistic of the slope coefficient. One problem is that because of the unit root, this test statistic is *not* t or normally distributed, not even asymptotically. Instead, we have to use special distribution tables for the critical values. The distribution also depends on whether we allow for a time trend in this regression.

The augmented Dickey-Fuller (ADF) test is a generalization that allows for richer dynamics in the process of y. To implement it, we add lagged values $\Delta y_{t-1}, \Delta y_{t-2}, \ldots$ to the differenced regression equation.

Of course, working with the special (A)DF tables of critical values is somewhat inconvenient. The module **`statsmodels`** offers automated DF and ADF tests for models with time trends. The command **`adfuller(y, maxlag = k)`** performs an ADF test with automatically selecting the number of lags in Δy (with **`k`** as the maximum amount of lags). For example, **`adfuller(y, maxlag = 0)`** requests zero lags, i.e. a simple DF test. If you set the argument **`autolag=None`** the value provided in **`maxlag`** determines the exact number of considered lags. The argument **`regression`** allows you to specify your model. Using **`regression='ct'`**, for example, means that you include a **`c`**onstant and a **`t`**rend.

Wooldridge, Example 18.4: Unit Root in Real GDP

Script 18.2 (`Example-18-4.py`) implements an ADF test for the logarithm of U.S. real GDP including a linear time trend. For a test with one lag in Δy and time trend, the equation to estimate is

$$\Delta y = \alpha + \theta y_{t-1} + \gamma_1 \Delta y_{t-1} + \delta_t t + e_t.$$

We already know how to implement such a regression using **`ols`**, so we demonstrate the use of **`adfuller`**. The relevant test statistic is $t = -2.421$ and the critical values are given in Wooldridge (2019, Table 18.3). More conveniently, the script also reports a p value of 0.37. So the null hypothesis of a unit root cannot be rejected with any reasonable significance level.

Script 18.2: `Example-18-4.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm

inven = woo.dataWoo('inven')
inven['lgdp'] = np.log(inven['gdp'])

# automated ADF:
res_ADF_aut = sm.tsa.stattools.adfuller(inven['lgdp'], maxlag=1, autolag=None,
                                        regression='ct', regresults=True)
ADF_stat_aut = res_ADF_aut[0]
ADF_pval_aut = res_ADF_aut[1]
table = pd.DataFrame({'names': res_ADF_aut[3].resols.model.exog_names,
                      'b': np.round(res_ADF_aut[3].resols.params, 4),
                      'se': np.round(res_ADF_aut[3].resols.bse, 4),
                      't': np.round(res_ADF_aut[3].resols.tvalues, 4),
                      'pval': np.round(res_ADF_aut[3].resols.pvalues, 4)})
print(f'table: \n{table}\n')
print(f'ADF_stat_aut: {ADF_stat_aut}\n')
print(f'ADF_pval_aut: {ADF_pval_aut}\n')
```

Output of Script 18.2: `Example-18-4.py`

```
table:
   names       b      se       t    pval
0     x1 -0.2096  0.0866 -2.4207  0.0215
1     x2  0.2638  0.1647  1.6010  0.1195
2  const  1.6627  0.6717  2.4752  0.0190
3     x3  0.0059  0.0027  2.1772  0.0372

ADF_stat_aut: -2.420732881476166

ADF_pval_aut: 0.3686558457135789
```

18.3. Spurious Regression

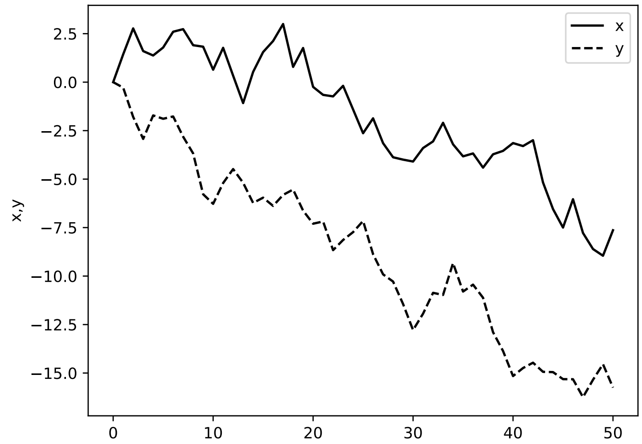
Unit roots generally destroy the usual (large sample) properties of estimators and tests. A leading example is spurious regression. Suppose two variables x and y are completely unrelated but both follow a random walk:

$$x_t = x_{t-1} + a_t$$
$$y_t = y_{t-1} + e_t,$$

where a_t and e_t are i.i.d. random innovations. If we want to test whether they are related from a random sample, we could simply regress y on x. A t test should reject the (true) null hypothesis that the slope coefficient is equal to zero with a probability of α, for example 5%. The phenomenon of spurious regression implies that this happens much more often.

Script 18.3 (`Simulate-Spurious-Regression-1.py`) simulates this model for one sample. Remember from Section 11.2 how to simulate a random walk in a simple way: with a starting value of zero, it is just the cumulative sum of the innovations. The time series for this simulated sample of size $n = 50$ is shown in Figure 18.1. When we regress y on x, the t statistic for the slope parameter is larger than 4 with a p value much smaller than 1%. So we would reject the (correct) null hypothesis that the variables are unrelated.

Figure 18.1. Spurious Regression: Simulated Data from Script 18.3



Script 18.3: `Simulate-Spurious-Regression-1.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# i.i.d. N(0,1) innovations:
n = 51
e = stats.norm.rvs(0, 1, size=n)
e[0] = 0
a = stats.norm.rvs(0, 1, size=n)
a[0] = 0

# independent random walks:
x = np.cumsum(a)
y = np.cumsum(e)
sim_data = pd.DataFrame({'y': y, 'x': x})

# regression:
reg = smf.ols(formula='y ~ x', data=sim_data)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

```
# graph:
plt.plot(x, color='black', marker='', linestyle='-', label='x')
plt.plot(y, color='black', marker='', linestyle='--', label='y')
plt.ylabel('x,y')
plt.legend()
plt.savefig('PyGraphs/Simulate-Spurious-Regression-1.pdf')
```

Output of Script 18.3: `Simulate-Spurious-Regression-1.py`

```
table:
                b      se        t  pval
Intercept -6.5100  0.3465 -18.7894   0.0
x          1.2695  0.0929  13.6607   0.0
```

We know that by definition, a valid test should reject a true null hypothesis with a probability of α, so maybe we were just unlucky with the specific sample we took. We therefore repeat the same analysis with 10,000 samples from the same data generating process in Script 18.4 (`Simulate-Spurious-Regression-2.py`). For each of the samples, we store the p value of the slope parameter in an array named **`pvals`**. After these simulations are run, we simply check how often we would have rejected $H_0 : \beta_1 = 0$ by comparing these p values with 0.05.

We find that in $6,652$ of the samples, so in 67% instead of $\alpha = 5\%$, we rejected H_0. So the t test seriously screws up the statistical inference because of the unit roots.

Script 18.4: `Simulate-Spurious-Regression-2.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

pvals = np.empty(10000)

# repeat r times:
for i in range(10000):
    # i.i.d. N(0,1) innovations:
    n = 51
    e = stats.norm.rvs(0, 1, size=n)
    e[0] = 0
    a = stats.norm.rvs(0, 1, size=n)
    a[0] = 0

    # independent random walks:
    x = np.cumsum(a)
    y = np.cumsum(e)
    sim_data = pd.DataFrame({'y': y, 'x': x})

    # regression:
    reg = smf.ols(formula='y ~ x', data=sim_data)
    results = reg.fit()
    pvals[i] = results.pvalues['x']
```

```
# how often is p<=5%:
count_pval_smaller = np.count_nonzero(pvals <= 0.05)  # counts True elements
print(f'count_pval_smaller: {count_pval_smaller}\n')

# how often is p>5%:
count_pval_greater = np.count_nonzero(pvals > 0.05)
print(f'count_pval_greater: {count_pval_greater}\n')
```

Output of Script 18.4: `Simulate-Spurious-Regression-2.py`

```
count_pval_smaller: 6652

count_pval_greater: 3348
```

18.4. Cointegration and Error Correction Models

In Section 18.3, we just saw that it is not a good idea to do linear regression with integrated variables. This is not generally true. If two variables are not only integrated (i.e. they have a unit root), but *cointegrated*, linear regression with them can actually make sense. Often, economic theory suggests a stable long-run relationship between integrated variables which implies cointegration. Cointegration implies that in the regression equation

$$y_t = \beta_0 + \beta_1 x_t + u_t,$$

the error term u does not have a unit root, while both y and x do. A test for cointegration can be based on this finding: We first estimate this model by OLS and then test for a unit root in the residuals $\hat{u}$. Again, we have to adjust the distribution of the test statistic and critical values. This approach is called Engle-Granger test in Wooldridge (2019, Section 18.4) or Phillips–Ouliaris (PO) test. See the documentation of **`coint`** in **`statsmodels`** for details on the implementation.

If we find cointegration, we can estimate error correction models. In the Engle-Granger procedure, these models can be estimated in a two-step procedure using OLS.

18.5. Forecasting

One major goal of time series analysis is forecasting. Given the information we have today, we want to give our best guess about the future and also quantify our uncertainty. Given a time series model for y, the best guess for y_{t+1} given information I_t is the conditional mean of $\mathrm{E}(y_{t+1}|I_t)$. For a model like

$$y_t = \delta_0 + \alpha_1 y_{t-1} + \gamma_1 z_{t-1} + u_t, \tag{18.5}$$

suppose we are at time t and know both y_t and z_t and want to predict y_{t+1}. Also suppose that $\mathrm{E}(u_t|I_{t-1}) = 0$. Then,

$$\mathrm{E}(y_{t+1}|I_t) = \delta_0 + \alpha_1 y_t + \gamma_1 z_t \tag{18.6}$$

and our prediction from an estimated model would be $\hat{y}_{t+1} = \hat{\delta}_0 + \hat{\alpha}_1 y_t + \hat{\gamma}_1 z_t$.

We already know how to get in-sample and (hypothetical) out-of-sample predictions including forecast intervals from linear models using the command **`get_prediction`**. It can also be used for our purposes.

There are several ways how the performance of forecast models can be evaluated. It makes a lot of sense not to look at the model fit within the estimation sample but at the out-of-sample forecast performances. Suppose we have used observations $y_1, \ldots, y_n$ for estimation and additionally have observations $y_{n+1}, \ldots, y_{n+m}$. For this set of observations, we obtain out-of-sample forecasts $f_{n+1}, \ldots, f_{n+m}$ and calculate the m forecast errors

$$e_t = y_t - f_t \qquad \text{for } t = n+1, \ldots, n+m. \tag{18.7}$$

We want these forecast errors to be as small (in absolute value) as possible. Useful measures are the root mean squared error ($RMSE$) and the mean absolute error (MAE):

$$RMSE = \sqrt{\frac{1}{m} \sum_{h=1}^{m} e_{n+h}^2} \tag{18.8}$$

$$MAE = \frac{1}{m} \sum_{h=1}^{m} \left| e_{n+h} \right| \tag{18.9}$$

(18.10)

Wooldridge, Example 18.8: Forecasting the U.S. Unemployment Rate

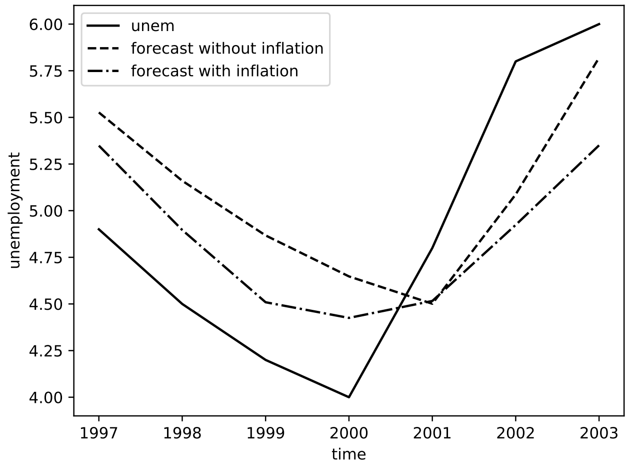
Script 18.5 (`Example-18-8.py`) estimates two simple models for forecasting the unemployment rate. The first one is a basic AR(1) model with only lagged unemployment as a regressor, the second one adds lagged inflation. We generate the Boolean variable **`yt96`** to restrict the estimation sample to years until 1996. After the estimation, we make predictions including 95% forecast intervals. Wooldridge (2019) explains how this can be done manually. We are somewhat lazy and simply use the command **`get_prediction`**.

Script 18.5 (`Example-18-8.py`) also calculates the forecast errors of the unemployment rate for the two models used in Example 18.8. Predictions are made for the other seven available years until 2003. The actual unemployment rate and the forecasts are plotted – the result is shown in Figure 18.2. Finally, we calculate the $RMSE$ and MAE for both models. Both measures suggest that the second model including the lagged inflation performs better.

Script 18.5: `Example-18-8.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

phillips = woo.dataWoo('phillips')

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=len(phillips), freq='Y')
phillips.index = date_range.year

# estimate models:
yt96 = (phillips['year'] <= 1996)
reg_1 = smf.ols(formula='unem ~ unem_1', data=phillips, subset=yt96)
results_1 = reg_1.fit()
reg_2 = smf.ols(formula='unem ~ unem_1 + inf_1', data=phillips, subset=yt96)
results_2 = reg_2.fit()

# predictions for 1997-2003 including 95% forecast intervals:
yf97 = (phillips['year'] > 1996)
pred_1 = results_1.get_prediction(phillips[yf97])
pred_1_FI = pred_1.summary_frame(
    alpha=0.05)[['mean', 'obs_ci_lower', 'obs_ci_upper']]
pred_1_FI.index = date_range.year[yf97]
print(f'pred_1_FI: \n{pred_1_FI}\n')

pred_2 = results_2.get_prediction(phillips[yf97])
pred_2_FI = pred_2.summary_frame(
    alpha=0.05)[['mean', 'obs_ci_lower', 'obs_ci_upper']]
pred_2_FI.index = date_range.year[yf97]
print(f'pred_2_FI: \n{pred_2_FI}\n')

# forecast errors:
e1 = phillips[yf97]['unem'] - pred_1_FI['mean']
e2 = phillips[yf97]['unem'] - pred_2_FI['mean']

# RMSE and MAE:
rmse1 = np.sqrt(np.mean(e1 ** 2))
print(f'rmse1: {rmse1}\n')
rmse2 = np.sqrt(np.mean(e2 ** 2))
print(f'rmse2: {rmse2}\n')
mae1 = np.mean(abs(e1))
print(f'mae1: {mae1}\n')
mae2 = np.mean(abs(e2))
print(f'mae2: {mae2}\n')

# graph:
plt.plot(phillips[yf97]['unem'], color='black', marker='', label='unem')
plt.plot(pred_1_FI['mean'], color='black',
         marker='', linestyle='--', label='forecast without inflation')
plt.plot(pred_2_FI['mean'], color='black',
         marker='', linestyle='-.', label='forecast with inflation')
plt.ylabel('unemployment')
plt.xlabel('time')
plt.legend()
plt.savefig('PyGraphs/Example-18-8.pdf')
```

Figure 18.2. Out-of-sample Forecasts for Unemployment



Output of Script 18.5: `Example-18-8.py`

```
pred_1_FI:
          mean  obs_ci_lower  obs_ci_upper
1997  5.526452      3.392840      7.660064
1998  5.160275      3.021340      7.299210
1999  4.867333      2.720958      7.013709
2000  4.647627      2.493832      6.801422
2001  4.501157      2.341549      6.660764
2002  5.087040      2.946509      7.227571
2003  5.819394      3.686837      7.951950

pred_2_FI:
          mean  obs_ci_lower  obs_ci_upper
1997  5.348468      3.548908      7.148027
1998  4.896451      3.090266      6.702636
1999  4.509137      2.693393      6.324881
2000  4.425175      2.607626      6.242724
2001  4.516062      2.696384      6.335740
2002  4.923537      3.118433      6.728641
2003  5.350271      3.540939      7.159603

rmse1: 0.5761199200210152

rmse2: 0.5217543207440963

mae1: 0.5420140442759066

mae2: 0.48419452667721685
```

19. Carrying Out an Empirical Project

We are now ready for serious empirical work. Chapter 19 of Wooldridge (2019) discusses the formulation of interesting theories, collection of raw data, and the writing of research papers. We are concerned with the data analysis part of a research project and will cover some aspects of using *Python* for real research.

This chapter is mainly about a few tips and tricks that might help to make our life easier by organizing the analyses and the output of *Python* in a systematic way. While we have worked with *Python* scripts throughout this book, Section 19.1 gives additional hints for using them effectively in larger projects. Section 19.2 shows how the results of our analyses can be written to a text file instead of just being displayed on the screen.

Section 19.3 discusses how Jupyter Notebooks can be used to generate nicely formatted documents that present *Python* code and output at least in a more structured way, potentially even ready for publication. Therefore we introduce Markdown, a straightforward markup language and $\LaTeX$ a widely used system which was for example used to generate this book. Jupyter Notebooks efficiently use *Python*, Markdown and $\LaTeX$ together to generate anything between clearly laid out results documentations and complete little research papers that automatically include the analysis results.

19.1. Working with *Python* Scripts

We already argued in Section 1.1.2 that anything we do in *Python* or any other statistical package should be done in scripts or the equivalent. In this way, it is always transparent how we generated our results. A typical empirical project has roughly the following steps:

1. Data Preparation: import raw data, recode and generate new variables, create sub-samples, ...
2. Generation of descriptive statistics, distribution of the main variables, ...
3. Estimation of the econometric models
4. Presentation of the results: tables, figures, ...

If we combine all these steps in one *Python* script, it is very easy for us to understand how we came up with the regression results even a year after we have done the analysis. At least as important: It is also easy for our thesis supervisor, collaborators or journal referees to understand where the results came from and to reproduce them. If we made a mistake at some point or get an updated raw data set, it is easy to repeat the whole analysis to generate new results.

It is crucial to add helpful comments to the *Python* scripts explaining what is done in each step. Scripts should start with an explanation like the following:

Script 19.1: ultimate-calcs.py

```
########################################################################
# Project X:
# "The Ultimate Question of Life, the Universe, and Everything"
# Project Collaborators: Mr. X, Mrs. Y
#
# Python Script "ultimate-calcs"
# by: F Heiss
# Date of this version: February 18, 2019
########################################################################
# external modules:
import numpy as np
import datetime as dt

# create a time stamp:
ts = dt.datetime.now()

# print to logfile.txt ('w' resets the logfile before writing output)
# in the provided path (make sure that the folder structure
# you may provide already exists):
print(f'This is a log file from: \n{ts}\n',
      file=open('Pyout/19/logfile.txt', 'w'))

# the first calculation using the function "square root" from numpy:
result1 = np.sqrt(1764)

# print to logfile.txt but with keeping the previous results ('a'):
print(f'result1: {result1}\n',
      file=open('Pyout/19/logfile.txt', 'a'))

# the second calculation reverses the first one:
result2 = result1 ** 2

# print to logfile.txt but with keeping the previous results ('a'):
print(f'result2: {result2}',
      file=open('Pyout/19/logfile.txt', 'a'))
```

In the next section, we will explain the details of Script 19.1 (`ultimate-calcs.py`). If a project requires many and/or time-consuming calculations, it might be useful to separate them into several *Python* scripts. For example, we could have four different scripts corresponding to the steps listed above:

- **data.py**
- **descriptives.py**
- **estimation.py**
- **results.py**

So once the potentially time-consuming data cleaning is done, we don't have to repeat it every time we run regressions. Instead, we save the cleaned data as an intermediary step and load it in subsequent analyses. To avoid confusion, it is highly advisable to document interdependencies. Both **descriptives.py** and **estimation.py** should at the beginning have a comment like:

```
# Depends on data.py
```

And **results.py** could have a comment like:

```
# Depends on estimation.py
```

19.2. Logging Output in Text Files

Having the results appear on the screen and being able to copy and paste from there might work for small projects. For larger projects, this is impractical. A straightforward way for writing all results to a file is to use the command **print** and route the output not to the console but a log file. If we want to write the output of a **print** command to a file `logfile.txt`, the basic syntax is:

```
print(result, file=open('logfile.txt', 'w'))
```

Script 19.1 (`ultimate-calcs.py`) gives a demonstration and also explains that the second argument of **open** controls for resetting the log file (**'w'**) or append the results to an existing one (**'a'**). See the documentation for other available options. We also include a time stamp, to document when we performed our analyses as the following log file resulting from Script 19.1 (`ultimate-calcs.py`) shows:

File **logfile.txt**

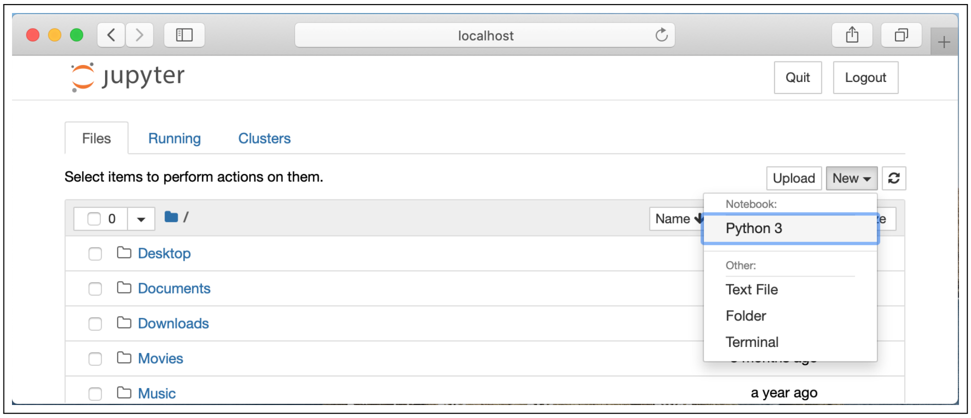
```
This is a log file from:
2020-05-14 12:57:38.996493

result1: 42.0

result2: 1764.0
```

There are other ways to document the results of your work. For example, you could globally define that all returns of **print** commands should be directed to the log file with **sys.stdout = open('logfile2.txt', 'w')**. Script 19.2 (`ultimate-calcs2.py`) demonstrates this alternative and produces the same log file. Finally, we want to mention the module **logging** providing a set of convenient functions to document events like errors or warnings during the execution of your program. For the scope of this book however, the usual **print** statement should be sufficient.

Script 19.2: **ultimate-calcs2.py**

```
# external modules:
import numpy as np
import datetime as dt
import sys

# make sure that the folder structure you may provide already exists:
sys.stdout = open('Pyout/19/logfile2.txt', 'w')

# create a time stamp:
ts = dt.datetime.now()

# print to logfile2.txt:
print(f'This is a log file from: \n{ts}\n')

# the first calculation using the function "square root" from numpy:
result1 = np.sqrt(1764)

# print to logfile2.txt:
print(f'result1: {result1}\n')

# the second calculation reverses the first one:
result2 = result1 ** 2

# print to logfile2.txt:
print(f'result2: {result2}')
```

Figure 19.1. Creating a Jupyter Notebook



19.3. Formatted Documents with Jupyter Notebook

Jupyter Notebook is an open source and web based environment that is maintained by the Project Jupyter.[1] A Jupyter Notebook is used to produce documents containing code, formatted text including equations and graphs. You can choose among many formats to export a Jupyter Notebook. Note that although we will use it for *Python* code only, many other languages like R or Julia are supported.[2]

The Anaconda distribution of *Python* already comes with everything we need to create a Jupyter Notebook. You can also install it manually as explained on `https://jupyter.org/`. In the following, we introduce the interface of Jupyter Notebook and the two important building blocks: `Code` and `Markdown` cells.

19.3.1. Getting Started

You find Jupyter Notebook in the Anaconda Navigator, which were both set up during the installation of Anaconda. After clicking on the icon, your web browser opens and should look similar to Figure 19.1. The figure also shows how to create a new Notebook: `New`→`Python 3`. This creates an empty Notebook similar as in Figure 19.2.

19.3.2. Cells

Let's start to enter some *Python* code into the displayed box starting with "`In[ ]:`" in Figure 19.2. This box is referred to as a "cell" in a Jupyter Notebook and we choose **`3**2`** as an exemplary input for such a cell in the upper screenshot in Figure 19.3. You can execute the code by clicking on ⏩ and immediately inspect the output in the appearing box starting with "`Out[ ]:`" (also shown in Figure 19.3). By default, Jupyter Notebook expects you to enter *Python* code in a cell, which is also visualized by the field next to ⏩ saying "`Code`". You can add more cells by clicking on +.

In the next step we create another cell and select `Markdown` in the drop down menu next to ⏩. We can now enter text and use Markdown commands to format it. The lower two screenshots of

[1] For more information, see Kluyver, Ragan-Kelley, Pérez, Granger, Bussonnier, Frederic, Kelley, Hamrick, Grout, Corlay, Ivanov, Avila, Abdalla, Willing, and development team (2016).

[2] Actually, the name Jupyter is based on the three languages **Ju**lia, **Pyt**hon and **R**.

Figure 19.2. An Empty Jupyter Notebook

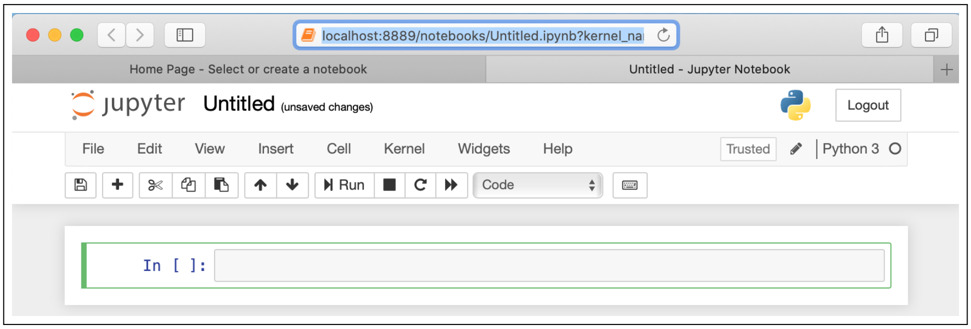


Figure 19.3 give an example. Here we use `**some text**` to print **bold text** and `*` to create a list with bullet points. More useful Markdown commands are explained in the next subsection. After entering the Markdown text click on ⏩ to apply your formatting commands. Instead of printing an output box, the cell you previously worked on is replaced by the formatted text. To edit the cell later, just double click on it.

To export your Notebook use `File`→`Download as` and choose a format, for example formatted `HTML` or `PDF` .

19.3.3. Markdown Basics

Markdown cells include normal text, formatting instructions and LaTeX equations.[3] There are countless possibilities to create appealing Markdown cells. We can only give a few examples for the most important formatting instructions:

- **`# Header 1`**, **`## Header 2`**, and **`### Header 3`** produce different levels of headers.
- **`*word*`** prints the word in *italics*.
- **`**word**`** prints the word in **bold**.
- **`` ``word`` ``** prints the word in code-like `typewriter font` (obviously **not** for *Python* code you want to execute).
- We can create lists with bullets using **`*`** at the beginning of a line followed by a whitespace.
- If you are familiar with LaTeX, displayed and inline formulas can be inserted using **`$...$`** and **`$$...$$`** and the usual LaTeX syntax, respectively.

Different formatting options are demonstrated in the following Jupyter Notebook. It can be downloaded in the `.ipynb` format from `http://www.UPfIE.net`. We start by showing you a collection of all `Code` and `Markdown` cells we entered in our Jupyter Notebook:

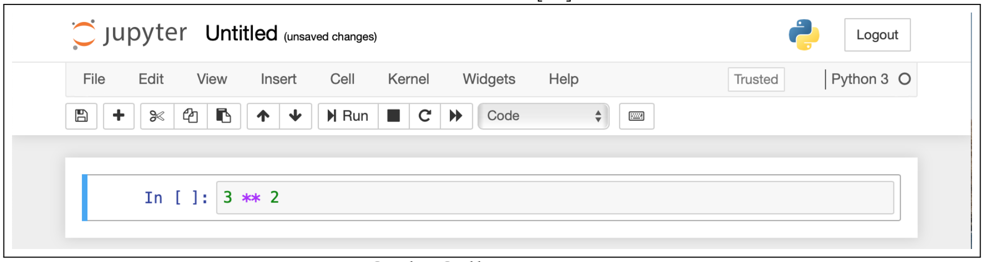
File **markdown-cell-1.txt**

```
# Working with Jupyter Notebook
The following example is based on Script ``Descr-Figures`` from Chapter 2 and
demonstrates the use of **Jupyter Notebooks** to document your work step by step.
We will describe the two most important building blocks:

* basic Markdown commands to format your text in ``Markdown`` cells
```

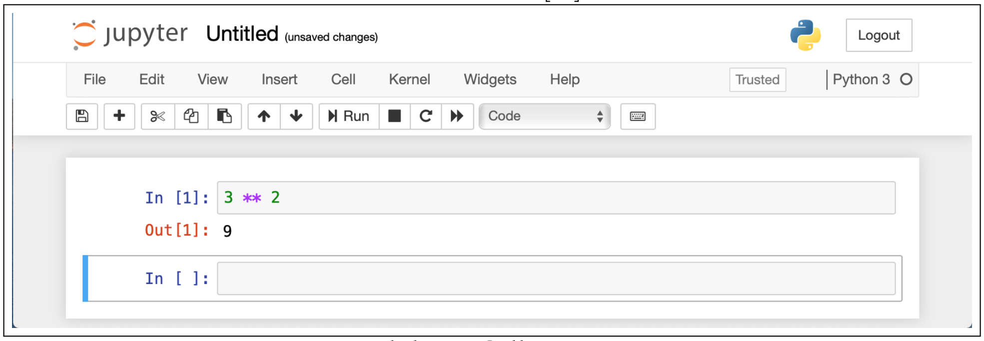
[3]LaTeX is a powerful and free system for generating documents. In economics and other fields with a lot of maths involved, it is widely used – in many areas, it is the *de facto* standard. It is also popular for typesetting articles and books. This book is an example for a complex document created by LaTeX. At least basic knowledge of LaTeX is needed to follow the equation related parts.

Figure 19.3. Cells in Jupyter Notebook

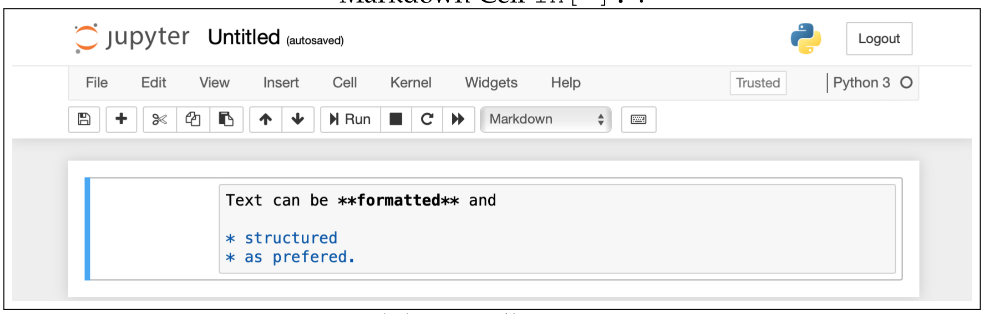
Code Cell `In[ ]:` :



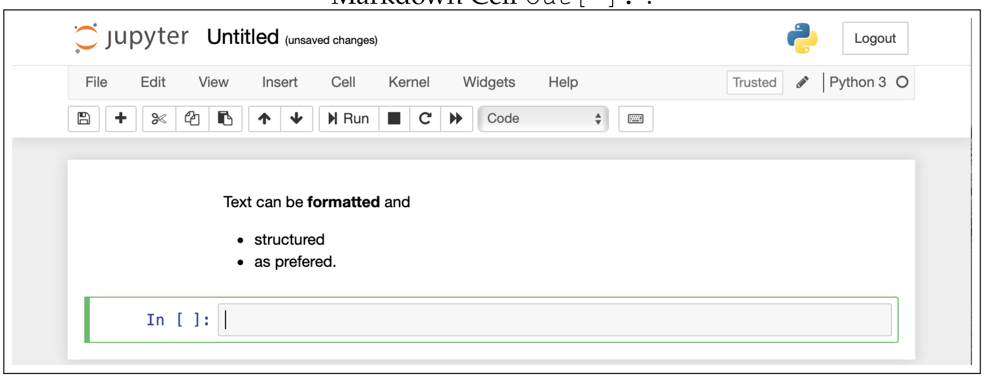
Code Cell `Out[ ]:` :



Markdown Cell `In[ ]:` :



Markdown Cell `Out[ ]:` :



```
* how to import and run Python code in ``Code`` cells

## Import and Prepare Data
Let's start by importing all external modules:
```

File code-cell-1.txt

```
import wooldridge as woo
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

File markdown-cell-2.txt

```
In the next step, we import our data and define important variables:
```

File code-cell-2.txt

```
affairs = woo.dataWoo('affairs')

# use a pandas.Categorical object to attach labels:
affairs['haskids'] = pd.Categorical.from_codes(affairs['kids'],
                                               categories=['no', 'yes'])
counts = affairs['haskids'].value_counts()
```

File markdown-cell-3.txt

```
## Analyse Data
### View your Data
To get an overview you could use ``affairs.head()``.

### Calculate Descriptive Statistics
Up to this point, the code cells above produced no output.
This will change now, as we are interested in some results.
Let's start with printing out the average age. We start with
its definition and use LaTeX to enter the equation:
$$ \bar{x} = \frac{1}{N} \sum_{i=1}^N x_{i} $$
The resulting Python code gives:
```

File code-cell-3.txt

```
age_mean = np.mean(affairs['age'])
print(age_mean)
```

File markdown-cell-4.txt

```
### Produce Graphic Results
In Chapter 2, we saw how to produce a pie chart. Let's repeat it here:
```

File code-cell-4.txt

```
plot = plt.pie(counts, labels=['no', 'yes'])
```

File markdown-cell-5.txt

````
You can also show Python code without executing it.
You can use ``inline code``, or for longer paragraphs
```python
plt.bar(['no', 'yes'], counts, color='dimgrey')
```
````

We exported the Jupyter Notebook into PDF and produced the following document:

Figure 19.4. Example of an Exported Jupyter Notebook

jupyter-example

May 14, 2020

1 Working with Jupyter Notebook

The following example is based on Script `Descr-Figures` from Chapter 2 and demonstrates the use of **Jupyter Notebooks** to document your work step by step. We will describe the two most important building blocks:

- basic Markdown commands to format your text in `Markdown` cells
- how to import and run Python code in `Code` cells

1.1 Import and Prepare Data

Let's start by importing all external modules:

[1]:
```
import wooldridge as woo
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In the next step, we import our data and define important variables:

[2]:
```
affairs = woo.dataWoo('affairs')

# use a pandas.Categorical object to attach labels:
affairs['haskids'] = pd.Categorical.from_codes(affairs['kids'],
                                               categories=['no', 'yes'])
counts = affairs['haskids'].value_counts()
```

1.2 Analyse Data

1.2.1 View your Data

To get an overview you could use `affairs.head()`.

1.2.2 Calculate Descriptive Statistics

Up to this point, the code cells above produced no output. This will change now, as we are interested in some results. Let's start with printing out the average age. We start with its definition and use LaTeX to enter the equation:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

1

Figure 19.5. Example of an Exported Jupyter Notebook (cont'ed)

The resulting Python code gives:

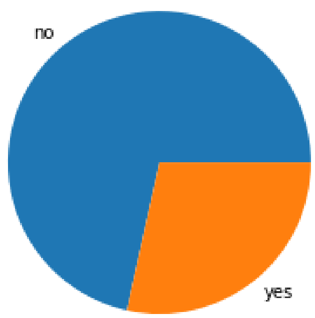
[4]:
```
age_mean = np.mean(affairs['age'])
print(age_mean)
```

```
32.48752079866888
```

1.2.3 Produce Graphic Results

In Chapter 2, we saw how to produce a pie chart. Let's repeat it here:

[5]:
```
plot = plt.pie(counts, labels=['no', 'yes'])
```



You can also show Python code without executing it. You can use `inline code`, or for longer paragraphs

```
plt.bar(['no', 'yes'], counts, color='dimgrey')
```

2

Part IV.

Appendices

Python Scripts

1. Scripts Used in Chapter 01

Script 1.1: `First-Python-Script.py`

```
# This is a comment.
# in the next line, we try to enter Shakespeare:
'To be, or not to be: that is the question'
# let's try some sensible math:
print((1 + 2) * 5)
16 ** 0.5
print('\n')
```

Script 1.2: `Python-as-a-Calculator.py`

```
result1 = 1 + 1
print(f'result1: {result1}\n')

result2 = 5 * (4 - 1) ** 2
print(f'result2: {result2}\n')

result3 = [result1, result2]
print(f'result3: \n{result3}\n')
```

Script 1.3: `Module-Math.py`

```
import math as someAlias

result1 = someAlias.sqrt(16)
print(f'result1: {result1}\n')

result2 = someAlias.pi
print(f'Pi: {result2}\n')

result3 = someAlias.e
print(f'Eulers number: {result3}\n')
```

Script 1.4: `Objects-in-Python.py`

```
result1 = 1 + 1
# determine the type:
type_result1 = type(result1)
# print the result:
print(f'type_result1: {type_result1}')

result2 = 2.5
type_result2 = type(result2)
print(f'type_result2: {type_result2}')

result3 = 'To be, or not to be: that is the question'
type_result3 = type(result3)
print(f'type_result3: {type_result3}\n')
```

Script 1.5: Lists-Copy.py

```
# define a list:
example_list = [1, 5, 41.3, 2.0]

# be careful with changes on variables pointing on example_list:
duplicate_list = example_list
duplicate_list[3] = 10000
print(f'duplicate_list: {duplicate_list}\n')
print(f'example_list: {example_list}\n')

# work on a copy of example_list:
example_list = [1, 5, 41.3, 2.0]
duplicate_list = example_list[:]
duplicate_list[3] = 10000
print(f'duplicate_list: {duplicate_list}\n')
print(f'example_list: {example_list}\n')
```

Script 1.6: Lists.py

```
# define a list:
example_list = [1, 5, 41.3, 2.0]
print(f'type(example_list): {type(example_list)}\n')

# access first entry by index:
first_entry = example_list[0]
print(f'first_entry: {first_entry}\n')

# access second to fourth entry by index:
range2to4 = example_list[1:4]
print(f'range2to4: {range2to4}\n')

# replace third entry by new value:
example_list[2] = 3
print(f'example_list: {example_list}\n')

# apply a function:
function_output = min(example_list)
print(f'function_output: {function_output}\n')

# apply a method:
example_list.sort()
print(f'example_list: {example_list}\n')

# delete third element of sorted list:
del example_list[2]
print(f'example_list: {example_list}\n')
```

Script 1.7: Dicts-Copy.py

```
# define and print a dict:
var1 = ['Florian', 'Daniel']
var2 = [96, 49]
var3 = [True, False]
example_dict = dict(name=var1, points=var2, passed=var3)
print(f'example_dict: {example_dict}\n')

# if you want to work on a copy:
import copy
copied_dict = copy.deepcopy(example_dict)
copied_dict['points'][1] = copied_dict['points'][1] - 40
```

```
print(f'example_dict: \n{example_dict}\n')
print(f'copied_dict: \n{copied_dict}\n')
```

Script 1.8: `Dicts.py`

```
# define and print a dict:
var1 = ['Florian', 'Daniel']
var2 = [96, 49]
var3 = [True, False]
example_dict = dict(name=var1, points=var2, passed=var3)
print(f'example_dict: \n{example_dict}\n')

# another way to define the dict:
example_dict2 = {'name': var1, 'points': var2, 'passed': var3}
print(f'example_dict2: \n{example_dict2}\n')

# get data type:
print(f'type(example_dict): {type(example_dict)}\n')

# access 'points':
points_all = example_dict['points']
print(f'points_all: {points_all}\n')

# access 'points' of Daniel:
points_daniel = example_dict['points'][1]
print(f'points_daniel: {points_daniel}\n')

# add 4 to 'points' of Daniel and let him pass:
example_dict['points'][1] = example_dict['points'][1] + 4
example_dict['passed'][1] = True
print(f'example_dict: \n{example_dict}\n')

# add a new variable 'grade':
example_dict['grade'] = [1.3, 4.0]

# delete variable 'points':
del example_dict['points']
print(f'example_dict: \n{example_dict}\n')
```

Script 1.9: `Numpy-Arrays.py`

```
import numpy as np

# define arrays in numpy:
testarray1D = np.array([1, 5, 41.3, 2.0])
print(f'type(testarray1D): {type(testarray1D)}\n')

testarray2D = np.array([[4, 9, 8, 3],
                        [2, 6, 3, 2],
                        [1, 1, 7, 4]])

# get dimensions of testarray2D:
dim = testarray2D.shape
print(f'dim: {dim}\n')

# access elements by indices:
third_elem = testarray1D[2]
print(f'third_elem: {third_elem}\n')

second_third_elem = testarray2D[1, 2]  # element in 2nd row and 3rd column
```

```
print(f'second_third_elem: {second_third_elem}\n')

second_to_third_col = testarray2D[:, 1:3]  # each row in the 2nd and 3rd column
print(f'second_to_third_col: \n{second_to_third_col}\n')

# access elements by lists:
first_third_elem = testarray1D[[0, 2]]
print(f'first_third_elem: {first_third_elem}\n')

# same with Boolean lists:
first_third_elem2 = testarray1D[[True, False, True, False]]
print(f'first_third_elem2: {first_third_elem2}\n')

k = np.array([[True, False, False, False],
              [False, False, True, False],
              [True, False, True, False]])
elem_by_index = testarray2D[k]  # 1st elem in 1st row, 3rd elem in 2nd row...
print(f'elem_by_index: {elem_by_index}\n')
```

Script 1.10: Numpy-SpecialCases.py

```
import numpy as np

# array of integers defined by the arguments start, end and sequence length:
sequence = np.linspace(0, 2, num=11)
print(f'sequence: \n{sequence}\n')

# sequence of integers starting at 0, ending at 5-1:
sequence_int = np.arange(5)
print(f'sequence_int: \n{sequence_int}\n')

# initialize array with each element set to zero:
zero_array = np.zeros((4, 3))
print(f'zero_array: \n{zero_array}\n')

# initialize array with each element set to one:
one_array = np.ones((2, 5))
print(f'one_array: \n{one_array}\n')

# uninitialized array (filled with arbitrary nonsense elements):
empty_array = np.empty((2, 3))
print(f'empty_array: \n{empty_array}\n')
```

Script 1.11: Numpy-Operations.py

```
import numpy as np

# define an arrays in numpy:
mat1 = np.array([[4, 9, 8],
                 [2, 6, 3]])
mat2 = np.array([[1, 5, 2],
                 [6, 6, 0],
                 [4, 8, 3]])

# use a numpy function:
result1 = np.exp(mat1)
print(f'result1: \n{result1}\n')

result2 = mat1 + mat2[[0, 1]]  # same as np.add(mat1, mat2[[0, 1]])
print(f'result2: \n{result2}\n')
```

```
# use a method:
mat1_tr = mat1.transpose()
print(f'mat1_tr: \n{mat1_tr}\n')

# matrix algebra:
matprod = mat1.dot(mat2)  # same as  mat1 @ mat2
print(f'matprod: \n{matprod}\n')
```

Script 1.12: `Pandas.py`

```
import numpy as np
import pandas as pd

# define a pandas DataFrame:
icecream_sales = np.array([30, 40, 35, 130, 120, 60])
weather_coded = np.array([0, 1, 0, 1, 1, 0])
customers = np.array([2000, 2100, 1500, 8000, 7200, 2000])
df = pd.DataFrame({'icecream_sales': icecream_sales,
                   'weather_coded': weather_coded,
                   'customers': customers})

# define and assign an index (six ends of month starting in April, 2010)
# (details on generating indices are given in Chapter 10):
ourIndex = pd.date_range(start='04/2010', freq='M', periods=6)
df.set_index(ourIndex, inplace=True)

# print the DataFrame
print(f'df: \n{df}\n')

# access columns by variable names:
subset1 = df[['icecream_sales', 'customers']]
print(f'subset1: \n{subset1}\n')

# access second to fourth row:
subset2 = df[1:4]  # same as df['2010-05-31':'2010-07-31']
print(f'subset2: \n{subset2}\n')

# access rows and columns by index and variable names:
subset3 = df.loc['2010-05-31', 'customers']  # same as df.iloc[1,2]
print(f'subset3: \n{subset3}\n')

# access rows and columns by index and variable integer positions:
subset4 = df.iloc[1:4, 0:2]
# same as df.loc['2010-05-31':'2010-07-31', ['icecream_sales','weather']]
print(f'subset4: \n{subset4}\n')
```

Script 1.13: `Pandas-Operations.py`

```
import numpy as np
import pandas as pd

# define a pandas DataFrame:
icecream_sales = np.array([30, 40, 35, 130, 120, 60])
weather_coded = np.array([0, 1, 0, 1, 1, 0])
customers = np.array([2000, 2100, 1500, 8000, 7200, 2000])
df = pd.DataFrame({'icecream_sales': icecream_sales,
                   'weather_coded': weather_coded,
                   'customers': customers})
```

```
# define and assign an index (six ends of month starting in April, 2010)
# (details on generating indices are given in Chapter 10):
ourIndex = pd.date_range(start='04/2010', freq='M', periods=6)
df.set_index(ourIndex, inplace=True)

# include sales two months ago:
df['icecream_sales_lag2'] = df['icecream_sales'].shift(2)
print(f'df: \n{df}\n')

# use a pandas.Categorical object to attach labels (0 = bad; 1 = good):
df['weather'] = pd.Categorical.from_codes(codes=df['weather_coded'],
                                          categories=['bad', 'good'])
print(f'df: \n{df}\n')

# mean sales for each weather category:
group_means = df.groupby('weather').mean()
print(f'group_means: \n{group_means}\n')
```

Script 1.14: `Wooldridge.py`

```
import wooldridge as woo

# load data:
wage1 = woo.dataWoo('wage1')

# get type:
print(f'type(wage1): \n{type(wage1)}\n')

# get an overview:
print(f'wage1.head(): \n{wage1.head()}\n')
```

Script 1.15: `Import-Export.py`

```
import pandas as pd

# import csv with pandas:
df1 = pd.read_csv('data/sales.csv', delimiter=',', header=None,
                  names=['year', 'product1', 'product2', 'product3'])
print(f'df1: \n{df1}\n')

# import txt with pandas:
df2 = pd.read_table('data/sales.txt', delimiter=' ')
print(f'df2: \n{df2}\n')

# add a row to df1:
df3 = df1.append({'year': 2014, 'product1': 10, 'product2': 8, 'product3': 2},
                 ignore_index=True)
print(f'df3: \n{df3}\n')

# export with pandas:
df3.to_csv('data/sales2.csv')
```

Script 1.16: `Import-StockData.py`

```
import pandas_datareader as pdr

# download data for 'F' (= Ford Motor Company) and define start and end:
tickers = ['F']
start_date = '2014-01-01'
end_date = '2015-12-31'
```

```
# use pandas_datareader for the import:
F_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# look at imported data:
print(f'F_data.head(): \n{F_data.head()}\n')
print(f'F_data.tail(): \n{F_data.tail()}\n')
```

Script 1.17: **Graphs-Basics.py**

```
import matplotlib.pyplot as plt

# create data:
x = [1, 3, 4, 7, 8, 9]
y = [0, 3, 6, 9, 7, 8]

# plot and save:
plt.plot(x, y, color='black')
plt.savefig('PyGraphs/Graphs-Basics-a.pdf')
plt.close()
```

Script 1.18: **Graphs-Basics2.py**

```
import matplotlib.pyplot as plt

# create data:
x = [1, 3, 4, 7, 8, 9]
y = [0, 3, 6, 9, 7, 8]

# plot and save:
plt.plot(x, y, color='black', linestyle='--')
plt.savefig('PyGraphs/Graphs-Basics-b.pdf')
plt.close()

plt.plot(x, y, color='black', linestyle=':')
plt.savefig('PyGraphs/Graphs-Basics-c.pdf')
plt.close()

plt.plot(x, y, color='black', linestyle='-', linewidth=3)
plt.savefig('PyGraphs/Graphs-Basics-d.pdf')
plt.close()

plt.plot(x, y, color='black', marker='o')
plt.savefig('PyGraphs/Graphs-Basics-e.pdf')
plt.close()

plt.plot(x, y, color='black', marker='v', linestyle='')
plt.savefig('PyGraphs/Graphs-Basics-f.pdf')
```

Script 1.19: **Graphs-Functions.py**

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support of quadratic function
# (creates an array with 100 equispaced elements from -3 to 2):
x1 = np.linspace(-3, 2, num=100)
# function values for all these values:
y1 = x1 ** 2
```

```
# plot quadratic function:
plt.plot(x1, y1, linestyle='-', color='black')
plt.savefig('PyGraphs/Graphs-Functions-a.pdf')
plt.close()

# same for normal density:
x2 = np.linspace(-4, 4, num=100)
y2 = stats.norm.pdf(x2)

# plot normal density:
plt.plot(x2, y2, linestyle='-', color='black')
plt.savefig('PyGraphs/Graphs-Functions-b.pdf')
```

Script 1.20: Graphs-BuildingBlocks.py

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support for all normal densities:
x = np.linspace(-4, 4, num=100)
# get different density evaluations:
y1 = stats.norm.pdf(x, 0, 1)
y2 = stats.norm.pdf(x, 1, 0.5)
y3 = stats.norm.pdf(x, 0, 2)

# plot:
plt.plot(x, y1, linestyle='-', color='black', label='standard normal')
plt.plot(x, y2, linestyle='--', color='0.3', label='mu = 1, sigma = 0.5')
plt.plot(x, y3, linestyle=':', color='0.6', label='$\mu = 0$, $\sigma = 2$')
plt.xlim(-3, 4)
plt.title('Normal Densities')
plt.ylabel('$\phi(x)$')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/Graphs-BuildingBlocks.pdf')
```

Script 1.21: Graphs-Export.py

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support for all normal densities:
x = np.linspace(-4, 4, num=100)

# get different density evaluations:
y1 = stats.norm.pdf(x, 0, 1)
y2 = stats.norm.pdf(x, 0, 3)

# plot (a):
plt.figure(figsize=(4, 6))
plt.plot(x, y1, linestyle='-', color='black')
plt.plot(x, y2, linestyle='--', color='0.3')
plt.savefig('PyGraphs/Graphs-Export-a.pdf')
plt.close()

# plot (b):
plt.figure(figsize=(6, 4))
```

```
plt.plot(x, y1, linestyle='-', color='black')
plt.plot(x, y2, linestyle='--', color='0.3')
plt.savefig('PyGraphs/Graphs-Export-b.png')
```

Script 1.22: `Descr-Tables.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd

affairs = woo.dataWoo('affairs')

# adjust codings to [0-4] (Categoricals require a start from 0):
affairs['ratemarr'] = affairs['ratemarr'] - 1

# use a pandas.Categorical object to attach labels for "haskids":
affairs['haskids'] = pd.Categorical.from_codes(affairs['kids'],
                                               categories=['no', 'yes'])
# ... and "marriage" (for example: 0 = 'very unhappy', 1 = 'unhappy',...):
mlab = ['very unhappy', 'unhappy', 'average', 'happy', 'very happy']
affairs['marriage'] = pd.Categorical.from_codes(affairs['ratemarr'],
                                                categories=mlab)

# frequency table in numpy (alphabetical order of elements):
ft_np = np.unique(affairs['marriage'], return_counts=True)
unique_elem_np = ft_np[0]
counts_np = ft_np[1]
print(f'unique_elem_np: \n{unique_elem_np}\n')
print(f'counts_np: \n{counts_np}\n')

# frequency table in pandas:
ft_pd = affairs['marriage'].value_counts()
print(f'ft_pd: \n{ft_pd}\n')

# frequency table with groupby:
ft_pd2 = affairs['marriage'].groupby(affairs['haskids']).value_counts()
print(f'ft_pd2: \n{ft_pd2}\n')

# contingency table in pandas:
ct_all_abs = pd.crosstab(affairs['marriage'], affairs['haskids'], margins=3)
print(f'ct_all_abs: \n{ct_all_abs}\n')
ct_all_rel = pd.crosstab(affairs['marriage'], affairs['haskids'], normalize='all')
print(f'ct_all_rel: \n{ct_all_rel}\n')

# share within "marriage" (i.e. within a row):
ct_row = pd.crosstab(affairs['marriage'], affairs['haskids'], normalize='index')
print(f'ct_row: \n{ct_row}\n')

# share within "haskids"  (i.e. within a column):
ct_col = pd.crosstab(affairs['marriage'], affairs['haskids'], normalize='columns')
print(f'ct_col: \n{ct_col}\n')
```

Script 1.23: `Descr-Figures.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

affairs = woo.dataWoo('affairs')
```

```
# attach labels (see previous script):
affairs['ratemarr'] = affairs['ratemarr'] - 1
affairs['haskids'] = pd.Categorical.from_codes(affairs['kids'],
                                                categories=['no', 'yes'])
mlab = ['very unhappy', 'unhappy', 'average', 'happy', 'very happy']
affairs['marriage'] = pd.Categorical.from_codes(affairs['ratemarr'],
                                                 categories=mlab)

# counts for all graphs:
counts = affairs['marriage'].value_counts()
counts_bykids = affairs['marriage'].groupby(affairs['haskids']).value_counts()
counts_yes = counts_bykids['yes']
counts_no = counts_bykids['no']

# pie chart (a):
grey_colors = ['0.3', '0.4', '0.5', '0.6', '0.7']
plt.pie(counts, labels=mlab, colors=grey_colors)
plt.savefig('PyGraphs/Descr-Pie.pdf')
plt.close()

# horizontal bar chart (b):
y_pos = [0, 1, 2, 3, 4]  # the y locations for the bars
plt.barh(y_pos, counts, color='0.6')
plt.yticks(y_pos, mlab, rotation=60)  # add and adjust labeling
plt.savefig('PyGraphs/Descr-Bar1.pdf')
plt.close()

# stacked bar plot (c):
x_pos = [0, 1, 2, 3, 4]  # the x locations for the bars
plt.bar(x_pos, counts_yes, width=0.4, color='0.6', label='Yes')
# with 'bottom=counts_yes' bars are added on top of previous ones:
plt.bar(x_pos, counts_no, width=0.4, bottom=counts_yes, color='0.3', label='No')
plt.ylabel('Counts')
plt.xticks(x_pos, mlab)  # add labels on x axis
plt.legend()
plt.savefig('PyGraphs/Descr-Bar2.pdf')
plt.close()

# grouped bar plot (d)
# add left bars first and move bars to the left:
x_pos_leftbar = [-0.2, 0.8, 1.8, 2.8, 3.8]
plt.bar(x_pos_leftbar, counts_yes, width=0.4, color='0.6', label='Yes')
# add right bars first and move bars to the right:
x_pos_rightbar = [0.2, 1.2, 2.2, 3.2, 4.2]
plt.bar(x_pos_rightbar, counts_no, width=0.4, color='0.3', label='No')
plt.ylabel('Counts')
plt.xticks(x_pos, mlab)
plt.legend()
plt.savefig('PyGraphs/Descr-Bar3.pdf')
```

Script 1.24: `Histogram.py`

```
import wooldridge as woo
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe:
roe = ceosal1['roe']
```

```
# subfigure a (histogram with counts):
plt.hist(roe, color='grey')
plt.ylabel('Counts')
plt.xlabel('roe')
plt.savefig('PyGraphs/Histogram1.pdf')
plt.close()

# subfigure b (histogram with density and explicit breaks):
breaks = [0, 5, 10, 20, 30, 60]
plt.hist(roe, color='grey', bins=breaks, density=True)
plt.ylabel('density')
plt.xlabel('roe')
plt.savefig('PyGraphs/Histogram2.pdf')
```

Script 1.25: `KDensity.py`

```
import wooldridge as woo
import statsmodels.api as sm
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe:
roe = ceosal1['roe']

# estimate kernel density:
kde = sm.nonparametric.KDEUnivariate(roe)
kde.fit()

# subfigure a (kernel density):
plt.plot(kde.support, kde.density, color='black', linewidth=2)
plt.ylabel('density')
plt.xlabel('roe')
plt.savefig('PyGraphs/Density1.pdf')
plt.close()

# subfigure b (kernel density with overlayed histogram):
plt.hist(roe, color='grey', density=True)
plt.plot(kde.support, kde.density, color='black', linewidth=2)
plt.ylabel('density')
plt.xlabel('roe')
plt.savefig('PyGraphs/Density2.pdf')
```

Script 1.26: `Descr-ECDF.py`

```
import wooldridge as woo
import numpy as np
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe:
roe = ceosal1['roe']

# calculate ECDF:
x = np.sort(roe)
n = x.size
y = np.arange(1, n + 1) / n  # generates cumulative shares of observations
```

```
# plot a step function:
plt.step(x, y, linestyle='-', color='black')
plt.xlabel('roe')
plt.savefig('PyGraphs/ecdf.pdf')
```

Script 1.27: `Descr-Stats.py`

```
import wooldridge as woo
import numpy as np

ceosal1 = woo.dataWoo('ceosal1')

# extract roe and salary:
roe = ceosal1['roe']
salary = ceosal1['salary']

# sample average:
roe_mean = np.mean(salary)
print(f'roe_mean: {roe_mean}\n')

# sample median:
roe_med = np.median(salary)
print(f'roe_med: {roe_med}\n')

# standard deviation:
roe_s = np.std(salary, ddof=1)
print(f'roe_s: {roe_s}\n')

# correlation with ROE:
roe_corr = np.corrcoef(roe, salary)
print(f'roe_corr: \n{roe_corr}\n')
```

Script 1.28: `Descr-Boxplot.py`

```
import wooldridge as woo
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# extract roe and salary:
roe = ceosal1['roe']
consprod = ceosal1['consprod']

# plotting descriptive statistics:
plt.boxplot(roe, vert=False)
plt.ylabel('roe')
plt.savefig('PyGraphs/Boxplot1.pdf')
plt.close()

# plotting descriptive statistics:
roe_cp0 = roe[consprod == 0]
roe_cp1 = roe[consprod == 1]

plt.boxplot([roe_cp0, roe_cp1])
plt.ylabel('roe')
plt.savefig('PyGraphs/Boxplot2.pdf')
```

Script 1.29: `PMF-binom.py`

```
import scipy.stats as stats
import math
```

```
# pedestrian approach:
c = math.factorial(10) / (math.factorial(2) * math.factorial(10 - 2))
p1 = c * (0.2 ** 2) * (0.8 ** 8)
print(f'p1: {p1}\n')

# scipy function:
p2 = stats.binom.pmf(2, 10, 0.2)
print(f'p2: {p2}\n')
```

Script 1.30: `PMF-example.py`

```
import scipy.stats as stats
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# values for x (all between 0 and 10):
x = np.linspace(0, 10, num=11)

# PMF for all these values:
fx = stats.binom.pmf(x, 10, 0.2)

# collect values in DataFrame:
result = pd.DataFrame({'x': x, 'fx': fx})
print(f'result: \n{result}\n')

# plot:
plt.bar(x, fx, color='0.6')
plt.ylabel('x')
plt.ylabel('fx')
plt.savefig('PyGraphs/PMF-example.pdf')
```

Script 1.31: `PDF-example.py`

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# support of normal density:
x_range = np.linspace(-4, 4, num=100)

# PDF for all these values:
pdf = stats.norm.pdf(x_range)

# plot:
plt.plot(x_range, pdf, linestyle='-', color='black')
plt.xlabel('x')
plt.ylabel('dx')
plt.savefig('PyGraphs/PDF-example.pdf')
```

Script 1.32: `CDF-example.py`

```
import scipy.stats as stats

# binomial CDF:
p1 = stats.binom.cdf(3, 10, 0.2)
print(f'p1: {p1}\n')

# normal CDF:
```

```
p2 = stats.norm.cdf(1.96) - stats.norm.cdf(-1.96)
print(f'p2: {p2}\n')
```

Script 1.33: `Example-B-6.py`

```
import scipy.stats as stats

# first example using the transformation:
p1_1 = stats.norm.cdf(2 / 3) - stats.norm.cdf(-2 / 3)
print(f'p1_1: {p1_1}\n')

# first example working directly with the distribution of X:
p1_2 = stats.norm.cdf(6, 4, 3) - stats.norm.cdf(2, 4, 3)
print(f'p1_2: {p1_2}\n')

# second example:
p2 = 1 - stats.norm.cdf(2, 4, 3) + stats.norm.cdf(-2, 4, 3)
print(f'p2: {p2}\n')
```

Script 1.34: `CDF-figure.py`

```
import scipy.stats as stats
import numpy as np
import matplotlib.pyplot as plt

# binomial:
# support of binomial PMF:
x_binom = np.linspace(-1, 10, num=1000)

# PMF for all these values:
cdf_binom = stats.binom.cdf(x_binom, 10, 0.2)

# plot:
plt.step(x_binom, cdf_binom, linestyle='-', color='black')
plt.xlabel('x')
plt.ylabel('Fx')
plt.savefig('PyGraphs/CDF-figure-discrete.pdf')
plt.close()

# normal:
# support of normal density:
x_norm = np.linspace(-4, 4, num=1000)

# PDF for all these values:
cdf_norm = stats.norm.cdf(x_norm)

# plot:
plt.plot(x_norm, cdf_norm, linestyle='-', color='black')
plt.xlabel('x')
plt.ylabel('Fx')
plt.savefig('PyGraphs/CDF-figure-cont.pdf')
```

Script 1.35: `Quantile-example.py`

```
import scipy.stats as stats

q_975 = stats.norm.ppf(0.975)
print(f'q_975: {q_975}\n')
```

Script 1.36: `smpl-bernoulli.py`

```
import scipy.stats as stats

sample = stats.bernoulli.rvs(0.5, size=10)
print(f'sample: {sample}\n')
```

Script 1.37: `smpl-norm.py`

```
import scipy.stats as stats

sample = stats.norm.rvs(size=10)
print(f'sample: {sample}\n')
```

Script 1.38: `Random-Numbers.py`

```
import numpy as np
import scipy.stats as stats

# sample from a standard normal RV with sample size n=5:
sample1 = stats.norm.rvs(size=5)
print(f'sample1: {sample1}\n')

# a different sample from the same distribution:
sample2 = stats.norm.rvs(size=5)
print(f'sample2: {sample2}\n')

# set the seed of the random number generator and take two samples:
np.random.seed(6254137)
sample3 = stats.norm.rvs(size=5)
print(f'sample3: {sample3}\n')

sample4 = stats.norm.rvs(size=5)
print(f'sample4: {sample4}\n')

# reset the seed to the same value to get the same samples again:
np.random.seed(6254137)
sample5 = stats.norm.rvs(size=5)
print(f'sample5: {sample5}\n')

sample6 = stats.norm.rvs(size=5)
print(f'sample6: {sample6}\n')
```

Script 1.39: `Example-C-2.py`

```
import numpy as np
import scipy.stats as stats

# manually enter raw data from Wooldridge, Table C.3:
SR87 = np.array([10, 1, 6, .45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
                 .98, 1, .45, 5.03, 8, 9, 18, .28, 7, 3.97])
SR88 = np.array([3, 1, 5, .5, 1.54, 1.5, .8, 2, .67, 1.17, .51,
                 .5, .61, 6.7, 4, 7, 19, .2, 5, 3.83])

# calculate change:
Change = SR88 - SR87

# ingredients to CI formula:
avgCh = np.mean(Change)
print(f'avgCh: {avgCh}\n')
```

```
n = len(Change)
sdCh = np.std(Change, ddof=1)
se = sdCh / np.sqrt(n)
print(f'se: {se}\n')

c = stats.t.ppf(0.975, n - 1)
print(f'c: {c}\n')

# confidence interval:
lowerCI = avgCh - c * se
print(f'lowerCI: {lowerCI}\n')

upperCI = avgCh + c * se
print(f'upperCI: {upperCI}\n')
```

Script 1.40: `Example-C-3.py`

```
import wooldridge as woo
import numpy as np
import scipy.stats as stats

audit = woo.dataWoo('audit')
y = audit['y']

# ingredients to CI formula:
avgy = np.mean(y)
n = len(y)
sdy = np.std(y, ddof=1)
se = sdy / np.sqrt(n)
c95 = stats.norm.ppf(0.975)
c99 = stats.norm.ppf(0.995)

# 95% confidence interval:
lowerCI95 = avgy - c95 * se
print(f'lowerCI95: {lowerCI95}\n')

upperCI95 = avgy + c95 * se
print(f'upperCI95: {upperCI95}\n')

# 99% confidence interval:
lowerCI99 = avgy - c99 * se
print(f'lowerCI99: {lowerCI99}\n')

upperCI99 = avgy + c99 * se
print(f'upperCI99: {upperCI99}\n')
```

Script 1.41: `Critical-Values-t.py`

```
import numpy as np
import pandas as pd
import scipy.stats as stats

# degrees of freedom = n-1:
df = 19

# significance levels:
alpha_one_tailed = np.array([0.1, 0.05, 0.025, 0.01, 0.005, .001])
alpha_two_tailed = alpha_one_tailed * 2

# critical values & table:
```

```
CV = stats.t.ppf(1 - alpha_one_tailed, df)
table = pd.DataFrame({'alpha_one_tailed': alpha_one_tailed,
                      'alpha_two_tailed': alpha_two_tailed, 'CV': CV})
print(f'table: \n{table}\n')
```

Script 1.42: `Example-C-5.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import scipy.stats as stats

audit = woo.dataWoo('audit')
y = audit['y']

# automated calculation of t statistic for H0 (mu=0):
test_auto = stats.ttest_1samp(y, popmean=0)
t_auto = test_auto.statistic  # access test statistic
p_auto = test_auto.pvalue  # access two-sided p value
print(f't_auto: {t_auto}\n')
print(f'p_auto/2: {p_auto / 2}\n')

# manual calculation of t statistic for H0 (mu=0):
avgy = np.mean(y)
n = len(y)
sdy = np.std(y, ddof=1)
se = sdy / np.sqrt(n)
t_manual = avgy / se
print(f't_manual: {t_manual}\n')

# critical values for t distribution with n-1=240 d.f.:
alpha_one_tailed = np.array([0.1, 0.05, 0.025, 0.01, 0.005, .001])
CV = stats.t.ppf(1 - alpha_one_tailed, 240)
table = pd.DataFrame({'alpha_one_tailed': alpha_one_tailed, 'CV': CV})
print(f'table: \n{table}\n')
```

Script 1.43: `Example-C-6.py`

```
import numpy as np
import scipy.stats as stats

# manually enter raw data from Wooldridge, Table C.3:
SR87 = np.array([10, 1, 6, .45, 1.25, 1.3, 1.06, 3, 8.18, 1.67,
                 .98, 1, .45, 5.03, 8, 9, 18, .28, 7, 3.97])
SR88 = np.array([3, 1, 5, .5, 1.54, 1.5, .8, 2, .67, 1.17, .51,
                 .5, .61, 6.7, 4, 7, 19, .2, 5, 3.83])
Change = SR88 - SR87

# automated calculation of t statistic for H0 (mu=0):
test_auto = stats.ttest_1samp(Change, popmean=0)
t_auto = test_auto.statistic
p_auto = test_auto.pvalue
print(f't_auto: {t_auto}\n')
print(f'p_auto/2: {p_auto / 2}\n')

# manual calculation of t statistic for H0 (mu=0):
avgCh = np.mean(Change)
n = len(Change)
sdCh = np.std(Change, ddof=1)
se = sdCh / np.sqrt(n)
```

```
t_manual = avgCh / se
print(f't_manual: {t_manual}\n')

# manual calculation of p value for H0 (mu=0):
p_manual = stats.t.cdf(t_manual, n - 1)
print(f'p_manual: {p_manual}\n')
```

Script 1.44: Example-C-7.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import scipy.stats as stats

audit = woo.dataWoo('audit')
y = audit['y']

# automated calculation of t statistic for H0 (mu=0):
test_auto = stats.ttest_1samp(y, popmean=0)
t_auto = test_auto.statistic
p_auto = test_auto.pvalue
print(f't_auto: {t_auto}\n')
print(f'p_auto/2: {p_auto/2}\n')

# manual calculation of t statistic for H0 (mu=0):
avgy = np.mean(y)
n = len(y)
sdy = np.std(y, ddof=1)
se = sdy / np.sqrt(n)
t_manual = avgy / se
print(f't_manual: {t_manual}\n')

# manual calculation of p value for H0 (mu=0):
p_manual = stats.t.cdf(t_manual, n - 1)
print(f'p_manual: {p_manual}\n')
```

Script 1.45: Adv-Loops.py

```
seq = [1, 2, 3, 4, 5, 6]
for i in seq:
    if i < 4:
        print(i ** 3)
    else:
        print(i ** 2)
```

Script 1.46: Adv-Loops2.py

```
seq = [1, 2, 3, 4, 5, 6]
for i in range(len(seq)):
    if seq[i] < 4:
        print(seq[i] ** 3)
    else:
        print(seq[i] ** 2)
```

Script 1.47: Adv-Functions.py

```
# define function:
def mysqrt(x):
    if x >= 0:
        result = x ** 0.5
    else:
```

```
        result = 'You fool!'
    return result


# call function and save result:
result1 = mysqrt(4)
print(f'result1: {result1}\n')

result2 = mysqrt(-1.5)
print(f'result2: {result2}\n')
```

Script 1.48: **Adv-ObjOr.py**

```
# use the predefined class 'list' to create an object:
a = [2, 6, 3, 6]

# access a local variable (to find out what kind of object we are dealing with):
check = type(a).__name__
print(f'check: {check}\n')

# make use of a method (how many 6 are in a?):
count_six = a.count(6)
print(f'count_six: {count_six}\n')

# use another method (sort data in a):
a.sort()
print(f'a: {a}\n')
```

Script 1.49: **Adv-ObjOr2.py**

```
import numpy as np

# multiply these two matrices:
a = np.array([[3, 6, 1], [2, 7, 4]])
b = np.array([[1, 8, 6], [3, 5, 8], [1, 1, 2]])

# the numpy way:
result_np = a.dot(b)
print(f'result_np: \n{result_np}\n')

# or, do it yourself by defining a class:
class myMatrices:
    def __init__(self, A, B):
        self.A = A
        self.B = B

    def mult(self):
        N = self.A.shape[0]  # number of rows in A
        K = self.B.shape[1]  # number of cols in B
        out = np.empty((N, K))  # initialize output
        for i in range(N):
            for j in range(K):
                out[i, j] = sum(self.A[i, :] * self.B[:, j])
        return out


# create an object:
test = myMatrices(a, b)

# access local variables:
```

```
print(f'test.A: \n{test.A}\n')
print(f'test.B: \n{test.B}\n')

# use object method:
result_own = test.mult()
print(f'result_own: \n{result_own}\n')
```

Script 1.50: `Adv-ObjOr3.py`

```
import numpy as np

# multiply these two matrices:
a = np.array([[3, 6, 1], [2, 7, 4]])
b = np.array([[1, 8, 6], [3, 5, 8], [1, 1, 2]])


# define your own class:
class myMatrices:
    def __init__(self, A, B):
        self.A = A
        self.B = B

    def mult(self):
        N = self.A.shape[0]  # number of rows in A
        K = self.B.shape[1]  # number of cols in B
        out = np.empty((N, K))  # initialize output
        for i in range(N):
            for j in range(K):
                out[i, j] = sum(self.A[i, :] * self.B[:, j])
        return out


# define a subclass:
class myMatNew(myMatrices):
    def getTotalElem(self):
        N = self.A.shape[0]  # number of rows in A
        K = self.B.shape[1]  # number of cols in B
        return N * K


# create an object of the subclass:
test = myMatNew(a, b)

# use a method of myMatrices:
result_own = test.mult()
print(f'result_own: \n{result_own}\n')

# use a method of myMatNew:
totalElem = test.getTotalElem()
print(f'totalElem: {totalElem}\n')
```

Script 1.51: `Simulate-Estimate.py`

```
import numpy as np
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# set sample size:
```

```
n = 100

# draw a sample given the population parameters:
sample1 = stats.norm.rvs(10, 2, size=n)

# estimate the population mean with the sample average:
estimate1 = np.mean(sample1)
print(f'estimate1: {estimate1}\n')

# draw a different sample and estimate again:
sample2 = stats.norm.rvs(10, 2, size=n)
estimate2 = np.mean(sample2)
print(f'estimate2: {estimate2}\n')

# draw a third sample and estimate again:
sample3 = stats.norm.rvs(10, 2, size=n)
estimate3 = np.mean(sample3)
print(f'estimate3: {estimate3}\n')
```

Script 1.52: `Simulation-Repeated.py`

```
import numpy as np
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# set sample size:
n = 100

# initialize ybar to an array of length r=10000 to later store results:
r = 10000
ybar = np.empty(r)

# repeat r times:
for j in range(r):
    # draw a sample and store the sample mean in pos. j=0,1,... of ybar:
    sample = stats.norm.rvs(10, 2, size=n)
    ybar[j] = np.mean(sample)
```

Script 1.53: `Simulation-Repeated-Results.py`

```
import numpy as np
import statsmodels.api as sm
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(123456)

# set sample size:
n = 100

# initialize ybar to an array of length r=10000 to later store results:
r = 10000
ybar = np.empty(r)

# repeat r times:
for j in range(r):
    # draw a sample and store the sample mean in pos. j=0,1,... of ybar:
```

```
    sample = stats.norm.rvs(10, 2, size=n)
    ybar[j] = np.mean(sample)

# the first 20 of 10000 estimates:
print(f'ybar[0:19]: \n{ybar[0:19]}\n')

# simulated mean:
print(f'np.mean(ybar): {np.mean(ybar)}\n')

# simulated variance:
print(f'np.var(ybar, ddof=1): {np.var(ybar, ddof=1)}\n')

# simulated density:
kde = sm.nonparametric.KDEUnivariate(ybar)
kde.fit()

# normal density:
x_range = np.linspace(9, 11)
y = stats.norm.pdf(x_range, 10, np.sqrt(0.04))

# create graph:
plt.plot(kde.support, kde.density, color='black', label='ybar')
plt.plot(x_range, y, linestyle='--', color='black', label='normal distribution')
plt.ylabel('density')
plt.xlabel('ybar')
plt.legend()
plt.savefig('PyGraphs/Simulation-Repeated-Results.pdf')
```

Script 1.54: `Simulation-Inference-Figure.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(123456)

# set sample size and MC simulations:
r = 10000
n = 100

# initialize arrays to later store results:
CIlower = np.empty(r)
CIupper = np.empty(r)
pvalue1 = np.empty(r)
pvalue2 = np.empty(r)

# repeat r times:
for j in range(r):
    # draw a sample:
    sample = stats.norm.rvs(10, 2, size=n)
    sample_mean = np.mean(sample)
    sample_sd = np.std(sample, ddof=1)
    # test the (correct) null hypothesis mu=10:
    testres1 = stats.ttest_1samp(sample, popmean=10)
    pvalue1[j] = testres1.pvalue
    cv = stats.t.ppf(0.975, df=n - 1)
    CIlower[j] = sample_mean - cv * sample_sd / np.sqrt(n)
    CIupper[j] = sample_mean + cv * sample_sd / np.sqrt(n)
    # test the (incorrect) null hypothesis mu=9.5 & store the p value:
```

```
    testres2 = stats.ttest_1samp(sample, popmean=9.5)
    pvalue2[j] = testres2.pvalue

##################
## correct H0   ##
##################

plt.figure(figsize=(3, 5))  # set figure ratio
plt.ylim(0, 101)
plt.xlim(9, 11)
for j in range(1, 101):
    if 10 > CIlower[j] and 10 < CIupper[j]:
        plt.plot([CIlower[j], CIupper[j]], [j, j], linestyle='-', color='grey')
    else:
        plt.plot([CIlower[j], CIupper[j]], [j, j], linestyle='-', color='black')
plt.axvline(10, linestyle='--', color='black', linewidth=0.5)
plt.ylabel('Sample No.')
plt.savefig('PyGraphs/Simulation-Inference-Figure1.pdf')

##################
## incorrect H0 ##
##################

plt.figure(figsize=(3, 5))  # set figure ratio
plt.ylim(0, 101)
plt.xlim(9, 11)
for j in range(1, 101):
    if 9.5 > CIlower[j] and 9.5 < CIupper[j]:
        plt.plot([CIlower[j], CIupper[j]], [j, j], linestyle='-', color='grey')
    else:
        plt.plot([CIlower[j], CIupper[j]], [j, j], linestyle='-', color='black')
plt.axvline(9.5, linestyle='--', color='black', linewidth=0.5)
plt.ylabel('Sample No.')
plt.savefig('PyGraphs/Simulation-Inference-Figure2.pdf')
```

Script 1.55: Simulation-Inference.py

```
import numpy as np
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# set sample size and MC simulations:
r = 10000
n = 100

# initialize arrays to later store results:
CIlower = np.empty(r)
CIupper = np.empty(r)
pvalue1 = np.empty(r)
pvalue2 = np.empty(r)

# repeat r times:
for j in range(r):
    # draw a sample:
    sample = stats.norm.rvs(10, 2, size=n)
    sample_mean = np.mean(sample)
    sample_sd = np.std(sample, ddof=1)
```

```
    # test the (correct) null hypothesis mu=10:
    testres1 = stats.ttest_1samp(sample, popmean=10)
    pvalue1[j] = testres1.pvalue
    cv = stats.t.ppf(0.975, df=n - 1)
    CIlower[j] = sample_mean - cv * sample_sd / np.sqrt(n)
    CIupper[j] = sample_mean + cv * sample_sd / np.sqrt(n)

    # test the (incorrect) null hypothesis mu=9.5 & store the p value:
    testres2 = stats.ttest_1samp(sample, popmean=9.5)
    pvalue2[j] = testres2.pvalue

# test results as logical value:
reject1 = pvalue1 <= 0.05
count1_true = np.count_nonzero(reject1)  # counts true
count1_false = r - count1_true
print(f'count1_true: {count1_true}\n')
print(f'count1_false: {count1_false}\n')

reject2 = pvalue2 <= 0.05
count2_true = np.count_nonzero(reject2)
count2_false = r - count2_true
print(f'count2_true: {count2_true}\n')
print(f'count2_false: {count2_false}\n')
```

2. Scripts Used in Chapter 02

Script 2.1: **Example-2-3.py**

```
import wooldridge as woo
import numpy as np

ceosal1 = woo.dataWoo('ceosal1')
x = ceosal1['roe']
y = ceosal1['salary']

# ingredients to the OLS formulas:
cov_xy = np.cov(x, y)[1, 0]  # access 2. row and 1. column of covariance matrix
var_x = np.var(x, ddof=1)
x_bar = np.mean(x)
y_bar = np.mean(y)

# manual calculation of OLS coefficients:
b1 = cov_xy / var_x
b0 = y_bar - b1 * x_bar
print(f'b1: {b1}\n')
print(f'b0: {b0}\n')
```

Script 2.2: **Example-2-3-2.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Script 2.3: **Example-2-3-3.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# OLS regression:
reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()

# scatter plot and fitted values:
plt.plot('roe', 'salary', data=ceosal1, color='grey', marker='o', linestyle='')
plt.plot(ceosal1['roe'], results.fittedvalues, color='black', linestyle='-')
plt.ylabel('salary')
plt.xlabel('roe')
plt.savefig('PyGraphs/Example-2-3-3.pdf')
```

Script 2.4: **Example-2-4.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='wage ~ educ', data=wage1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Script 2.5: **Example-2-5.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

vote1 = woo.dataWoo('vote1')

# OLS regression:
reg = smf.ols(formula='voteA ~ shareA', data=vote1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')

# scatter plot and fitted values:
plt.plot('shareA', 'voteA', data=vote1, color='grey', marker='o', linestyle='')
plt.plot(vote1['shareA'], results.fittedvalues, color='black', linestyle='-')
plt.ylabel('voteA')
plt.xlabel('shareA')
plt.savefig('PyGraphs/Example-2-5.pdf')
```

Script 2.6: **Example-2-6.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

# OLS regression:
```

```
reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()

# obtain predicted values and residuals:
salary_hat = results.fittedvalues
u_hat = results.resid

# Wooldridge, Table 2.2:
table = pd.DataFrame({'roe': ceosal1['roe'],
                      'salary': ceosal1['salary'],
                      'salary_hat': salary_hat,
                      'u_hat': u_hat})
print(f'table.head(15): \n{table.head(15)}\n')
```

Script 2.7: **Example-2-7.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')
reg = smf.ols(formula='wage ~ educ', data=wage1)
results = reg.fit()

# obtain coefficients, predicted values and residuals:
b = results.params
wage_hat = results.fittedvalues
u_hat = results.resid

# confirm property (1):
u_hat_mean = np.mean(u_hat)
print(f'u_hat_mean: {u_hat_mean}\n')

# confirm property (2):
educ_u_cov = np.cov(wage1['educ'], u_hat)[1, 0]
print(f'educ_u_cov: {educ_u_cov}\n')

# confirm property (3):
educ_mean = np.mean(wage1['educ'])
wage_pred = b[0] + b[1] * educ_mean
print(f'wage_pred: {wage_pred}\n')

wage_mean = np.mean(wage1['wage'])
print(f'wage_mean: {wage_mean}\n')
```

Script 2.8: **Example-2-8.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

# OLS regression:
reg = smf.ols(formula='salary ~ roe', data=ceosal1)
results = reg.fit()

# calculate predicted values & residuals:
sal_hat = results.fittedvalues
u_hat = results.resid
```

```
# calculate R^2 in three different ways:
sal = ceosal1['salary']
R2_a = np.var(sal_hat, ddof=1) / np.var(sal, ddof=1)
R2_b = 1 - np.var(u_hat, ddof=1) / np.var(sal, ddof=1)
R2_c = np.corrcoef(sal, sal_hat)[1, 0] ** 2

print(f'R2_a: {R2_a}\n')
print(f'R2_b: {R2_b}\n')
print(f'R2_c: {R2_c}\n')
```

Script 2.9: `Example-2-9.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

vote1 = woo.dataWoo('vote1')

# OLS regression:
reg = smf.ols(formula='voteA ~ shareA', data=vote1)
results = reg.fit()

# print results using summary:
print(f'results.summary(): \n{results.summary()}\n')

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 2.10: `Example-2-10.py`

```
import numpy as np
import wooldridge as woo
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

# estimate log-level model:
reg = smf.ols(formula='np.log(wage) ~ educ', data=wage1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Script 2.11: `Example-2-11.py`

```
import numpy as np
import wooldridge as woo
import statsmodels.formula.api as smf

ceosal1 = woo.dataWoo('ceosal1')

# estimate log-log model:
reg = smf.ols(formula='np.log(salary) ~ np.log(sales)', data=ceosal1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

Script 2.12: **SLR-Origin-Const.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

ceosal1 = woo.dataWoo('ceosal1')

# usual OLS regression:
reg1 = smf.ols(formula='salary ~ roe', data=ceosal1)
results1 = reg1.fit()
b_1 = results1.params
print(f'b_1: \n{b_1}\n')

# regression without intercept (through origin):
reg2 = smf.ols(formula='salary ~ 0 + roe', data=ceosal1)
results2 = reg2.fit()
b_2 = results2.params
print(f'b_2: \n{b_2}\n')

# regression without slope (on a constant):
reg3 = smf.ols(formula='salary ~ 1', data=ceosal1)
results3 = reg3.fit()
b_3 = results3.params
print(f'b_3: \n{b_3}\n')

# average y:
sal_mean = np.mean(ceosal1['salary'])
print(f'sal_mean: {sal_mean}\n')

# scatter plot and fitted values:
plt.plot('roe', 'salary', data=ceosal1, color='grey', marker='o',
         linestyle='', label='')
plt.plot(ceosal1['roe'], results1.fittedvalues, color='black',
         linestyle='-', label='full')
plt.plot(ceosal1['roe'], results2.fittedvalues, color='black',
         linestyle=':', label='through origin')
plt.plot(ceosal1['roe'], results3.fittedvalues, color='black',
         linestyle='-.', label='const only')
plt.ylabel('salary')
plt.xlabel('roe')
plt.legend()
plt.savefig('PyGraphs/SLR-Origin-Const.pdf')
```

Script 2.13: **Example-2-12.py**

```
import numpy as np
import wooldridge as woo
import statsmodels.formula.api as smf

meap93 = woo.dataWoo('meap93')

# estimate the model and save the results as "results":
reg = smf.ols(formula='math10 ~ lnchprg', data=meap93)
results = reg.fit()

# number of obs.:
n = results.nobs

# SER:
```

```
u_hat_var = np.var(results.resid, ddof=1)
SER = np.sqrt(u_hat_var) * np.sqrt((n - 1) / (n - 2))
print(f'SER: {SER}\n')

# SE of b0 & b1, respectively:
lnchprg_sq_mean = np.mean(meap93['lnchprg'] ** 2)
lnchprg_var = np.var(meap93['lnchprg'], ddof=1)
b1_se = SER / (np.sqrt(lnchprg_var)
               * np.sqrt(n - 1)) * np.sqrt(lnchprg_sq_mean)
b0_se = SER / (np.sqrt(lnchprg_var) * np.sqrt(n - 1))
print(f'b1_se: {b1_se}\n')
print(f'b0_se: {b0_se}\n')

# automatic calculations:
print(f'results.summary(): \n{results.summary()}\n')
```

Script 2.14: `SLR-Sim-Sample.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# set sample size:
n = 1000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# draw a sample of size n:
x = stats.norm.rvs(4, 1, size=n)
u = stats.norm.rvs(0, su, size=n)
y = beta0 + beta1 * x + u
df = pd.DataFrame({'y': y, 'x': x})

# estimate parameters by OLS:
reg = smf.ols(formula='y ~ x', data=df)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')

# features of the sample for the variance formula:
x_sq_mean = np.mean(x ** 2)
print(f'x_sq_mean: {x_sq_mean}\n')
x_var = np.sum((x - np.mean(x)) ** 2)
print(f'x_var: {x_var}\n')

# graph:
x_range = np.linspace(0, 8, num=100)
plt.ylim([-2, 10])
plt.plot(x, y, color='lightgrey', marker='o', linestyle='')
plt.plot(x_range, beta0 + beta1 * x_range, color='black',
         linestyle='-', linewidth=2, label='pop. regr. fct.')
plt.plot(x_range, b[0] + b[1] * x_range, color='grey',
```

```
        linestyle='-', linewidth=2, label='OLS regr. fct.')
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/SLR-Sim-Sample.pdf')
```

Script 2.15: `SLR-Sim-Model.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations
n = 1000
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
su = 2
sx = 1
ex = 4

# initialize b0 and b1 to store results later:
b0 = np.empty(r)
b1 = np.empty(r)

# repeat r times:
for i in range(r):
    # draw a sample:
    x = stats.norm.rvs(ex, sx, size=n)
    u = stats.norm.rvs(0, su, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b0[i] = results.params['Intercept']
    b1[i] = results.params['x']
```

Script 2.16: `SLR-Sim-Model-Condx.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000
```

```
# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
b0 = np.empty(r)
b1 = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of y:
    u = stats.norm.rvs(0, su, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate and store parameters by OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b0[i] = results.params['Intercept']
    b1[i] = results.params['x']

# MC estimate of the expected values:
b0_mean = np.mean(b0)
b1_mean = np.mean(b1)

print(f'b0_mean: {b0_mean}\n')
print(f'b1_mean: {b1_mean}\n')

# MC estimate of the variances:
b0_var = np.var(b0, ddof=1)
b1_var = np.var(b1, ddof=1)

print(f'b0_var: {b0_var}\n')
print(f'b1_var: {b1_var}\n')

# graph:
x_range = np.linspace(0, 8, num=100)
plt.ylim([0, 6])

# add population regression line:
plt.plot(x_range, beta0 + beta1 * x_range, color='black',
         linestyle='-', linewidth=2, label='Population')

# add first OLS regression line (to attach a label):
plt.plot(x_range, b0[0] + b1[0] * x_range, color='grey',
         linestyle='-', linewidth=0.5, label='OLS regressions')

# add OLS regression lines no. 2 to 10:
for i in range(1, 10):
    plt.plot(x_range, b0[i] + b1[i] * x_range, color='grey',
             linestyle='-', linewidth=0.5)
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
```

```
plt.savefig('PyGraphs/SLR-Sim-Model-Condx.pdf')
```

Script 2.17: `SLR-Sim-Model-ViolSLR4.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas and sd of u):
beta0 = 1
beta1 = 0.5
su = 2

# initialize b0 and b1 to store results later:
b0 = np.empty(r)
b1 = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of y:
    u_mean = np.array((x - 4) / 5)
    u = stats.norm.rvs(u_mean, su, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate and store parameters by OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b0[i] = results.params['Intercept']
    b1[i] = results.params['x']

# MC estimate of the expected values:
b0_mean = np.mean(b0)
b1_mean = np.mean(b1)

print(f'b0_mean: {b0_mean}\n')
print(f'b1_mean: {b1_mean}\n')

# MC estimate of the variances:
b0_var = np.var(b0, ddof=1)
b1_var = np.var(b1, ddof=1)

print(f'b0_var: {b0_var}\n')
print(f'b1_var: {b1_var}\n')
```

Script 2.18: `SLR-Sim-Model-ViolSLR5.py`

```
import numpy as np
import pandas as pd
```

```
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize b0 and b1 to store results later:
b0 = np.empty(r)
b1 = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of y:
    u_var = np.array(4 / np.exp(4.5) * np.exp(x))
    u = stats.norm.rvs(0, np.sqrt(u_var), size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate and store parameters by OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    results = reg.fit()
    b0[i] = results.params['Intercept']
    b1[i] = results.params['x']

# MC estimate of the expected values:
b0_mean = np.mean(b0)
b1_mean = np.mean(b1)

print(f'b0_mean: {b0_mean}\n')
print(f'b1_mean: {b1_mean}\n')

# MC estimate of the variances:
b0_var = np.var(b0, ddof=1)
b1_var = np.var(b1, ddof=1)

print(f'b0_var: {b0_var}\n')
print(f'b1_var: {b1_var}\n')
```

3. Scripts Used in Chapter 03

Script 3.1: **Example-3-1.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf
```

```
gpa1 = woo.dataWoo('gpa1')

reg = smf.ols(formula='colGPA ~ hsGPA + ACT', data=gpa1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 3.2: **Example-3-2.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ educ + exper + tenure', data=wage1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 3.3: **Example-3-3.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

k401k = woo.dataWoo('401k')

reg = smf.ols(formula='prate ~ mrate + age', data=k401k)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 3.4: **Example-3-5a.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf

crime1 = woo.dataWoo('crime1')

# model without avgsen:
reg = smf.ols(formula='narr86 ~ pcnv + ptime86 + qemp86', data=crime1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 3.5: **Example-3-5b.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf

crime1 = woo.dataWoo('crime1')

# model with avgsen:
reg = smf.ols(formula='narr86 ~ pcnv + avgsen + ptime86 + qemp86', data=crime1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 3.6: **Example-3-6.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')
```

```
reg = smf.ols(formula='np.log(wage) ~ educ', data=wage1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 3.7: OLS-Matrices.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import patsy as pt

gpa1 = woo.dataWoo('gpa1')

# determine sample size & no. of regressors:
n = len(gpa1)
k = 2

# extract y:
y = gpa1['colGPA']

# extract X & add a column of ones:
X = pd.DataFrame({'const': 1, 'hsGPA': gpa1['hsGPA'], 'ACT': gpa1['ACT']})

# alternative with patsy:
y2, X2 = pt.dmatrices('colGPA ~ hsGPA + ACT', data=gpa1, return_type='dataframe')

# display first rows of X:
print(f'X.head(): \n{X.head()}\n')

# parameter estimates:
X = np.array(X)
y = np.array(y).reshape(n, 1)  # creates a row vector
b = np.linalg.inv(X.T @ X) @ X.T @ y
print(f'b: \n{b}\n')

# residuals, estimated variance of u and SER:
u_hat = y - X @ b
sigsq_hat = (u_hat.T @ u_hat) / (n - k - 1)
SER = np.sqrt(sigsq_hat)
print(f'SER: {SER}\n')

# estimated variance of the parameter estimators and SE:
Vbeta_hat = sigsq_hat * np.linalg.inv(X.T @ X)
se = np.sqrt(np.diagonal(Vbeta_hat))
print(f'se: {se}\n')
```

Script 3.8: Omitted-Vars.py

```
import wooldridge as woo
import statsmodels.formula.api as smf

gpa1 = woo.dataWoo('gpa1')

# parameter estimates for full and simple model:
reg = smf.ols(formula='colGPA ~ ACT + hsGPA', data=gpa1)
results = reg.fit()
b = results.params
print(f'b: \n{b}\n')
```

```
# relation between regressors:
reg_delta = smf.ols(formula='hsGPA ~ ACT', data=gpa1)
results_delta = reg_delta.fit()
delta_tilde = results_delta.params
print(f'delta_tilde: \n{delta_tilde}\n')

# omitted variables formula for b1_tilde:
b1_tilde = b['ACT'] + b['hsGPA'] * delta_tilde['ACT']
print(f'b1_tilde:  \n{b1_tilde}\n')

# actual regression with hsGPA omitted:
reg_om = smf.ols(formula='colGPA ~ ACT', data=gpa1)
results_om = reg_om.fit()
b_om = results_om.params
print(f'b_om: \n{b_om}\n')
```

Script 3.9: `MLR-SE.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

gpa1 = woo.dataWoo('gpa1')

# full estimation results including automatic SE:
reg = smf.ols(formula='colGPA ~ hsGPA + ACT', data=gpa1)
results = reg.fit()

# extract SER (instead of calculation via residuals):
SER = np.sqrt(results.mse_resid)

# regressing hsGPA on ACT for calculation of R2 & VIF:
reg_hsGPA = smf.ols(formula='hsGPA ~ ACT', data=gpa1)
results_hsGPA = reg_hsGPA.fit()
R2_hsGPA = results_hsGPA.rsquared
VIF_hsGPA = 1 / (1 - R2_hsGPA)
print(f'VIF_hsGPA: {VIF_hsGPA}\n')

# manual calculation of SE of hsGPA coefficient:
n = results.nobs
sdx = np.std(gpa1['hsGPA'], ddof=1) * np.sqrt((n - 1) / n)
SE_hsGPA = 1 / np.sqrt(n) * SER / sdx * np.sqrt(VIF_hsGPA)
print(f'SE_hsGPA: {SE_hsGPA}\n')
```

Script 3.10: `MLR-VIF.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.stats.outliers_influence as smo
import patsy as pt

wage1 = woo.dataWoo('wage1')

# extract matrices using patsy:
y, X = pt.dmatrices('np.log(wage) ~ educ + exper + tenure',
                    data=wage1, return_type='dataframe')

# get VIF:
K = X.shape[1]
VIF = np.empty(K)
```

```
for i in range(K):
    VIF[i] = smo.variance_inflation_factor(X.values, i)
print(f'VIF: \n{VIF}\n')
```

4. Scripts Used in Chapter 04

Script 4.1: **Example-4-3-cv.py**

```
import scipy.stats as stats
import numpy as np

# CV for alpha=5% and 1% using the t distribution with 137 d.f.:
alpha = np.array([0.05, 0.01])
cv_t = stats.t.ppf(1 - alpha / 2, 137)
print(f'cv_t: {cv_t}\n')

# CV for alpha=5% and 1% using the normal approximation:
cv_n = stats.norm.ppf(1 - alpha / 2)
print(f'cv_n: {cv_n}\n')
```

Script 4.2: **Example-4-3.py**

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

gpa1 = woo.dataWoo('gpa1')

# store and display results:
reg = smf.ols(formula='colGPA ~ hsGPA + ACT + skipped', data=gpa1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')

# manually confirm the formulas, i.e. extract coefficients and SE:
b = results.params
se = results.bse

# reproduce t statistic:
tstat = b / se
print(f'tstat: \n{tstat}\n')

# reproduce p value:
pval = 2 * stats.t.cdf(-abs(tstat), 137)
print(f'pval: \n{pval}\n')
```

Script 4.3: **Example-4-1-cv.py**

```
import scipy.stats as stats
import numpy as np

# CV for alpha=5% and 1% using the t distribution with 522 d.f.:
alpha = np.array([0.05, 0.01])
cv_t = stats.t.ppf(1 - alpha, 522)
print(f'cv_t: {cv_t}\n')

# CV for alpha=5% and 1% using the normal approximation:
cv_n = stats.norm.ppf(1 - alpha)
print(f'cv_n: {cv_n}\n')
```

Script 4.4: **Example-4-1.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ educ + exper + tenure', data=wage1)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')
```

Script 4.5: **Example-4-8.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

rdchem = woo.dataWoo('rdchem')

# OLS regression:
reg = smf.ols(formula='np.log(rd) ~ np.log(sales) + profmarg', data=rdchem)
results = reg.fit()
print(f'results.summary(): \n{results.summary()}\n')

# 95% CI:
CI95 = results.conf_int(0.05)
print(f'CI95: \n{CI95}\n')

# 99% CI:
CI99 = results.conf_int(0.01)
print(f'CI99: \n{CI99}\n')
```

Script 4.6: **F-Test.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf
import scipy.stats as stats

mlb1 = woo.dataWoo('mlb1')
n = mlb1.shape[0]

# unrestricted OLS regression:
reg_ur = smf.ols(
    formula='np.log(salary) ~ years + gamesyr + bavg + hrunsyr + rbisyr',
    data=mlb1)
fit_ur = reg_ur.fit()
r2_ur = fit_ur.rsquared
print(f'r2_ur: {r2_ur}\n')

# restricted OLS regression:
reg_r = smf.ols(formula='np.log(salary) ~ years + gamesyr', data=mlb1)
fit_r = reg_r.fit()
r2_r = fit_r.rsquared
print(f'r2_r: {r2_r}\n')

# F statistic:
fstat = (r2_ur - r2_r) / (1 - r2_ur) * (n - 6) / 3
print(f'fstat: {fstat}\n')
```

```
# CV for alpha=1% using the F distribution with 3 and 347 d.f.:
cv = stats.f.ppf(1 - 0.01, 3, 347)
print(f'cv: {cv}\n')

# p value = 1-cdf of the appropriate F distribution:
fpval = 1 - stats.f.cdf(fstat, 3, 347)
print(f'fpval: {fpval}\n')
```

Script 4.7: `F-Test-Automatic.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

mlb1 = woo.dataWoo('mlb1')

# OLS regression:
reg = smf.ols(
    formula='np.log(salary) ~ years + gamesyr + bavg + hrunsyr + rbisyr',
    data=mlb1)
results = reg.fit()

# automated F test:
hypotheses = ['bavg = 0', 'hrunsyr = 0', 'rbisyr = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Script 4.8: `F-Test-Automatic2.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

mlb1 = woo.dataWoo('mlb1')

# OLS regression:
reg = smf.ols(
    formula='np.log(salary) ~ years + gamesyr + bavg + hrunsyr + rbisyr',
    data=mlb1)
results = reg.fit()

# automated F test:
hypotheses = ['bavg = 0', 'hrunsyr = 2*rbisyr']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

5. Scripts Used in Chapter 05

Script 5.1: `Sim-Asy-OLS-norm.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(ex, sx, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u (std. normal):
    u = stats.norm.rvs(0, 1, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate conditional OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b1[i] = results.params['x']
```

Script 5.2: `Sim-Asy-OLS-chisq.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = np.empty(r)
```

```
# draw a sample of x, fixed over replications:
x = stats.norm.rvs(ex, sx, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u (standardized chi-squared[1]):
    u = (stats.chi2.rvs(1, size=n) - 1) / np.sqrt(2)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate conditional OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b1[i] = results.params['x']
```

Script 5.3: `Sim-Asy-OLS-uncond.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 100
r = 10000

# set true parameters:
beta0 = 1
beta1 = 0.5
sx = 1
ex = 4

# initialize b1 to store results later:
b1 = np.empty(r)

# repeat r times:
for i in range(r):
    # draw a sample of x, varying over replications:
    x = stats.norm.rvs(ex, sx, size=n)

    # draw a sample of u (std. normal):
    u = stats.norm.rvs(0, 1, size=n)
    y = beta0 + beta1 * x + u
    df = pd.DataFrame({'y': y, 'x': x})

    # estimate unconditional OLS:
    reg = smf.ols(formula='y ~ x', data=df)
    results = reg.fit()
    b1[i] = results.params['x']
```

Script 5.4: `Example-5-3.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

crime1 = woo.dataWoo('crime1')
```

```
# 1. estimate restricted model:
reg_r = smf.ols(formula='narr86 ~ pcnv + ptime86 + qemp86', data=crime1)
fit_r = reg_r.fit()
r2_r = fit_r.rsquared
print(f'r2_r: {r2_r}\n')

# 2. regression of residuals from restricted model:
crime1['utilde'] = fit_r.resid
reg_LM = smf.ols(formula='utilde ~ pcnv + ptime86 + qemp86 + avgsen + tottime',
                 data=crime1)
fit_LM = reg_LM.fit()
r2_LM = fit_LM.rsquared
print(f'r2_LM: {r2_LM}\n')

# 3. calculation of LM test statistic:
LM = r2_LM * fit_LM.nobs
print(f'LM: {LM}\n')

# 4. critical value from chi-squared distribution, alpha=10%:
cv = stats.chi2.ppf(1 - 0.10, 2)
print(f'cv: {cv}\n')

# 5. p value (alternative to critical value):
pval = 1 - stats.chi2.cdf(LM, 2)
print(f'pval: {pval}\n')

# 6. compare to F-test:
reg = smf.ols(formula='narr86 ~ pcnv + ptime86 + qemp86 + avgsen + tottime',
              data=crime1)
results = reg.fit()
hypotheses = ['avgsen = 0', 'tottime = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue
print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

6. Scripts Used in Chapter 06

Script 6.1: `Data-Scaling.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

bwght = woo.dataWoo('bwght')

# regress and report coefficients:
reg = smf.ols(formula='bwght ~ cigs + faminc', data=bwght)
results = reg.fit()

# weight in pounds, manual way:
bwght['bwght_lbs'] = bwght['bwght'] / 16
reg_lbs = smf.ols(formula='bwght_lbs ~ cigs + faminc', data=bwght)
results_lbs = reg_lbs.fit()
```

```
# weight in pounds, direct way:
reg_lbs2 = smf.ols(formula='I(bwght/16) ~ cigs + faminc', data=bwght)
results_lbs2 = reg_lbs2.fit()

# packs of cigarettes:
reg_packs = smf.ols(formula='bwght ~ I(cigs/20) + faminc', data=bwght)
results_packs = reg_packs.fit()

# compare results:
table = pd.DataFrame({'b': round(results.params, 4),
                      'b_lbs': round(results_lbs.params, 4),
                      'b_lbs2': round(results_lbs2.params, 4),
                      'b_packs': round(results_packs.params, 4)})
print(f'table: \n{table}\n')
```

Script 6.2: `Example-6-1.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf


# define a function for the standardization:
def scale(x):
    x_mean = np.mean(x)
    x_var = np.var(x, ddof=1)
    x_scaled = (x - x_mean) / np.sqrt(x_var)
    return x_scaled


# standardize and estimate:
hprice2 = woo.dataWoo('hprice2')
hprice2['price_sc'] = scale(hprice2['price'])
hprice2['nox_sc'] = scale(hprice2['nox'])
hprice2['crime_sc'] = scale(hprice2['crime'])
hprice2['rooms_sc'] = scale(hprice2['rooms'])
hprice2['dist_sc'] = scale(hprice2['dist'])
hprice2['stratio_sc'] = scale(hprice2['stratio'])

reg = smf.ols(
    formula='price_sc ~ 0 + nox_sc + crime_sc + rooms_sc + dist_sc + stratio_sc',
    data=hprice2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 6.3: `Formula-Logarithm.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

hprice2 = woo.dataWoo('hprice2')
```

```
reg = smf.ols(formula='np.log(price) ~ np.log(nox) + rooms', data=hprice2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 6.4: **Example-6-2.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

hprice2 = woo.dataWoo('hprice2')

reg = smf.ols(
    formula='np.log(price) ~ np.log(nox)+np.log(dist)+rooms+I(rooms**2)+stratio',
    data=hprice2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 6.5: **Example-6-2-Ftest.py**

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

hprice2 = woo.dataWoo('hprice2')
n = hprice2.shape[0]

reg = smf.ols(
    formula='np.log(price) ~ np.log(nox)+np.log(dist)+rooms+I(rooms**2)+stratio',
    data=hprice2)
results = reg.fit()

# implemented F test for rooms:
hypotheses = ['rooms = 0', 'I(rooms ** 2) = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Script 6.6: **Example-6-3.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
```

```
import statsmodels.formula.api as smf

attend = woo.dataWoo('attend')
n = attend.shape[0]

reg = smf.ols(formula='stndfnl ~ atndrte*priGPA + ACT + I(priGPA**2) + I(ACT**2)',
              data=attend)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# estimate for partial effect at priGPA=2.59:
b = results.params
partial_effect = b['atndrte'] + 2.59 * b['atndrte:priGPA']
print(f'partial_effect: {partial_effect}\n')

# F test for partial effect at priGPA=2.59:
hypotheses = 'atndrte + 2.59 * atndrte:priGPA = 0'
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Script 6.7: `Predictions.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import pandas as pd

gpa2 = woo.dataWoo('gpa2')

reg = smf.ols(formula='colgpa ~ sat + hsperc + hsize + I(hsize**2)', data=gpa2)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# generate data set containing the regressor values for predictions:
cvalues1 = pd.DataFrame({'sat': [1200], 'hsperc': [30],
                        'hsize': [5]}, index=['newPerson1'])
print(f'cvalues1: \n{cvalues1}\n')

# point estimate of prediction (cvalues1):
colgpa_pred1 = results.predict(cvalues1)
print(f'colgpa_pred1: \n{colgpa_pred1}\n')

# define three sets of regressor variables:
cvalues2 = pd.DataFrame({'sat': [1200, 900, 1400, ],
                        'hsperc': [30, 20, 5], 'hsize': [5, 3, 1]},
```

```
                       index=['newPerson1', 'newPerson2', 'newPerson3'])
print(f'cvalues2: \n{cvalues2}\n')

# point estimate of prediction (cvalues2):
colgpa_pred2 = results.predict(cvalues2)
print(f'colgpa_pred2: \n{colgpa_pred2}\n')
```

Script 6.8: `Example-6-5.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import pandas as pd

gpa2 = woo.dataWoo('gpa2')

reg = smf.ols(formula='colgpa ~ sat + hsperc + hsize + I(hsize**2)', data=gpa2)
results = reg.fit()

# define three sets of regressor variables:
cvalues2 = pd.DataFrame({'sat': [1200, 900, 1400, ],
                        'hsperc': [30, 20, 5], 'hsize': [5, 3, 1]},
                       index=['newPerson1', 'newPerson2', 'newPerson3'])

# point estimates and 95% confidence and prediction intervals:
colgpa_PICI_95 = results.get_prediction(cvalues2).summary_frame(alpha=0.05)
print(f'colgpa_PICI_95: \n{colgpa_PICI_95}\n')

# point estimates and 99% confidence and prediction intervals:
colgpa_PICI_99 = results.get_prediction(cvalues2).summary_frame(alpha=0.01)
print(f'colgpa_PICI_99: \n{colgpa_PICI_99}\n')
```

Script 6.9: `Effects-Manual.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

hprice2 = woo.dataWoo('hprice2')

# repeating the regression from Example 6.2:
reg = smf.ols(
    formula='np.log(price) ~ np.log(nox)+np.log(dist)+rooms+I(rooms**2)+stratio',
    data=hprice2)
results = reg.fit()

# predictions with rooms = 4-8, all others at the sample mean:
nox_mean = np.mean(hprice2['nox'])
dist_mean = np.mean(hprice2['dist'])
stratio_mean = np.mean(hprice2['stratio'])
X = pd.DataFrame({'rooms': np.linspace(4, 8, num=5),
                  'nox': nox_mean,
                  'dist': dist_mean,
                  'stratio': stratio_mean})
print(f'X: \n{X}\n')

# calculate 95% confidence interval:
lpr_PICI = results.get_prediction(X).summary_frame(alpha=0.05)
lpr_CI = lpr_PICI[['mean', 'mean_ci_lower', 'mean_ci_upper']]
```

```
print(f'lpr_CI: \n{lpr_CI}\n')

# plot:
plt.plot(X['rooms'], lpr_CI['mean'], color='black',
         linestyle='-', label='')
plt.plot(X['rooms'], lpr_CI['mean_ci_upper'], color='lightgrey',
         linestyle='--', label='upper CI')
plt.plot(X['rooms'], lpr_CI['mean_ci_lower'], color='darkgrey',
         linestyle='--', label='lower CI')
plt.ylabel('lprice')
plt.xlabel('rooms')
plt.legend()
plt.savefig('PyGraphs/Effects-Manual.pdf')
```

7. Scripts Used in Chapter 07

Script 7.1: **Example-7-1.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='wage ~ female + educ + exper + tenure', data=wage1)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 7.2: **Example-7-6.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

reg = smf.ols(formula='np.log(wage) ~ married*female + educ + exper +'
                      'I(exper**2) + tenure + I(tenure**2)', data=wage1)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 7.3: **Example-7-1-Boolean.py**

```
import wooldridge as woo
import pandas as pd
```

```
import statsmodels.formula.api as smf

wage1 = woo.dataWoo('wage1')

# regression with boolean variable:
wage1['isfemale'] = (wage1['female'] == 1)
reg = smf.ols(formula='wage ~ isfemale + educ + exper + tenure', data=wage1)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 7.4: `Regr-Categorical.py`

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf

CPS1985 = pd.read_csv('data/CPS1985.csv')
# rename variable to make outputs more compact:
CPS1985['oc'] = CPS1985['occupation']

# table of categories and frequencies for two categorical variables:
freq_gender = pd.crosstab(CPS1985['gender'], columns='count')
print(f'freq_gender: \n{freq_gender}\n')

freq_occupation = pd.crosstab(CPS1985['oc'], columns='count')
print(f'freq_occupation: \n{freq_occupation}\n')

# directly using categorical variables in regression formula:
reg = smf.ols(formula='np.log(wage) ~ education +'
                      'experience + C(gender) + C(oc)', data=CPS1985)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# rerun regression with different reference category:
reg_newref = smf.ols(formula='np.log(wage) ~ education + experience + '
                             'C(gender, Treatment("male")) + '
                             'C(oc, Treatment("technical"))', data=CPS1985)
results_newref = reg_newref.fit()

# print results:
table_newref = pd.DataFrame({'b': round(results_newref.params, 4),
                             'se': round(results_newref.bse, 4),
                             't': round(results_newref.tvalues, 4),
                             'pval': round(results_newref.pvalues, 4)})
print(f'table_newref: \n{table_newref}\n')
```

Script 7.5: **Regr-Categorical-Anova.py**

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

CPS1985 = pd.read_csv('data/CPS1985.csv')

# run regression:
reg = smf.ols(
    formula='np.log(wage) ~ education + experience + gender + occupation',
    data=CPS1985)
results = reg.fit()

# print regression table:
table_reg = pd.DataFrame({'b': round(results.params, 4),
                          'se': round(results.bse, 4),
                          't': round(results.tvalues, 4),
                          'pval': round(results.pvalues, 4)})
print(f'table_reg: \n{table_reg}\n')

# ANOVA table:
table_anova = sm.stats.anova_lm(results, typ=2)
print(f'table_anova: \n{table_anova}\n')
```

Script 7.6: **Example-7-8.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

lawsch85 = woo.dataWoo('lawsch85')

# define cut points for the rank:
cutpts = [0, 10, 25, 40, 60, 100, 175]

# create categorical variable containing ranges for the rank:
lawsch85['rc'] = pd.cut(lawsch85['rank'], bins=cutpts,
                        labels=['(0,10]', '(10,25]', '(25,40]',
                                '(40,60]', '(60,100]', '(100,175]'])

# display frequencies:
freq = pd.crosstab(lawsch85['rc'], columns='count')
print(f'freq: \n{freq}\n')

# run regression:
reg = smf.ols(formula='np.log(salary) ~ C(rc, Treatment("(100,175]")) +'
                      'LSAT + GPA + np.log(libvol) + np.log(cost)',
              data=lawsch85)
results = reg.fit()

# print regression table:
table_reg = pd.DataFrame({'b': round(results.params, 4),
                          'se': round(results.bse, 4),
                          't': round(results.tvalues, 4),
                          'pval': round(results.pvalues, 4)})
print(f'table_reg: \n{table_reg}\n')
```

```
# ANOVA table:
table_anova = sm.stats.anova_lm(results, typ=2)
print(f'table_anova: \n{table_anova}\n')
```

Script 7.7: `Dummy-Interact.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# model with full interactions with female dummy (only for spring data):
reg = smf.ols(formula='cumgpa ~ female * (sat + hsperc + tothrs)',
              data=gpa3, subset=(gpa3['spring'] == 1))
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# F-Test for H0 (the interaction coefficients of 'female' are zero):
hypotheses = ['female = 0', 'female:sat = 0',
              'female:hsperc = 0', 'female:tothrs = 0']
ftest = results.f_test(hypotheses)
fstat = ftest.statistic[0][0]
fpval = ftest.pvalue

print(f'fstat: {fstat}\n')
print(f'fpval: {fpval}\n')
```

Script 7.8: `Dummy-Interact-Sep.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# estimate model for males (& spring data):
reg_m = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs',
                data=gpa3,
                subset=(gpa3['spring'] == 1) & (gpa3['female'] == 0))
results_m = reg_m.fit()

# print regression table:
table_m = pd.DataFrame({'b': round(results_m.params, 4),
                        'se': round(results_m.bse, 4),
                        't': round(results_m.tvalues, 4),
                        'pval': round(results_m.pvalues, 4)})
print(f'table_m: \n{table_m}\n')

# estimate model for females (& spring data):
reg_f = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs',
                data=gpa3,
                subset=(gpa3['spring'] == 1) & (gpa3['female'] == 1))
results_f = reg_f.fit()
```

```
# print regression table:
table_f = pd.DataFrame({'b': round(results_f.params, 4),
                        'se': round(results_f.bse, 4),
                        't': round(results_f.tvalues, 4),
                        'pval': round(results_f.pvalues, 4)})
print(f'table_f: \n{table_f}\n')
```

8. Scripts Used in Chapter 08

Script 8.1: `Example-8-2.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# define regression model:
reg = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs + female + black + white',
              data=gpa3, subset=(gpa3['spring'] == 1))

# estimate default model (only for spring data):
results_default = reg.fit()

table_default = pd.DataFrame({'b': round(results_default.params, 5),
                              'se': round(results_default.bse, 5),
                              't': round(results_default.tvalues, 5),
                              'pval': round(results_default.pvalues, 5)})
print(f'table_default: \n{table_default}\n')

# estimate model with White SE (only for spring data):
results_white = reg.fit(cov_type='HC0')

table_white = pd.DataFrame({'b': round(results_white.params, 5),
                            'se': round(results_white.bse, 5),
                            't': round(results_white.tvalues, 5),
                            'pval': round(results_white.pvalues, 5)})
print(f'table_white: \n{table_white}\n')

# estimate model with refined White SE (only for spring data):
results_refined = reg.fit(cov_type='HC3')

table_refined = pd.DataFrame({'b': round(results_refined.params, 5),
                              'se': round(results_refined.bse, 5),
                              't': round(results_refined.tvalues, 5),
                              'pval': round(results_refined.pvalues, 5)})
print(f'table_refined: \n{table_refined}\n')
```

Script 8.2: `Example-8-2-cont.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

gpa3 = woo.dataWoo('gpa3')

# definition of model and hypotheses:
reg = smf.ols(formula='cumgpa ~ sat + hsperc + tothrs + female + black + white',
```

```
              data=gpa3, subset=(gpa3['spring'] == 1))
hypotheses = ['black = 0', 'white = 0']

# F-Tests using different variance-covariance formulas:
# ususal VCOV:
results_default = reg.fit()
ftest_default = results_default.f_test(hypotheses)
fstat_default = ftest_default.statistic[0][0]
fpval_default = ftest_default.pvalue
print(f'fstat_default: {fstat_default}\n')
print(f'fpval_default: {fpval_default}\n')

# refined White VCOV:
results_hc3 = reg.fit(cov_type='HC3')
ftest_hc3 = results_hc3.f_test(hypotheses)
fstat_hc3 = ftest_hc3.statistic[0][0]
fpval_hc3 = ftest_hc3.pvalue
print(f'fstat_hc3: {fstat_hc3}\n')
print(f'fpval_hc3: {fpval_hc3}\n')

# classical White VCOV:
results_hc0 = reg.fit(cov_type='HC0')
ftest_hc0 = results_hc0.f_test(hypotheses)
fstat_hc0 = ftest_hc0.statistic[0][0]
fpval_hc0 = ftest_hc0.pvalue
print(f'fstat_hc0: {fstat_hc0}\n')
print(f'fpval_hc0: {fpval_hc0}\n')
```

Script 8.3: `Example-8-4.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy as pt

hprice1 = woo.dataWoo('hprice1')

# estimate model:
reg = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results = reg.fit()
table_results = pd.DataFrame({'b': round(results.params, 4),
                              'se': round(results.bse, 4),
                              't': round(results.tvalues, 4),
                              'pval': round(results.pvalues, 4)})
print(f'table_results: \n{table_results}\n')

# automatic BP test (LM version):
y, X = pt.dmatrices('price ~ lotsize + sqrft + bdrms',
                    data=hprice1, return_type='dataframe')
result_bp_lm = sm.stats.diagnostic.het_breuschpagan(results.resid, X)
bp_lm_statistic = result_bp_lm[0]
bp_lm_pval = result_bp_lm[1]
print(f'bp_lm_statistic: {bp_lm_statistic}\n')
print(f'bp_lm_pval: {bp_lm_pval}\n')

# manual BP test (F version):
hprice1['resid_sq'] = results.resid ** 2
reg_resid = smf.ols(formula='resid_sq ~ lotsize + sqrft + bdrms', data=hprice1)
results_resid = reg_resid.fit()
```

```
bp_F_statistic = results_resid.fvalue
bp_F_pval = results_resid.f_pvalue
print(f'bp_F_statistic: {bp_F_statistic}\n')
print(f'bp_F_pval: {bp_F_pval}\n')
```

Script 8.4: **Example-8-5.py**

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy as pt

hprice1 = woo.dataWoo('hprice1')

# estimate model:
reg = smf.ols(formula='np.log(price) ~ np.log(lotsize) + np.log(sqrft) + bdrms',
              data=hprice1)
results = reg.fit()

# BP test:
y, X_bp = pt.dmatrices('np.log(price) ~ np.log(lotsize) + np.log(sqrft) + bdrms',
                       data=hprice1, return_type='dataframe')
result_bp = sm.stats.diagnostic.het_breuschpagan(results.resid, X_bp)
bp_statistic = result_bp[0]
bp_pval = result_bp[1]
print(f'bp_statistic: {bp_statistic}\n')
print(f'bp_pval: {bp_pval}\n')

# White test:
X_wh = pd.DataFrame({'const': 1, 'fitted_reg': results.fittedvalues,
                     'fitted_reg_sq': results.fittedvalues ** 2})
result_white = sm.stats.diagnostic.het_breuschpagan(results.resid, X_wh)
white_statistic = result_white[0]
white_pval = result_white[1]
print(f'white_statistic: {white_statistic}\n')
print(f'white_pval: {white_pval}\n')
```

Script 8.5: **Example-8-6.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

k401ksubs = woo.dataWoo('401ksubs')

# subsetting data:
k401ksubs_sub = k401ksubs[k401ksubs['fsize'] == 1]

# OLS (only for singles, i.e. 'fsize'==1):
reg_ols = smf.ols(formula='nettfa ~ inc + I((age-25)**2) + male + e401k',
                  data=k401ksubs_sub)
results_ols = reg_ols.fit(cov_type='HC0')

# print regression table:
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
```

```
print(f'table_ols: \n{table_ols}\n')

# WLS:
wls_weight = list(1 / k401ksubs_sub['inc'])
reg_wls = smf.wls(formula='nettfa ~ inc + I((age-25)**2) + male + e401k',
                  weights=wls_weight, data=k401ksubs_sub)
results_wls = reg_wls.fit()

# print regression table:
table_wls = pd.DataFrame({'b': round(results_wls.params, 4),
                          'se': round(results_wls.bse, 4),
                          't': round(results_wls.tvalues, 4),
                          'pval': round(results_wls.pvalues, 4)})
print(f'table_wls: \n{table_wls}\n')
```

Script 8.6: `WLS-Robust.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

k401ksubs = woo.dataWoo('401ksubs')

# subsetting data:
k401ksubs_sub = k401ksubs[k401ksubs['fsize'] == 1]

# WLS:
wls_weight = list(1 / k401ksubs_sub['inc'])
reg_wls = smf.wls(formula='nettfa ~ inc + I((age-25)**2) + male + e401k',
                  weights=wls_weight, data=k401ksubs_sub)

# non-robust (default) results:
results_wls = reg_wls.fit()
table_default = pd.DataFrame({'b': round(results_wls.params, 4),
                              'se': round(results_wls.bse, 4),
                              't': round(results_wls.tvalues, 4),
                              'pval': round(results_wls.pvalues, 4)})
print(f'table_default: \n{table_default}\n')

# robust results (Refined White SE):
results_white = reg_wls.fit(cov_type='HC3')
table_white = pd.DataFrame({'b': round(results_white.params, 4),
                            'se': round(results_white.bse, 4),
                            't': round(results_white.tvalues, 4),
                            'pval': round(results_white.pvalues, 4)})
print(f'table_white: \n{table_white}\n')
```

Script 8.7: `Example-8-7.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy as pt

smoke = woo.dataWoo('smoke')

# OLS:
reg_ols = smf.ols(formula='cigs ~ np.log(income) + np.log(cigpric) +'
```

```
                          'educ + age + I(age**2) + restaurn',
                  data=smoke)
results_ols = reg_ols.fit()
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# BP test:
y, X = pt.dmatrices('cigs ~ np.log(income) + np.log(cigpric) + educ +'
                    'age + I(age**2) + restaurn',
                    data=smoke, return_type='dataframe')
result_bp = sm.stats.diagnostic.het_breuschpagan(results_ols.resid, X)
bp_statistic = result_bp[0]
bp_pval = result_bp[1]
print(f'bp_statistic: {bp_statistic}\n')
print(f'bp_pval: {bp_pval}\n')

# FGLS (estimation of the variance function):
smoke['logu2'] = np.log(results_ols.resid ** 2)
reg_fgls = smf.ols(formula='logu2 ~ np.log(income) + np.log(cigpric) +'
                           'educ + age + I(age**2) + restaurn', data=smoke)
results_fgls = reg_fgls.fit()
table_fgls = pd.DataFrame({'b': round(results_fgls.params, 4),
                           'se': round(results_fgls.bse, 4),
                           't': round(results_fgls.tvalues, 4),
                           'pval': round(results_fgls.pvalues, 4)})
print(f'table_fgls: \n{table_fgls}\n')

# FGLS (WLS):
wls_weight = list(1 / np.exp(results_fgls.fittedvalues))
reg_wls = smf.wls(formula='cigs ~ np.log(income) + np.log(cigpric) +'
                          'educ + age + I(age**2) + restaurn',
                  weights=wls_weight, data=smoke)
results_wls = reg_wls.fit()
table_wls = pd.DataFrame({'b': round(results_wls.params, 4),
                          'se': round(results_wls.bse, 4),
                          't': round(results_wls.tvalues, 4),
                          'pval': round(results_wls.pvalues, 4)})
print(f'table_wls: \n{table_wls}\n')
```

9. Scripts Used in Chapter 09

Script 9.1: `Example-9-2-manual.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

hprice1 = woo.dataWoo('hprice1')

# original OLS:
reg = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results = reg.fit()

# regression for RESET test:
```

```
hprice1['fitted_sq'] = results.fittedvalues ** 2
hprice1['fitted_cub'] = results.fittedvalues ** 3
reg_reset = smf.ols(formula='price ~ lotsize + sqrft + bdrms +'
                            ' fitted_sq + fitted_cub', data=hprice1)
results_reset = reg_reset.fit()

# print regression table:
table = pd.DataFrame({'b': round(results_reset.params, 4),
                      'se': round(results_reset.bse, 4),
                      't': round(results_reset.tvalues, 4),
                      'pval': round(results_reset.pvalues, 4)})
print(f'table: \n{table}\n')

# RESET test (H0: all coeffs including "fitted" are=0):
hypotheses = ['fitted_sq = 0', 'fitted_cub = 0']
ftest_man = results_reset.f_test(hypotheses)
fstat_man = ftest_man.statistic[0][0]
fpval_man = ftest_man.pvalue

print(f'fstat_man: {fstat_man}\n')
print(f'fpval_man: {fpval_man}\n')
```

Script 9.2: `Example-9-2-automatic.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import statsmodels.stats.outliers_influence as smo

hprice1 = woo.dataWoo('hprice1')

# original linear regression:
reg = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results = reg.fit()

# automated RESET test:
reset_output = smo.reset_ramsey(res=results, degree=3)
fstat_auto = reset_output.statistic[0][0]
fpval_auto = reset_output.pvalue

print(f'fstat_auto: {fstat_auto}\n')
print(f'fpval_auto: {fpval_auto}\n')
```

Script 9.3: `Nonnested-Test.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

hprice1 = woo.dataWoo('hprice1')

# two alternative models:
reg1 = smf.ols(formula='price ~ lotsize + sqrft + bdrms', data=hprice1)
results1 = reg1.fit()

reg2 = smf.ols(formula='price ~ np.log(lotsize) +'
                       'np.log(sqrft) + bdrms', data=hprice1)
results2 = reg2.fit()

# encompassing test of Davidson & MacKinnon:
```

```
# comprehensive model:
reg3 = smf.ols(formula='price ~ lotsize + sqrft + bdrms + '
                       'np.log(lotsize) + np.log(sqrft)', data=hprice1)
results3 = reg3.fit()

# model 1 vs. comprehensive model:
anovaResults1 = sm.stats.anova_lm(results1, results3)
print(f'anovaResults1: \n{anovaResults1}\n')

# model 2 vs. comprehensive model:
anovaResults2 = sm.stats.anova_lm(results2, results3)
print(f'anovaResults2: \n{anovaResults2}\n')
```

Script 9.4: `Sim-ME-Dep.py`

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import statsmodels.formula.api as smf

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize arrays to store results later (b1 without ME, b1_me with ME):
b1 = np.empty(r)
b1_me = np.empty(r)

# draw a sample of x, fixed over replications:
x = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u:
    u = stats.norm.rvs(0, 1, size=n)

    # draw a sample of ystar:
    ystar = beta0 + beta1 * x + u

    # measurement error and mismeasured y:
    e0 = stats.norm.rvs(0, 1, size=n)
    y = ystar + e0
    df = pd.DataFrame({'ystar': ystar, 'y': y, 'x': x})

    # regress ystar on x and store slope estimate at position i:
    reg_star = smf.ols(formula='ystar ~ x', data=df)
    results_star = reg_star.fit()
    b1[i] = results_star.params['x']

    # regress y on x and store slope estimate at position i:
    reg_me = smf.ols(formula='y ~ x', data=df)
    results_me = reg_me.fit()
    b1_me[i] = results_me.params['x']
```

```
# mean with and without ME:
b1_mean = np.mean(b1)
b1_me_mean = np.mean(b1_me)
print(f'b1_mean: {b1_mean}\n')
print(f'b1_me_mean: {b1_me_mean}\n')

# variance with and without ME:
b1_var = np.var(b1, ddof=1)
b1_me_var = np.var(b1_me, ddof=1)
print(f'b1_var: {b1_var}\n')
print(f'b1_me_var: {b1_me_var}\n')
```

Script 9.5: `Sim-ME-Explan.py`

```
import numpy as np
import scipy.stats as stats
import pandas as pd
import statsmodels.formula.api as smf

# set the random seed:
np.random.seed(1234567)

# set sample size and number of simulations:
n = 1000
r = 10000

# set true parameters (betas):
beta0 = 1
beta1 = 0.5

# initialize b1 arrays to store results later:
b1 = np.empty(r)
b1_me = np.empty(r)

# draw a sample of x, fixed over replications:
xstar = stats.norm.rvs(4, 1, size=n)

# repeat r times:
for i in range(r):
    # draw a sample of u:
    u = stats.norm.rvs(0, 1, size=n)

    # draw a sample of y:
    y = beta0 + beta1 * xstar + u

    # measurement error and mismeasured x:
    e1 = stats.norm.rvs(0, 1, size=n)
    x = xstar + e1
    df = pd.DataFrame({'y': y, 'xstar': xstar, 'x': x})

    # regress y on xstar and store slope estimate at position i:
    reg_star = smf.ols(formula='y ~ xstar', data=df)
    results_star = reg_star.fit()
    b1[i] = results_star.params['xstar']

    # regress y on x and store slope estimate at position i:
    reg_me = smf.ols(formula='y ~ x', data=df)
    results_me = reg_me.fit()
    b1_me[i] = results_me.params['x']
```

```
# mean with and without ME:
b1_mean = np.mean(b1)
b1_me_mean = np.mean(b1_me)
print(f'b1_mean: {b1_mean}\n')
print(f'b1_me_mean: {b1_me_mean}\n')

# variance with and without ME:
b1_var = np.var(b1, ddof=1)
b1_me_var = np.var(b1_me, ddof=1)
print(f'b1_var: {b1_var}\n')
print(f'b1_me_var: {b1_me_var}\n')
```

Script 9.6: NA-NaN-Inf.py

```
import numpy as np
import pandas as pd
import scipy.stats as stats

# nan and inf handling in numpy:
x = np.array([-1, 0, 1, np.nan, np.inf, -np.inf])
logx = np.log(x)
invx = np.array(1 / x)
ncdf = np.array(stats.norm.cdf(x))
isnanx = np.isnan(x)

results = pd.DataFrame({'x': x, 'logx': logx, 'invx': invx,
                        'logx': logx, 'ncdf': ncdf, 'isnanx': isnanx})
print(f'results: \n{results}\n')
```

Script 9.7: Missings.py

```
import wooldridge as woo
import pandas as pd

lawsch85 = woo.dataWoo('lawsch85')
lsat_pd = lawsch85['LSAT']

# create boolean indicator for missings:
missLSAT = lsat_pd.isna()

# LSAT and indicator for Schools No. 120-129:
preview = pd.DataFrame({'lsat_pd': lsat_pd[119:129],
                        'missLSAT': missLSAT[119:129]})
print(f'preview: \n{preview}\n')

# frequencies of indicator:
freq_missLSAT = pd.crosstab(missLSAT, columns='count')
print(f'freq_missLSAT: \n{freq_missLSAT}\n')

# missings for all variables in data frame (counts):
miss_all = lawsch85.isna()
colsums = miss_all.sum(axis=0)
print(f'colsums: \n{colsums}\n')

# computing amount of complete cases:
complete_cases = (miss_all.sum(axis=1) == 0)
freq_complete_cases = pd.crosstab(complete_cases, columns='count')
print(f'freq_complete_cases: \n{freq_complete_cases}\n')
```

Script 9.8: `Missings-Analyses.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.formula.api as smf

lawsch85 = woo.dataWoo('lawsch85')

# missings in numpy:
x_np = np.array(lawsch85['LSAT'])
x_np_bar1 = np.mean(x_np)
x_np_bar2 = np.nanmean(x_np)
print(f'x_np_bar1: {x_np_bar1}\n')
print(f'x_np_bar2: {x_np_bar2}\n')

# missings in pandas:
x_pd = lawsch85['LSAT']
x_pd_bar1 = np.mean(x_pd)
x_pd_bar2 = np.nanmean(x_pd)
print(f'x_pd_bar1: {x_pd_bar1}\n')
print(f'x_pd_bar2: {x_pd_bar2}\n')

# observations and variables:
print(f'lawsch85.shape: {lawsch85.shape}\n')

# regression (missings are taken care of by default):
reg = smf.ols(formula='np.log(salary) ~ LSAT + cost + age', data=lawsch85)
results = reg.fit()
print(f'results.nobs: {results.nobs}\n')
```

Script 9.9: `Outliers.py`

```
import wooldridge as woo
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

rdchem = woo.dataWoo('rdchem')

# OLS regression:
reg = smf.ols(formula='rdintens ~ sales + profmarg', data=rdchem)
results = reg.fit()

# studentized residuals for all observations:
studres = results.get_influence().resid_studentized_external

# display extreme values:
studres_max = np.max(studres)
studres_min = np.min(studres)
print(f'studres_max: {studres_max}\n')
print(f'studres_min: {studres_min}\n')

# histogram (and overlayed density plot):
kde = sm.nonparametric.KDEUnivariate(studres)
kde.fit()

plt.hist(studres, color='grey', density=True)
plt.plot(kde.support, kde.density, color='black', linewidth=2)
plt.ylabel('density')
```

```
plt.xlabel('studres')
plt.savefig('PyGraphs/Outliers.pdf')
```

Script 9.10: LAD.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

rdchem = woo.dataWoo('rdchem')

# OLS regression:
reg_ols = smf.ols(formula='rdintens ~ I(sales/1000) + profmarg', data=rdchem)
results_ols = reg_ols.fit()

table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# LAD regression:
reg_lad = smf.quantreg(formula='rdintens ~ I(sales/1000) + profmarg', data=rdchem)
results_lad = reg_lad.fit(q=.5)

table_lad = pd.DataFrame({'b': round(results_lad.params, 4),
                          'se': round(results_lad.bse, 4),
                          't': round(results_lad.tvalues, 4),
                          'pval': round(results_lad.pvalues, 4)})
print(f'table_lad: \n{table_lad}\n')
```

10. Scripts Used in Chapter 10

Script 10.1: Example-10-2.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

intdef = woo.dataWoo('intdef')

# linear regression of static model (Q function avoids conflicts with keywords):
reg = smf.ols(formula='i3 ~ Q("inf") + Q("def")', data=intdef)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 10.2: Example-Barium.py

```
import wooldridge as woo
import pandas as pd
import matplotlib.pyplot as plt
```

```
barium = woo.dataWoo('barium')
T = len(barium)

# monthly time series starting Feb. 1978:
barium.index = pd.date_range(start='1978-02', periods=T, freq='M')
print(f'barium["chnimp"].head(): \n{barium["chnimp"].head()}\n')

# plot chnimp (default: index on the x-axis):
plt.plot('chnimp', data=barium, color='black', linestyle='-')
plt.ylabel('chnimp')
plt.xlabel('time')
plt.savefig('PyGraphs/Example-Barium.pdf')
```

Script 10.3: Example-StockData.py

```
import pandas_datareader as pdr
import matplotlib.pyplot as plt

# download data for 'F' (= Ford Motor Company) and define start and end:
tickers = ['F']
start_date = '2014-01-01'
end_date = '2015-12-31'

# use pandas_datareader for the import:
F_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# look at imported data:
print(f'F_data.head(): \n{F_data.head()}\n')
print(f'F_data.tail(): \n{F_data.tail()}\n')

# time series plot of adjusted closing prices:
plt.plot('Close', data=F_data, color='black', linestyle='-')
plt.ylabel('Ford Close Price')
plt.xlabel('time')
plt.savefig('PyGraphs/Example-StockData.pdf')
```

Script 10.4: Example-10-4.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

fertil3 = woo.dataWoo('fertil3')
T = len(fertil3)

# define yearly time series beginning in 1913:
fertil3.index = pd.date_range(start='1913', periods=T, freq='Y').year

# add all lags of 'pe' up to order 2:
fertil3['pe_lag1'] = fertil3['pe'].shift(1)
fertil3['pe_lag2'] = fertil3['pe'].shift(2)

# linear regression of model with lags:
reg = smf.ols(formula='gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill', data=fertil3)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
```

```
                       'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 10.5: `Example-10-4-cont.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

fertil3 = woo.dataWoo('fertil3')
T = len(fertil3)

# define yearly time series beginning in 1913:
fertil3.index = pd.date_range(start='1913', periods=T, freq='Y').year

# add all lags of 'pe' up to order 2:
fertil3['pe_lag1'] = fertil3['pe'].shift(1)
fertil3['pe_lag2'] = fertil3['pe'].shift(2)

# linear regression of model with lags:
reg = smf.ols(formula='gfr ~ pe + pe_lag1 + pe_lag2 + ww2 + pill', data=fertil3)
results = reg.fit()

# F test (H0: all pe coefficients are=0):
hypotheses1 = ['pe = 0', 'pe_lag1 = 0', 'pe_lag2 = 0']
ftest1 = results.f_test(hypotheses1)
fstat1 = ftest1.statistic[0][0]
fpval1 = ftest1.pvalue

print(f'fstat1: {fstat1}\n')
print(f'fpval1: {fpval1}\n')

# calculating the LRP:
b = results.params
b_pe_tot = b['pe'] + b['pe_lag1'] + b['pe_lag2']
print(f'b_pe_tot: {b_pe_tot}\n')

# F test (H0: LRP=0):
hypotheses2 = ['pe + pe_lag1 + pe_lag2 = 0']
ftest2 = results.f_test(hypotheses2)
fstat2 = ftest2.statistic[0][0]
fpval2 = ftest2.pvalue

print(f'fstat2: {fstat2}\n')
print(f'fpval2: {fpval2}\n')
```

Script 10.6: `Example-10-7.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

hseinv = woo.dataWoo('hseinv')

# linear regression without time trend:
reg_wot = smf.ols(formula='np.log(invpc) ~ np.log(price)', data=hseinv)
results_wot = reg_wot.fit()

# print regression table:
```

```
table_wot = pd.DataFrame({'b': round(results_wot.params, 4),
                          'se': round(results_wot.bse, 4),
                          't': round(results_wot.tvalues, 4),
                          'pval': round(results_wot.pvalues, 4)})
print(f'table_wot: \n{table_wot}\n')

# linear regression with time trend (data set includes a time variable t):
reg_wt = smf.ols(formula='np.log(invpc) ~ np.log(price) + t', data=hseinv)
results_wt = reg_wt.fit()

# print regression table:
table_wt = pd.DataFrame({'b': round(results_wt.params, 4),
                         'se': round(results_wt.bse, 4),
                         't': round(results_wt.tvalues, 4),
                         'pval': round(results_wt.pvalues, 4)})
print(f'table_wt: \n{table_wt}\n')
```

Script 10.7: `Example-10-11.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

barium = woo.dataWoo('barium')

# linear regression with seasonal effects:
reg = smf.ols(formula='np.log(chnimp) ~ np.log(chempi) + np.log(gas) +'
                      'np.log(rtwex) + befile6 + affile6 + afdec6 +'
                      'feb + mar + apr + may + jun + jul +'
                      'aug + sep + oct + nov + dec',
              data=barium)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

11. Scripts Used in Chapter 11

Script 11.1: `Example-11-4.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

nyse = woo.dataWoo('nyse')
nyse['ret'] = nyse['return']

# add all lags up to order 3:
nyse['ret_lag1'] = nyse['ret'].shift(1)
nyse['ret_lag2'] = nyse['ret'].shift(2)
nyse['ret_lag3'] = nyse['ret'].shift(3)

# linear regression of model with lags:
```

```
reg1 = smf.ols(formula='ret ~ ret_lag1', data=nyse)
reg2 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2', data=nyse)
reg3 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2 + ret_lag3', data=nyse)
results1 = reg1.fit()
results2 = reg2.fit()
results3 = reg3.fit()

# print regression tables:
table1 = pd.DataFrame({'b': round(results1.params, 4),
                       'se': round(results1.bse, 4),
                       't': round(results1.tvalues, 4),
                       'pval': round(results1.pvalues, 4)})
print(f'table1: \n{table1}\n')

table2 = pd.DataFrame({'b': round(results2.params, 4),
                       'se': round(results2.bse, 4),
                       't': round(results2.tvalues, 4),
                       'pval': round(results2.pvalues, 4)})
print(f'table2: \n{table2}\n')

table3 = pd.DataFrame({'b': round(results3.params, 4),
                       'se': round(results3.bse, 4),
                       't': round(results3.tvalues, 4),
                       'pval': round(results3.pvalues, 4)})
print(f'table3: \n{table3}\n')
```

Script 11.2: `Example-EffMkts.py`

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

# download data for 'AAPL' (= Apple) and define start and end:
tickers = ['AAPL']
start_date = '2007-12-31'
end_date = '2016-12-31'

# use pandas_datareader for the import:
AAPL_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# drop ticker symbol from column name:
AAPL_data.columns = AAPL_data.columns.droplevel(level=1)

# calculate return as the log difference:
AAPL_data['ret'] = np.log(AAPL_data['Adj Close']).diff()

# time series plot of adjusted closing prices:
plt.plot('ret', data=AAPL_data, color='black', linestyle='-')
plt.ylabel('Apple Log Returns')
plt.xlabel('time')
plt.savefig('PyGraphs/Example-EffMkts.pdf')

# linear regression of models with lags:
AAPL_data['ret_lag1'] = AAPL_data['ret'].shift(1)
AAPL_data['ret_lag2'] = AAPL_data['ret'].shift(2)
AAPL_data['ret_lag3'] = AAPL_data['ret'].shift(3)

reg1 = smf.ols(formula='ret ~ ret_lag1', data=AAPL_data)
```

```
reg2 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2', data=AAPL_data)
reg3 = smf.ols(formula='ret ~ ret_lag1 + ret_lag2 + ret_lag3', data=AAPL_data)
results1 = reg1.fit()
results2 = reg2.fit()
results3 = reg3.fit()

# print regression tables:
table1 = pd.DataFrame({'b': round(results1.params, 4),
                       'se': round(results1.bse, 4),
                       't': round(results1.tvalues, 4),
                       'pval': round(results1.pvalues, 4)})
print(f'table1: \n{table1}\n')

table2 = pd.DataFrame({'b': round(results2.params, 4),
                       'se': round(results2.bse, 4),
                       't': round(results2.tvalues, 4),
                       'pval': round(results2.pvalues, 4)})
print(f'table2: \n{table2}\n')

table3 = pd.DataFrame({'b': round(results3.params, 4),
                       'se': round(results3.bse, 4),
                       't': round(results3.tvalues, 4),
                       'pval': round(results3.pvalues, 4)})
print(f'table3: \n{table3}\n')
```

Script 11.3: `Simulate-RandomWalk.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# initialize plot:
x_range = np.linspace(0, 50, num=51)
plt.ylim([-18, 18])
plt.xlim([0, 50])

# loop over draws:
for r in range(0, 30):
    # i.i.d. standard normal shock:
    e = stats.norm.rvs(0, 1, size=51)

    # set first entry to 0 (gives y_0 = 0):
    e[0] = 0

    # random walk as cumulative sum of shocks:
    y = np.cumsum(e)

    # add line to graph:
    plt.plot(x_range, y, color='lightgrey', linestyle='-')

plt.axhline(linewidth=2, linestyle='--', color='black')
plt.ylabel('y')
plt.xlabel('time')
plt.savefig('PyGraphs/Simulate-RandomWalk.pdf')
```

Script 11.4: `Simulate-RandomWalkDrift.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# initialize plot:
x_range = np.linspace(0, 50, num=51)
plt.ylim([0, 100])
plt.xlim([0, 50])

# loop over draws:
for r in range(0, 30):
    # i.i.d. standard normal shock:
    e = stats.norm.rvs(0, 1, size=51)

    # set first entry to 0 (gives y_0 = 0):
    e[0] = 0

    # random walk as cumulative sum of shocks plus drift:
    y = np.cumsum(e) + 2 * x_range

    # add line to graph:
    plt.plot(x_range, y, color='lightgrey', linestyle='-')

plt.plot(x_range, 2 * x_range, linewidth=2, linestyle='--', color='black')
plt.ylabel('y')
plt.xlabel('time')
plt.savefig('PyGraphs/Simulate-RandomWalkDrift.pdf')
```

Script 11.5: `Simulate-RandomWalkDrift-Diff.py`

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

# initialize plot:
x_range = np.linspace(1, 50, num=50)
plt.ylim([-1, 5])
plt.xlim([0, 50])

# loop over draws:
for r in range(0, 30):
    # i.i.d. standard normal shock and cumulative sum of shocks:
    e = stats.norm.rvs(0, 1, size=51)
    e[0] = 0
    y = np.cumsum(2 + e)

    # first difference:
    Dy = y[1:51] - y[0:50]

    # add line to graph:
    plt.plot(x_range, Dy, color='lightgrey', linestyle='-')

plt.axhline(y=2, linewidth=2, linestyle='--', color='black')
```

```
plt.ylabel('y')
plt.xlabel('time')
plt.savefig('PyGraphs/Simulate-RandomWalkDrift-Diff.pdf')
```

Script 11.6: `Example-11-6.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

fertil3 = woo.dataWoo('fertil3')
T = len(fertil3)

# define time series (years only) beginning in 1913:
fertil3.index = pd.date_range(start='1913', periods=T, freq='Y').year

# compute first differences:
fertil3['gfr_diff1'] = fertil3['gfr'].diff()
fertil3['pe_diff1'] = fertil3['pe'].diff()
print(f'fertil3.head(): \n{fertil3.head()}\n')

# linear regression of model with first differences:
reg1 = smf.ols(formula='gfr_diff1 ~ pe_diff1', data=fertil3)
results1 = reg1.fit()

# print regression table:
table1 = pd.DataFrame({'b': round(results1.params, 4),
                       'se': round(results1.bse, 4),
                       't': round(results1.tvalues, 4),
                       'pval': round(results1.pvalues, 4)})
print(f'table1: \n{table1}\n')

# linear regression of model with lagged differences:
fertil3['pe_diff1_lag1'] = fertil3['pe_diff1'].shift(1)
fertil3['pe_diff1_lag2'] = fertil3['pe_diff1'].shift(2)

reg2 = smf.ols(formula='gfr_diff1 ~ pe_diff1 + pe_diff1_lag1 + pe_diff1_lag2',
               data=fertil3)
results2 = reg2.fit()

# print regression table:
table2 = pd.DataFrame({'b': round(results2.params, 4),
                       'se': round(results2.bse, 4),
                       't': round(results2.tvalues, 4),
                       'pval': round(results2.pvalues, 4)})
print(f'table2: \n{table2}\n')
```

12. Scripts Used in Chapter 12

Script 12.1: `Example-12-2-Static.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

phillips = woo.dataWoo('phillips')
T = len(phillips)
```

```
# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=T, freq='Y')
phillips.index = date_range.year

# estimation of static Phillips curve:
yt96 = (phillips['year'] <= 1996)
reg_s = smf.ols(formula='Q("inf") ~ unem', data=phillips, subset=yt96)
results_s = reg_s.fit()

# residuals and AR(1) test:
phillips['resid_s'] = results_s.resid
phillips['resid_s_lag1'] = phillips['resid_s'].shift(1)
reg = smf.ols(formula='resid_s ~ resid_s_lag1', data=phillips, subset=yt96)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 12.2: `Example-12-2-ExpAug.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

phillips = woo.dataWoo('phillips')
T = len(phillips)

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=T, freq='Y')
phillips.index = date_range.year

# estimation of expectations-augmented Phillips curve:
yt96 = (phillips['year'] <= 1996)
phillips['inf_diff1'] = phillips['inf'].diff()
reg_ea = smf.ols(formula='inf_diff1 ~ unem', data=phillips, subset=yt96)
results_ea = reg_ea.fit()

phillips['resid_ea'] = results_ea.resid
phillips['resid_ea_lag1'] = phillips['resid_ea'].shift(1)
reg = smf.ols(formula='resid_ea ~ resid_ea_lag1', data=phillips, subset=yt96)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 12.3: `Example-12-4.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

```
barium = woo.dataWoo('barium')
T = len(barium)

# monthly time series starting Feb. 1978:
barium.index = pd.date_range(start='1978-02', periods=T, freq='M')

reg = smf.ols(formula='np.log(chnimp) ~ np.log(chempi) + np.log(gas) +'
                      'np.log(rtwex) + befile6 + affile6 + afdec6',
              data=barium)
results = reg.fit()

# automatic test:
bg_result = sm.stats.diagnostic.acorr_breusch_godfrey(results, nlags=3)
fstat_auto = bg_result[2]
fpval_auto = bg_result[3]
print(f'fstat_auto: {fstat_auto}\n')
print(f'fpval_auto: {fpval_auto}\n')

# pedestrian test:
barium['resid'] = results.resid
barium['resid_lag1'] = barium['resid'].shift(1)
barium['resid_lag2'] = barium['resid'].shift(2)
barium['resid_lag3'] = barium['resid'].shift(3)

reg_manual = smf.ols(formula='resid ~ resid_lag1 + resid_lag2 + resid_lag3 +'
                             'np.log(chempi) + np.log(gas) + np.log(rtwex) +'
                             'befile6 + affile6 + afdec6', data=barium)
results_manual = reg_manual.fit()

hypotheses = ['resid_lag1 = 0', 'resid_lag2 = 0', 'resid_lag3 = 0']
ftest_manual = results_manual.f_test(hypotheses)
fstat_manual = ftest_manual.statistic[0][0]
fpval_manual = ftest_manual.pvalue
print(f'fstat_manual: {fstat_manual}\n')
print(f'fpval_manual: {fpval_manual}\n')
```

Script 12.4: `Example-DWtest.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

phillips = woo.dataWoo('phillips')
T = len(phillips)

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=T, freq='Y')
phillips.index = date_range.year

# estimation of both Phillips curve models:
yt96 = (phillips['year'] <= 1996)
phillips['inf_diff1'] = phillips['inf'].diff()
reg_s = smf.ols(formula='Q("inf") ~ unem', data=phillips, subset=yt96)
reg_ea = smf.ols(formula='inf_diff1 ~ unem', data=phillips, subset=yt96)
results_s = reg_s.fit()
results_ea = reg_ea.fit()

# DW tests:
```

```
DW_s = sm.stats.stattools.durbin_watson(results_s.resid)
DW_ea = sm.stats.stattools.durbin_watson(results_ea.resid)
print(f'DW_s: {DW_s}\n')
print(f'DW_ea: {DW_ea}\n')
```

Script 12.5: `Example-12-5.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.api as sm
import patsy as pt

barium = woo.dataWoo('barium')
T = len(barium)

# monthly time series starting Feb. 1978:
barium.index = pd.date_range(start='1978-02', periods=T, freq='M')

# perform the Cochrane-Orcutt estimation (iterative procedure):
y, X = pt.dmatrices('np.log(chnimp) ~ np.log(chempi) + np.log(gas) +'
                    'np.log(rtwex) + befile6 + affile6 + afdec6',
                    data=barium, return_type='dataframe')
reg = sm.GLSAR(y, X)
CORC_results = reg.iterative_fit(maxiter=100)
table = pd.DataFrame({'b_CORC': CORC_results.params,
                      'se_CORC': CORC_results.bse})
print(f'reg.rho: {reg.rho}\n')
print(f'table: \n{table}\n')
```

Script 12.6: `Example-12-1.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf

prminwge = woo.dataWoo('prminwge')
T = len(prminwge)
prminwge['time'] = prminwge['year'] - 1949
prminwge.index = pd.date_range(start='1950', periods=T, freq='Y').year

# OLS regression:
reg = smf.ols(formula='np.log(prepop) ~ np.log(mincov) + np.log(prgnp) +'
                      'np.log(usgnp) + time', data=prminwge)

# results with regular SE:
results_regu = reg.fit()

# print regression table:
table_regu = pd.DataFrame({'b': round(results_regu.params, 4),
                           'se': round(results_regu.bse, 4),
                           't': round(results_regu.tvalues, 4),
                           'pval': round(results_regu.pvalues, 4)})
print(f'table_regu: \n{table_regu}\n')

# results with HAC SE:
results_hac = reg.fit(cov_type='HAC', cov_kwds={'maxlags': 2})

# print regression table:
```

```
table_hac = pd.DataFrame({'b': round(results_hac.params, 4),
                          'se': round(results_hac.bse, 4),
                          't': round(results_hac.tvalues, 4),
                          'pval': round(results_hac.pvalues, 4)})
print(f'table_hac: \n{table_hac}\n')
```

Script 12.7: **Example-12-9.py**

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

nyse = woo.dataWoo('nyse')
nyse['ret'] = nyse['return']
nyse['ret_lag1'] = nyse['ret'].shift(1)

# linear regression of model:
reg = smf.ols(formula='ret ~ ret_lag1', data=nyse)
results = reg.fit()

# squared residuals:
nyse['resid_sq'] = results.resid ** 2
nyse['resid_sq_lag1'] = nyse['resid_sq'].shift(1)

# model for squared residuals:
ARCHreg = smf.ols(formula='resid_sq ~ resid_sq_lag1', data=nyse)
results_ARCH = ARCHreg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results_ARCH.params, 4),
                      'se': round(results_ARCH.bse, 4),
                      't': round(results_ARCH.tvalues, 4),
                      'pval': round(results_ARCH.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 12.8: **Example-ARCH.py**

```
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.formula.api as smf

# download data for 'AAPL' (= Apple) and define start and end:
tickers = ['AAPL']
start_date = '2007-12-31'
end_date = '2016-12-31'

# use pandas_datareader for the import:
AAPL_data = pdr.data.DataReader(tickers, 'yahoo', start_date, end_date)

# drop ticker symbol from column name:
AAPL_data.columns = AAPL_data.columns.droplevel(level=1)

# calculate return as the difference of logged prices:
AAPL_data['ret'] = np.log(AAPL_data['Adj Close']).diff()
AAPL_data['ret_lag1'] = AAPL_data['ret'].shift(1)

# AR(1) model for returns:
reg = smf.ols(formula='ret ~ ret_lag1', data=AAPL_data)
results = reg.fit()
```

```
# squared residuals:
AAPL_data['resid_sq'] = results.resid ** 2
AAPL_data['resid_sq_lag1'] = AAPL_data['resid_sq'].shift(1)

# model for squared residuals:
ARCHreg = smf.ols(formula='resid_sq ~ resid_sq_lag1', data=AAPL_data)
results_ARCH = ARCHreg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results_ARCH.params, 4),
                      'se': round(results_ARCH.bse, 4),
                      't': round(results_ARCH.tvalues, 4),
                      'pval': round(results_ARCH.pvalues, 4)})
print(f'table: \n{table}\n')
```

13. Scripts Used in Chapter 13

Script 13.1: `Example-13-2.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

cps78_85 = woo.dataWoo('cps78_85')

# OLS results including interaction terms:
reg = smf.ols(formula='lwage ~ y85*(educ+female) + exper +'
                      'I((exper**2)/100) + union',
              data=cps78_85)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 13.2: `Example-13-3-1.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

kielmc = woo.dataWoo('kielmc')

# separate regressions for 1978 and 1981:
y78 = (kielmc['year'] == 1978)
reg78 = smf.ols(formula='rprice ~ nearinc', data=kielmc, subset=y78)
results78 = reg78.fit()

y81 = (kielmc['year'] == 1981)
reg81 = smf.ols(formula='rprice ~ nearinc', data=kielmc, subset=y81)
results81 = reg81.fit()

# joint regression including an interaction term:
reg_joint = smf.ols(formula='rprice ~ nearinc * C(year)', data=kielmc)
```

```
results_joint = reg_joint.fit()

# print regression tables:
table_78 = pd.DataFrame({'b': round(results78.params, 4),
                         'se': round(results78.bse, 4),
                         't': round(results78.tvalues, 4),
                         'pval': round(results78.pvalues, 4)})
print(f'table_78: \n{table_78}\n')

table_81 = pd.DataFrame({'b': round(results81.params, 4),
                         'se': round(results81.bse, 4),
                         't': round(results81.tvalues, 4),
                         'pval': round(results81.pvalues, 4)})
print(f'table_81: \n{table_81}\n')

table_joint = pd.DataFrame({'b': round(results_joint.params, 4),
                            'se': round(results_joint.bse, 4),
                            't': round(results_joint.tvalues, 4),
                            'pval': round(results_joint.pvalues, 4)})
print(f'table_joint: \n{table_joint}\n')
```

Script 13.3: Example-13-3-2.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

kielmc = woo.dataWoo('kielmc')

# difference in difference (DiD):
reg_did = smf.ols(formula='np.log(rprice) ~ nearinc*C(year)', data=kielmc)
results_did = reg_did.fit()

# print regression table:
table_did = pd.DataFrame({'b': round(results_did.params, 4),
                          'se': round(results_did.bse, 4),
                          't': round(results_did.tvalues, 4),
                          'pval': round(results_did.pvalues, 4)})
print(f'table_did: \n{table_did}\n')

# DiD with control variables:
reg_didC = smf.ols(formula='np.log(rprice) ~ nearinc*C(year) + age +'
                           'I(age**2) + np.log(intst) + np.log(land) +'
                           'np.log(area) + rooms + baths',
                   data=kielmc)
results_didC = reg_didC.fit()

# print regression table:
table_didC = pd.DataFrame({'b': round(results_didC.params, 4),
                           'se': round(results_didC.bse, 4),
                           't': round(results_didC.tvalues, 4),
                           'pval': round(results_didC.pvalues, 4)})
print(f'table_didC: \n{table_didC}\n')
```

Script 13.4: Example-FD.py

```
import wooldridge as woo
import numpy as np
import pandas as pd
```

```
import statsmodels.formula.api as smf
import linearmodels as plm

crime2 = woo.dataWoo('crime2')

# create time variable dummy by converting a Boolean variable to an integer:
crime2['t'] = (crime2['year'] == 87).astype(int)  # False=0, True=1

# create an index in this balanced data set by combining two arrays:
id_tmp = np.linspace(1, 46, num=46)
crime2['id'] = np.sort(np.concatenate([id_tmp, id_tmp]))

# manually calculate first differences per entity for crmrte and unem:
crime2['crmrte_diff1'] = \
    crime2.sort_values(['id', 'year']).groupby('id')['crmrte'].diff()
crime2['unem_diff1'] = \
    crime2.sort_values(['id', 'year']).groupby('id')['unem'].diff()
var_selection = ['id', 't', 'crimes', 'unem', 'crmrte_diff1', 'unem_diff1']
print(f'crime2[var_selection].head(): \n{crime2[var_selection].head()}\n')

# estimate FD model with statmodels on differenced data:
reg_sm = smf.ols(formula='crmrte_diff1 ~ unem_diff1', data=crime2)
results_sm = reg_sm.fit()

# print results:
table_sm = pd.DataFrame({'b': round(results_sm.params, 4),
                         'se': round(results_sm.bse, 4),
                         't': round(results_sm.tvalues, 4),
                         'pval': round(results_sm.pvalues, 4)})
print(f'table_sm: \n{table_sm}\n')

# estimate FD model with linearmodels:
crime2 = crime2.set_index(['id', 'year'])
reg_plm = plm.FirstDifferenceOLS.from_formula(formula='crmrte ~ t + unem',
                                              data=crime2)
results_plm = reg_plm.fit()

# print results:
table_plm = pd.DataFrame({'b': round(results_plm.params, 4),
                          'se': round(results_plm.std_errors, 4),
                          't': round(results_plm.tstats, 4),
                          'pval': round(results_plm.pvalues, 4)})
print(f'table_plm: \n{table_plm}\n')
```

Script 13.5: `Example-13-9.py`

```
import wooldridge as woo
import numpy as np
import linearmodels as plm

crime4 = woo.dataWoo('crime4')
crime4 = crime4.set_index(['county', 'year'], drop=False)

# estimate FD model:
reg = plm.FirstDifferenceOLS.from_formula(
    formula='np.log(crmrte) ~ year + d83 + d84 + d85 + d86 + d87 +'
            'lprbarr + lprbconv + lprbpris + lavgsen + lpolpc',
    data=crime4)
```

```
results = reg.fit()
print(f'results: \n{results}\n')
```

14. Scripts Used in Chapter 14

Script 14.1: `Example-14-2.py`

```
import wooldridge as woo
import pandas as pd
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan = wagepan.set_index(['nr', 'year'], drop=False)

# FE model estimation:
reg = plm.PanelOLS.from_formula(
    formula='lwage ~ married + union + C(year)*educ + EntityEffects',
    data=wagepan, drop_absorbed=True)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.std_errors, 4),
                      't': round(results.tstats, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 14.2: `Example-14-4-1.py`

```
import wooldridge as woo

wagepan = woo.dataWoo('wagepan')

# print relevant dimensions for panel:
N = wagepan.shape[0]
T = wagepan['year'].drop_duplicates().shape[0]
n = wagepan['nr'].drop_duplicates().shape[0]
print(f'N: {N}\n')
print(f'T: {T}\n')
print(f'n: {n}\n')

# check non-varying variables

# (I) across time and within individuals by calculating individual
# specific variances for each variable:
isv_nr = (wagepan.groupby('nr').var() == 0)  # True, if variance is zero
# choose variables where all grouped variances are zero:
noVar_nr = isv_nr.all(axis=0)  # which cols are completely True
print(f'isv_nr.columns[noVar_nr]: \n{isv_nr.columns[noVar_nr]}\n')

# (II) across individuals within one point in time for each variable:
isv_t = (wagepan.groupby('year').var() == 0)
noVar_t = isv_t.all(axis=0)
print(f'isv_t.columns[noVar_t]: \n{isv_t.columns[noVar_t]}\n')
```

Script 14.3: `Example-14-4-2.py`

```
import wooldridge as woo
import pandas as pd
```

```
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')

# estimate different models:
wagepan = wagepan.set_index(['nr', 'year'], drop=False)

reg_ols = plm.PooledOLS.from_formula(
    formula='lwage ~ educ + black + hisp + exper + I(exper**2) +'
            'married + union + C(year)', data=wagepan)
results_ols = reg_ols.fit()

reg_re = plm.RandomEffects.from_formula(
    formula='lwage ~ educ + black + hisp + exper + I(exper**2) +'
            'married + union + C(year)', data=wagepan)
results_re = reg_re.fit()

reg_fe = plm.PanelOLS.from_formula(
    formula='lwage ~ I(exper**2) + married + union +'
            'C(year) + EntityEffects', data=wagepan)
results_fe = reg_fe.fit()

# print results:
theta_hat = results_re.theta.iloc[0, 0]
print(f'theta_hat: {theta_hat}\n')

table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.std_errors, 4),
                          't': round(results_ols.tstats, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

table_re = pd.DataFrame({'b': round(results_re.params, 4),
                         'se': round(results_re.std_errors, 4),
                         't': round(results_re.tstats, 4),
                         'pval': round(results_re.pvalues, 4)})
print(f'table_re: \n{table_re}\n')

table_fe = pd.DataFrame({'b': round(results_fe.params, 4),
                         'se': round(results_fe.std_errors, 4),
                         't': round(results_fe.tstats, 4),
                         'pval': round(results_fe.pvalues, 4)})
print(f'table_fe: \n{table_fe}\n')
```

Script 14.4: `Example-HausmTest.py`

```
import wooldridge as woo
import numpy as np
import linearmodels as plm
import scipy.stats as stats

wagepan = woo.dataWoo('wagepan')
wagepan = wagepan.set_index(['nr', 'year'], drop=False)

# estimation of FE and RE:
reg_fe = plm.PanelOLS.from_formula(formula='lwage ~ I(exper**2) + married +'
                                          'union + C(year) + EntityEffects',
                                   data=wagepan)
results_fe = reg_fe.fit()
b_fe = results_fe.params
```

```
b_fe_cov = results_fe.cov

reg_re = plm.RandomEffects.from_formula(
    formula='lwage ~ educ + black + hisp + exper + I(exper**2)'
            '+ married + union + C(year)', data=wagepan)
results_re = reg_re.fit()
b_re = results_re.params
b_re_cov = results_re.cov

# Hausman test of FE vs. RE
# (I) find overlapping coefficients:
common_coef = set(results_fe.params.index).intersection(results_re.params.index)

# (II) calculate differences between FE and RE:
b_diff = np.array(results_fe.params[common_coef] - results_re.params[common_coef])
df = len(b_diff)
b_diff.reshape((df, 1))
b_cov_diff = np.array(b_fe_cov.loc[common_coef, common_coef] -
                      b_re_cov.loc[common_coef, common_coef])
b_cov_diff.reshape((df, df))

# (III) calculate test statistic:
stat = abs(np.transpose(b_diff) @ np.linalg.inv(b_cov_diff) @ b_diff)
pval = 1 - stats.chi2.cdf(stat, df)

print(f'stat: {stat}\n')
print(f'pval: {pval}\n')
```

Script 14.5: `Example-Dummy-CRE-1.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan['t'] = wagepan['year']
wagepan['entity'] = wagepan['nr']
wagepan = wagepan.set_index(['nr'])

# include group specific means:
wagepan['married_b'] = wagepan.groupby('nr').mean()['married']
wagepan['union_b'] = wagepan.groupby('nr').mean()['union']
wagepan = wagepan.set_index(['year'], append=True)

# estimate FE parameters in 3 different ways:
reg_we = plm.PanelOLS.from_formula(
    formula='lwage ~ married + union + C(t)*educ + EntityEffects',
    drop_absorbed=True, data=wagepan)
results_we = reg_we.fit()

reg_dum = smf.ols(
    formula='lwage ~ married + union + C(t)*educ + C(entity)',
    data=wagepan)
results_dum = reg_dum.fit()

reg_cre = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + C(t)*educ + married_b + union_b',
    data=wagepan)
results_cre = reg_cre.fit()
```

```
# compare to RE estimates:
reg_re = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + C(t)*educ',
    data=wagepan)
results_re = reg_re.fit()

var_selection = ['married', 'union', 'C(t)[T.1982]:educ']

# print results:
table = pd.DataFrame({'b_we': round(results_we.params[var_selection], 4),
                      'b_dum': round(results_dum.params[var_selection], 4),
                      'b_cre': round(results_cre.params[var_selection], 4),
                      'b_re': round(results_re.params[var_selection], 4)})
print(f'table: \n{table}\n')
```

Script 14.6: **Example-CRE-test-RE.py**

```
import wooldridge as woo
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan['t'] = wagepan['year']
wagepan['entity'] = wagepan['nr']
wagepan = wagepan.set_index(['nr'])

# include group specific means:
wagepan['married_b'] = wagepan.groupby('nr').mean()['married']
wagepan['union_b'] = wagepan.groupby('nr').mean()['union']
wagepan = wagepan.set_index(['year'], append=True)

# estimate CRE:
reg_cre = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + C(t)*educ  + married_b + union_b',
    data=wagepan)
results_cre = reg_cre.fit()

# RE test as an Wald test on the CRE specific coefficients:
wtest = results_cre.wald_test(formula='married_b = union_b = 0')
print(f'wtest: \n{wtest}\n')
```

Script 14.7: **Example-CRE-2.py**

```
import wooldridge as woo
import pandas as pd
import linearmodels as plm

wagepan = woo.dataWoo('wagepan')
wagepan['t'] = wagepan['year']
wagepan['entity'] = wagepan['nr']
wagepan = wagepan.set_index(['nr'])

# include group specific means:
wagepan['married_b'] = wagepan.groupby('nr').mean()['married']
wagepan['union_b'] = wagepan.groupby('nr').mean()['union']
wagepan = wagepan.set_index(['year'], append=True)

# estimate CRE paramters:
reg = plm.RandomEffects.from_formula(
    formula='lwage ~ married + union + educ +'
```

```
            'black + hisp + married_b + union_b',
    data=wagepan)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.std_errors, 4),
                      't': round(results.tstats, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 14.8: `Example-13-9-ClSE.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels as plm

crime4 = woo.dataWoo('crime4')
crime4 = crime4.set_index(['county', 'year'], drop=False)

# estimate FD model:
reg = plm.FirstDifferenceOLS.from_formula(
    formula='np.log(crmrte) ~ year + d83 + d84 + d85 + d86 + d87 +'
            'lprbarr + lprbconv + lprbpris + lavgsen + lpolpc',
    data=crime4)

# regression with standard SE:
results_default = reg.fit()

# regression with "clustered" SE:
results_cluster = reg.fit(cov_type='clustered', cluster_entity=True,
                          debiased=False)

# regression with "clustered" SE (small-sample correction):
results_css = reg.fit(cov_type='clustered', cluster_entity=True)

# print results:
table = pd.DataFrame({'b': round(results_default.params, 4),
                      'se_default': round(results_default.std_errors, 4),
                      'se_cluster': round(results_cluster.std_errors, 4),
                      'se_css': round(results_css.std_errors, 4)})
print(f'table: \n{table}\n')
```

15. Scripts Used in Chapter 15

Script 15.1: `Example-15-1.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])
```

```
cov_yz = np.cov(mroz['lwage'], mroz['fatheduc'])[1, 0]
cov_xy = np.cov(mroz['educ'], mroz['lwage'])[1, 0]
cov_xz = np.cov(mroz['educ'], mroz['fatheduc'])[1, 0]
var_x = np.var(mroz['educ'], ddof=1)
x_bar = np.mean(mroz['educ'])
y_bar = np.mean(mroz['lwage'])

# OLS slope parameter manually:
b_ols_man = cov_xy / var_x
print(f'b_ols_man: {b_ols_man}\n')

# IV slope parameter manually:
b_iv_man = cov_yz / cov_xz
print(f'b_iv_man: {b_iv_man}\n')

# OLS automatically:
reg_ols = smf.ols(formula='np.log(wage) ~ educ', data=mroz)
results_ols = reg_ols.fit()

# print regression table:
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# IV automatically:
reg_iv = iv.IV2SLS.from_formula(formula='np.log(wage) ~ 1 + [educ ~ fatheduc]',
                                data=mroz)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Script 15.2: `Example-15-4.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf

card = woo.dataWoo('card')

# checking for relevance with reduced form:
reg_redf = smf.ols(
    formula='educ ~ nearc4 + exper + I(exper**2) + black + smsa +'
    'south + smsa66 + reg662 + reg663 + reg664 + reg665 + reg666 +'
    'reg667 + reg668 + reg669', data=card)
results_redf = reg_redf.fit()

# print regression table:
table_redf = pd.DataFrame({'b': round(results_redf.params, 4),
                           'se': round(results_redf.bse, 4),
                           't': round(results_redf.tvalues, 4),
```

```
                              'pval': round(results_redf.pvalues, 4)})
print(f'table_redf: \n{table_redf}\n')

# OLS:
reg_ols = smf.ols(
    formula='np.log(wage) ~ educ + exper + I(exper**2) + black + smsa +'
    'south + smsa66 + reg662 + reg663 + reg664 + reg665 +'
    'reg666 + reg667 + reg668 + reg669', data=card)
results_ols = reg_ols.fit()

# print regression table:
table_ols = pd.DataFrame({'b': round(results_ols.params, 4),
                          'se': round(results_ols.bse, 4),
                          't': round(results_ols.tvalues, 4),
                          'pval': round(results_ols.pvalues, 4)})
print(f'table_ols: \n{table_ols}\n')

# IV automatically:
reg_iv = iv.IV2SLS.from_formula(
    formula='np.log(wage)~ 1 + exper + I(exper**2) + black + smsa + '
            'south + smsa66 + reg662 + reg663 + reg664 + reg665 +'
            'reg666 + reg667 + reg668 + reg669 + [educ ~ nearc4]',
    data=card)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Script 15.3: `Example-15-5.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 1st stage (reduced form):
reg_redf = smf.ols(formula='educ ~ exper + I(exper**2) + motheduc + fatheduc',
                   data=mroz)
results_redf = reg_redf.fit()
mroz['educ_fitted'] = results_redf.fittedvalues

# print regression table:
table_redf = pd.DataFrame({'b': round(results_redf.params, 4),
                           'se': round(results_redf.bse, 4),
                           't': round(results_redf.tvalues, 4),
                           'pval': round(results_redf.pvalues, 4)})
print(f'table_redf: \n{table_redf}\n')

# 2nd stage:
reg_secstg = smf.ols(formula='np.log(wage) ~ educ_fitted + exper + I(exper**2)',
```

```
                     data=mroz)
results_secstg = reg_secstg.fit()

# print regression table:
table_secstg = pd.DataFrame({'b': round(results_secstg.params, 4),
                             'se': round(results_secstg.bse, 4),
                             't': round(results_secstg.tvalues, 4),
                             'pval': round(results_secstg.pvalues, 4)})
print(f'table_secstg: \n{table_secstg}\n')

# IV automatically:
reg_iv = iv.IV2SLS.from_formula(
    formula='np.log(wage) ~ 1 + exper + I(exper**2) +'
            '[educ  ~ motheduc + fatheduc]',
    data=mroz)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

Script 15.4: `Example-15-7.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 1st stage (reduced form):
reg_redf = smf.ols(formula='educ ~ exper + I(exper**2) + motheduc + fatheduc',
                   data=mroz)
results_redf = reg_redf.fit()
mroz['resid'] = results_redf.resid

# 2nd stage:
reg_secstg = smf.ols(formula='np.log(wage)~ resid + educ + exper + I(exper**2)',
                     data=mroz)
results_secstg = reg_secstg.fit()

# print regression table:
table_secstg = pd.DataFrame({'b': round(results_secstg.params, 4),
                             'se': round(results_secstg.bse, 4),
                             't': round(results_secstg.tvalues, 4),
                             'pval': round(results_secstg.pvalues, 4)})
print(f'table_secstg: \n{table_secstg}\n')
```

Script 15.5: `Example-15-8.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv
```

```
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# IV regression:
reg_iv = iv.IV2SLS.from_formula(formula='np.log(wage) ~ 1 + exper + I(exper**2) +'
                                         '[educ ~ motheduc + fatheduc]', data=mroz)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')

# auxiliary regression:
mroz['resid_iv'] = results_iv.resids
reg_aux = smf.ols(formula='resid_iv ~ exper + I(exper**2) + motheduc + fatheduc',
                  data=mroz)
results_aux = reg_aux.fit()

# calculations for test:
r2 = results_aux.rsquared
n = results_aux.nobs
teststat = n * r2
pval = 1 - stats.chi2.cdf(teststat, 1)

print(f'r2: {r2}\n')
print(f'n: {n}\n')
print(f'teststat: {teststat}\n')
print(f'pval: {pval}\n')
```

Script 15.6: `Example-15-10.py`

```
import wooldridge as woo
import pandas as pd
import linearmodels.iv as iv

jtrain = woo.dataWoo('jtrain')

# define panel data (for 1987 and 1988 only):
jtrain_87_88 = jtrain.loc[(jtrain['year'] == 1987) | (jtrain['year'] == 1988), :]
jtrain_87_88 = jtrain_87_88.set_index(['fcode', 'year'])

# manual computation of deviations of entity means:
jtrain_87_88['lscrap_diff1'] = \
    jtrain_87_88.sort_values(['fcode', 'year']).groupby('fcode')['lscrap'].diff()
jtrain_87_88['hrsemp_diff1'] = \
    jtrain_87_88.sort_values(['fcode', 'year']).groupby('fcode')['hrsemp'].diff()
jtrain_87_88['grant_diff1'] = \
    jtrain_87_88.sort_values(['fcode', 'year']).groupby('fcode')['grant'].diff()

# IV regression:
reg_iv = iv.IV2SLS.from_formula(
    formula='lscrap_diff1 ~ 1 + [hrsemp_diff1 ~ grant_diff1]',
```

```
    data=jtrain_87_88)
results_iv = reg_iv.fit(cov_type='unadjusted', debiased=True)

# print regression table:
table_iv = pd.DataFrame({'b': round(results_iv.params, 4),
                         'se': round(results_iv.std_errors, 4),
                         't': round(results_iv.tstats, 4),
                         'pval': round(results_iv.pvalues, 4)})
print(f'table_iv: \n{table_iv}\n')
```

16. Scripts Used in Chapter 16

Script 16.1: `Example-16-5-2SLS.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import linearmodels.iv as iv

mroz = woo.dataWoo('mroz')

# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 2SLS regressions:
reg_iv1 = iv.IV2SLS.from_formula(
    'hours ~ 1 + educ +  age + kidslt6 + nwifeinc +'
    '[np.log(wage) ~ exper + I(exper**2)]', data=mroz)
results_iv1 = reg_iv1.fit(cov_type='unadjusted', debiased=True)

reg_iv2 = iv.IV2SLS.from_formula(
    'np.log(wage) ~ 1 + educ + exper + I(exper**2) +'
    '[hours ~ age + kidslt6 + nwifeinc]', data=mroz)
results_iv2 = reg_iv2.fit(cov_type='unadjusted', debiased=True)

# print results:
table_iv1 = pd.DataFrame({'b': round(results_iv1.params, 4),
                          'se': round(results_iv1.std_errors, 4),
                          't': round(results_iv1.tstats, 4),
                          'pval': round(results_iv1.pvalues, 4)})
print(f'table_iv1: \n{table_iv1}\n')

table_iv2 = pd.DataFrame({'b': round(results_iv2.params, 4),
                          'se': round(results_iv2.std_errors, 4),
                          't': round(results_iv2.tstats, 4),
                          'pval': round(results_iv2.pvalues, 4)})
print(f'table_iv2: \n{table_iv2}\n')

cor_u1u2 = np.corrcoef(results_iv1.resids, results_iv2.resids)[0, 1]
print(f'cor_u1u2: {cor_u1u2}\n')
```

Script 16.2: `Example-16-5-3SLS.py`

```
import wooldridge as woo
import numpy as np
import linearmodels.system as iv3

mroz = woo.dataWoo('mroz')
```

```
# restrict to non-missing wage observations:
mroz = mroz.dropna(subset=['lwage'])

# 3SLS regressions:
formula = {'eq1': 'hours ~ 1 + educ + age + kidslt6 + nwifeinc +'
                  '[np.log(wage) ~ exper+I(exper**2)]',
           'eq2': 'np.log(wage) ~ 1 + educ + exper + I(exper**2) +'
                  '[hours ~ age + kidslt6 + nwifeinc]'}

reg_3sls = iv3.IV3SLS.from_formula(formula, data=mroz)

results_3sls = reg_3sls.fit(cov_type='unadjusted', debiased=True)
print(f'results_3sls: \n{results_3sls}\n')
```

17. Scripts Used in Chapter 17

Script 17.1: Example-17-1-1.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate linear probability model:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper +'
                          'I(exper**2) + age + kidslt6 + kidsge6',
                  data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

# print regression table:
table = pd.DataFrame({'b': round(results_lin.params, 4),
                      'se': round(results_lin.bse, 4),
                      't': round(results_lin.tvalues, 4),
                      'pval': round(results_lin.pvalues, 4)})
print(f'table: \n{table}\n')
```

Script 17.2: Example-17-1-2.py

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate linear probability model:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper +'
                          'I(exper**2) + age + kidslt6 + kidsge6',
                  data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

# predictions for two "extreme" women:
X_new = pd.DataFrame(
    {'nwifeinc': [100, 0], 'educ': [5, 17],
     'exper': [0, 30], 'age': [20, 52],
     'kidslt6': [2, 0], 'kidsge6': [0, 0]})
predictions = results_lin.predict(X_new)
```

```
print(f'predictions: \n{predictions}\n')
```

Script 17.3: `Example-17-1-3.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate logit model:
reg_logit = smf.logit(formula='inlf ~ nwifeinc + educ + exper +'
                              'I(exper**2) + age + kidslt6 + kidsge6',
                      data=mroz)

# disp = 0 avoids printing out information during the estimation:
results_logit = reg_logit.fit(disp=0)
print(f'results_logit.summary(): \n{results_logit.summary()}\n')

# log likelihood value:
print(f'results_logit.llf: {results_logit.llf}\n')

# McFadden's pseudo R2:
print(f'results_logit.prsquared: {results_logit.prsquared}\n')
```

Script 17.4: `Example-17-1-4.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate probit model:
reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper +'
                                'I(exper**2) + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)
print(f'results_probit.summary(): \n{results_probit.summary()}\n')

# log likelihood value:
print(f'results_probit.llf: {results_probit.llf}\n')

# McFadden's pseudo R2:
print(f'results_probit.prsquared: {results_probit.prsquared}\n')
```

Script 17.5: `Example-17-1-5.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# estimate probit model:
reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper +'
                                'I(exper**2) + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)

# test of overall significance (test statistic and pvalue):
```

```
llr1_manual = 2 * (results_probit.llf - results_probit.llnull)
print(f'llr1_manual: {llr1_manual}\n')
print(f'results_probit.llr: {results_probit.llr}\n')
print(f'results_probit.llr_pvalue: {results_probit.llr_pvalue}\n')

# automatic Wald test of H0 (experience and age are irrelevant):
hypotheses = ['exper=0', 'I(exper ** 2)=0', 'age=0']
waldstat = results_probit.wald_test(hypotheses)
teststat2_autom = waldstat.statistic
pval2_autom = waldstat.pvalue
print(f'teststat2_autom: {teststat2_autom}\n')
print(f'pval2_autom: {pval2_autom}\n')

# manual likelihood ratio statistic test
# of H0 (experience and age are irrelevant):
reg_probit_restr = smf.probit(formula='inlf ~ nwifeinc + educ +'
                                      'kidslt6 + kidsge6',
                              data=mroz)
results_probit_restr = reg_probit_restr.fit(disp=0)

llr2_manual = 2 * (results_probit.llf - results_probit_restr.llf)
pval2_manual = 1 - stats.chi2.cdf(llr2_manual, 3)
print(f'llr2_manual2: {llr2_manual}\n')
print(f'pval2_manual2: {pval2_manual}\n')
```

Script 17.6: `Example-17-1-6.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf

mroz = woo.dataWoo('mroz')

# estimate models:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper +'
                          'I(exper**2) + age + kidslt6 + kidsge6',
                  data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

reg_logit = smf.logit(formula='inlf ~ nwifeinc + educ + exper +'
                              'I(exper**2) + age + kidslt6 + kidsge6',
                      data=mroz)
results_logit = reg_logit.fit(disp=0)

reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper +'
                                'I(exper**2) + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)

# predictions for two "extreme" women:
X_new = pd.DataFrame(
    {'nwifeinc': [100, 0], 'educ': [5, 17],
     'exper': [0, 30], 'age': [20, 52],
     'kidslt6': [2, 0], 'kidsge6': [0, 0]})
predictions_lin = results_lin.predict(X_new)
predictions_logit = results_logit.predict(X_new)
predictions_probit = results_probit.predict(X_new)

print(f'predictions_lin: \n{predictions_lin}\n')
```

```
print(f'predictions_logit: \n{predictions_logit}\n')
print(f'predictions_probit: \n{predictions_probit}\n')
```

Script 17.7: **Binary-Predictions.py**

```
import pandas as pd
import numpy as np
import scipy.stats as stats
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

# set the random seed:
np.random.seed(1234567)

y = stats.binom.rvs(1, 0.5, size=100)
x = stats.norm.rvs(0, 1, size=100) + 2 * y
sim_data = pd.DataFrame({'y': y, 'x': x})

# estimation:
reg_lin = smf.ols(formula='y ~ x', data=sim_data)
results_lin = reg_lin.fit()
reg_logit = smf.logit(formula='y ~ x', data=sim_data)
results_logit = reg_logit.fit(disp=0)
reg_probit = smf.probit(formula='y ~ x', data=sim_data)
results_probit = reg_probit.fit(disp=0)

# prediction for regular grid of x values:
X_new = pd.DataFrame({'x': np.linspace(min(x), max(x), 50)})
predictions_lin = results_lin.predict(X_new)
predictions_logit = results_logit.predict(X_new)
predictions_probit = results_probit.predict(X_new)

# scatter plot and fitted values:
plt.plot(x, y, color='grey', marker='o', linestyle='')
plt.plot(X_new['x'], predictions_lin,
         color='black', linestyle='-.', label='linear')
plt.plot(X_new['x'], predictions_logit,
         color='black', linestyle='-', linewidth=0.5, label='logit')
plt.plot(X_new['x'], predictions_probit,
         color='black', linestyle='--', label='probit')
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/Binary-Predictions.pdf')
```

Script 17.8: **Binary-Margeff.py**

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

y = stats.binom.rvs(1, 0.5, size=100)
x = stats.norm.rvs(0, 1, size=100) + 2 * y
sim_data = pd.DataFrame({'y': y, 'x': x})
```

```
# estimation:
reg_lin = smf.ols(formula='y ~ x', data=sim_data)
results_lin = reg_lin.fit()
reg_logit = smf.logit(formula='y ~ x', data=sim_data)
results_logit = reg_logit.fit(disp=0)
reg_probit = smf.probit(formula='y ~ x', data=sim_data)
results_probit = reg_probit.fit(disp=0)

# calculate partial effects:
PE_lin = np.repeat(results_lin.params['x'], 100)

xb_logit = results_logit.fittedvalues
factor_logit = stats.logistic.pdf(xb_logit)
PE_logit = results_logit.params['x'] * factor_logit

xb_probit = results_probit.fittedvalues
factor_probit = stats.norm.pdf(xb_probit)
PE_probit = results_probit.params['x'] * factor_probit


# plot APE's:
plt.plot(x, PE_lin, color='black',
         marker='o', linestyle='', label='linear')
plt.plot(x, PE_logit, color='black',
         marker='+', linestyle='', label='logit')
plt.plot(x, PE_probit, color='black',
         marker='*', linestyle='', label='probit')
plt.ylabel('partial effects')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/Binary-margeff.pdf')
```

Script 17.9: `Example-17-1-7.py`

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# estimate models:
reg_lin = smf.ols(formula='inlf ~ nwifeinc + educ + exper + I(exper**2) +'
                          'age + kidslt6 + kidsge6', data=mroz)
results_lin = reg_lin.fit(cov_type='HC3')

reg_logit = smf.logit(formula='inlf ~ nwifeinc + educ + exper + I(exper**2) +'
                              'age + kidslt6 + kidsge6', data=mroz)
results_logit = reg_logit.fit(disp=0)

reg_probit = smf.probit(formula='inlf ~ nwifeinc + educ + exper + I(exper**2) +'
                                'age + kidslt6 + kidsge6', data=mroz)
results_probit = reg_probit.fit(disp=0)

# manual average partial effects:
APE_lin = np.array(results_lin.params)

xb_logit = results_logit.fittedvalues
factor_logit = np.mean(stats.logistic.pdf(xb_logit))
```

```
APE_logit_manual = results_logit.params * factor_logit

xb_probit = results_probit.fittedvalues
factor_probit = np.mean(stats.norm.pdf(xb_probit))
APE_probit_manual = results_probit.params * factor_probit

table_manual = pd.DataFrame({'APE_lin': np.round(APE_lin, 4),
                             'APE_logit_manual': np.round(APE_logit_manual, 4),
                             'APE_probit_manual': np.round(APE_probit_manual, 4)})
print(f'table_manual: \n{table_manual}\n')

# automatic average partial effects:
coef_names = np.array(results_lin.model.exog_names)
coef_names = np.delete(coef_names, 0)  # drop Intercept

APE_logit_autom = results_logit.get_margeff().margeff
APE_probit_autom = results_probit.get_margeff().margeff

table_auto = pd.DataFrame({'coef_names': coef_names,
                           'APE_logit_autom': np.round(APE_logit_autom, 4),
                           'APE_probit_autom': np.round(APE_probit_autom, 4)})
print(f'table_auto: \n{table_auto}\n')
```

Script 17.10: `Example-17-3.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

crime1 = woo.dataWoo('crime1')

# estimate linear model:
reg_lin = smf.ols(formula='narr86 ~ pcnv + avgsen + tottime + ptime86 +'
                          'qemp86 + inc86 + black + hispan + born60',
                  data=crime1)
results_lin = reg_lin.fit()

# print regression table:
table_lin = pd.DataFrame({'b': round(results_lin.params, 4),
                          'se': round(results_lin.bse, 4),
                          't': round(results_lin.tvalues, 4),
                          'pval': round(results_lin.pvalues, 4)})
print(f'table_lin: \n{table_lin}\n')

# estimate Poisson model:
reg_poisson = smf.poisson(formula='narr86 ~ pcnv + avgsen + tottime +'
                                  'ptime86 + qemp86 + inc86 + black +'
                                  'hispan + born60',
                          data=crime1)
results_poisson = reg_poisson.fit(disp=0)

# print regression table:
table_poisson = pd.DataFrame({'b': round(results_poisson.params, 4),
                              'se': round(results_poisson.bse, 4),
                              't': round(results_poisson.tvalues, 4),
                              'pval': round(results_poisson.pvalues, 4)})
print(f'table_poisson: \n{table_poisson}\n')

# estimate Quasi-Poisson model:
```

```
reg_qpoisson = smf.glm(formula='narr86 ~ pcnv + avgsen + tottime + ptime86 +'
                               'qemp86 + inc86 + black + hispan + born60',
                       family=sm.families.Poisson(),
                       data=crime1)
# the argument scale controls for the dispersion in exponential dispersion models,
# see the module documentation for more details:
results_qpoisson = reg_qpoisson.fit(scale='X2', disp=0)

# print regression table:
table_qpoisson = pd.DataFrame({'b': round(results_qpoisson.params, 4),
                               'se': round(results_qpoisson.bse, 4),
                               't': round(results_qpoisson.tvalues, 4),
                               'pval': round(results_qpoisson.pvalues, 4)})
print(f'table_qpoisson: \n{table_qpoisson}\n')
```

Script 17.11: `Tobit-CondMean.py`

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

x = np.sort(stats.norm.rvs(0, 1, size=100) + 4)
xb = -4 + 1 * x
y_star = xb + stats.norm.rvs(0, 1, size=100)
y = np.copy(y_star)
y[y_star < 0] = 0

# conditional means:
Eystar = xb
Ey = stats.norm.cdf(xb / 1) * xb + 1 * stats.norm.pdf(xb / 1)

# plot data and conditional means:
plt.axhline(y=0, linewidth=0.5,
            linestyle='-', color='grey')
plt.plot(x, y_star, color='black',
         marker='x', linestyle='', label='y*')
plt.plot(x, y, color='black', marker='+',
         linestyle='', label='y')
plt.plot(x, Eystar, color='black', marker='',
         linestyle='-', label='E(y*)')
plt.plot(x, Ey, color='black', marker='',
         linestyle='--', label='E(y)')
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/Tobit-CondMean.pdf')
```

Script 17.12: `Example-17-2.py`

```
import wooldridge as woo
import numpy as np
import patsy as pt
import scipy.stats as stats
import statsmodels.formula.api as smf
import statsmodels.base.model as smclass

mroz = woo.dataWoo('mroz')
```

```
y, X = pt.dmatrices('hours ~ nwifeinc + educ + exper +'
                    'I(exper**2)+ age + kidslt6 + kidsge6',
                    data=mroz, return_type='dataframe')

# generate starting solution:
reg_ols = smf.ols(formula='hours ~ nwifeinc + educ + exper + I(exper**2) +'
                          'age + kidslt6 + kidsge6', data=mroz)
results_ols = reg_ols.fit()
sigma_start = np.log(sum(results_ols.resid ** 2) / len(results_ols.resid))
params_start = np.concatenate((np.array(results_ols.params), sigma_start),
                              axis=None)

# extend statsmodels class by defining nloglikeobs:
class Tobit(smclass.GenericLikelihoodModel):
    # define a function that returns the negative log likelihood per observation
    # for a set of parameters that is provided by the argument "params":
    def nloglikeobs(self, params):
        # objects in "self" are defined in the parent class:
        X = self.exog
        y = self.endog
        p = X.shape[1]
        # for details on the implementation see Wooldridge (2019), formula 17.22:
        beta = params[0:p]
        sigma = np.exp(params[p])
        y_hat = np.dot(X, beta)
        y_eq = (y == 0)
        y_g = (y > 0)
        ll = np.empty(len(y))
        ll[y_eq] = np.log(stats.norm.cdf(-y_hat[y_eq] / sigma))
        ll[y_g] = np.log(stats.norm.pdf((y - y_hat)[y_g] / sigma)) - np.log(sigma)
        # return an array of log likelihoods for each observation:
        return -ll

# results of MLE:
reg_tobit = Tobit(endog=y, exog=X)
results_tobit = reg_tobit.fit(start_params=params_start, maxiter=10000, disp=0)
print(f'results_tobit.summary(): \n{results_tobit.summary()}\n')
```

Script 17.13: Example-17-4.py

```
import wooldridge as woo
import numpy as np
import patsy as pt
import scipy.stats as stats
import statsmodels.formula.api as smf
import statsmodels.base.model as smclass

recid = woo.dataWoo('recid')

# define dummy for censored observations:
censored = recid['cens'] != 0
y, X = pt.dmatrices('ldurat ~ workprg + priors + tserved + felon +'
                    'alcohol + drugs + black + married + educ + age',
                    data=recid, return_type='dataframe')

# generate starting solution:
reg_ols = smf.ols(formula='ldurat ~ workprg + priors + tserved + felon +'
                          'alcohol + drugs + black + married + educ + age',
                  data=recid)
results_ols = reg_ols.fit()
```

```
sigma_start = np.log(sum(results_ols.resid ** 2) / len(results_ols.resid))
params_start = np.concatenate((np.array(results_ols.params), sigma_start),
                              axis=None)


# extend statsmodels class by defining nloglikeobs:
class CensReg(smclass.GenericLikelihoodModel):
    def __init__(self, endog, cens, exog):
        self.cens = cens
        super(smclass.GenericLikelihoodModel, self).__init__(endog, exog,
                                                             missing='none')

    def nloglikeobs(self, params):
        X = self.exog
        y = self.endog
        cens = self.cens
        p = X.shape[1]
        beta = params[0:p]
        sigma = np.exp(params[p])
        y_hat = np.dot(X, beta)
        ll = np.empty(len(y))
        # uncensored:
        ll[~cens] = np.log(stats.norm.pdf((y - y_hat)[~cens] /
                                          sigma)) - np.log(sigma)
        # censored:
        ll[cens] = np.log(stats.norm.cdf(-(y - y_hat)[cens] / sigma))
        return -ll


# results of MLE:
reg_censReg = CensReg(endog=y, exog=X, cens=censored)
results_censReg = reg_censReg.fit(start_params=params_start,
                                  maxiter=10000, method='BFGS', disp=0)
print(f'results_censReg.summary(): \n{results_censReg.summary()}\n')
```

Script 17.14: `TruncReg-Simulation.py`

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(1234567)

x = np.sort(stats.norm.rvs(0, 1, size=100) + 4)
y = -4 + 1 * x + stats.norm.rvs(0, 1, size=100)

# complete observations and observed sample:
compl = pd.DataFrame({'x': x, 'y': y})
sample = compl.loc[y > 0]

# predictions OLS:
reg_ols = smf.ols(formula='y ~ x', data=sample)
results_ols = reg_ols.fit()
yhat_ols = results_ols.fittedvalues

# predictions truncated regression:
reg_tr = smf.ols(formula='y ~ x', data=compl)
```

```
results_tr = reg_tr.fit()
yhat_tr = results_tr.fittedvalues

# plot data and conditional means:
plt.axhline(y=0, linewidth=0.5, linestyle='-', color='grey')
plt.plot(compl['x'], compl['y'], color='black',
         marker='o', fillstyle='none', linestyle='', label='all data')
plt.plot(sample['x'], sample['y'], color='black',
         marker='o', fillstyle='full', linestyle='', label='sample data')
plt.plot(sample['x'], yhat_ols, color='black',
         marker='', linestyle='--', label='OLS fit')
plt.plot(compl['x'], yhat_tr, color='black',
         marker='', linestyle='-', label='Trunc. Reg. fit')
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('PyGraphs/TruncReg-Simulation.pdf')
```

Script 17.15: `Example-17-5.py`

```
import wooldridge as woo
import statsmodels.formula.api as smf
import scipy.stats as stats

mroz = woo.dataWoo('mroz')

# step 1 (use all n observations to estimate a probit model of s_i on z_i):
reg_probit = smf.probit(formula='inlf ~ educ + exper + I(exper**2) +'
                                'nwifeinc + age + kidslt6 + kidsge6',
                        data=mroz)
results_probit = reg_probit.fit(disp=0)
pred_inlf = results_probit.fittedvalues
mroz['inv_mills'] = stats.norm.pdf(pred_inlf) / stats.norm.cdf(pred_inlf)

# step 2 (regress y_i on x_i and inv_mills in sample selection):
reg_heckit = smf.ols(formula='lwage ~ educ + exper + I(exper**2) + inv_mills',
                     subset=(mroz['inlf'] == 1), data=mroz)
results_heckit = reg_heckit.fit()

# print results:
print(f'results_heckit.summary(): \n{results_heckit.summary()}\n')
```

18. Scripts Used in Chapter 18

Script 18.1: `Example-18-1.py`

```
import wooldridge as woo
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm

hseinv = woo.dataWoo('hseinv')

# add lags and detrend:
hseinv['linvpc_det'] = sm.tsa.tsatools.detrend(hseinv['linvpc'])
hseinv['gprice_lag1'] = hseinv['gprice'].shift(1)
hseinv['linvpc_det_lag1'] = hseinv['linvpc_det'].shift(1)
```

```
# Koyck geometric d.l.:
reg_koyck = smf.ols(formula='linvpc_det ~ gprice + linvpc_det_lag1',
                    data=hseinv)
results_koyck = reg_koyck.fit()

# print regression table:
table_koyck = pd.DataFrame({'b': round(results_koyck.params, 4),
                            'se': round(results_koyck.bse, 4),
                            't': round(results_koyck.tvalues, 4),
                            'pval': round(results_koyck.pvalues, 4)})
print(f'table_koyck: \n{table_koyck}\n')

# rational d.l.:
reg_rational = smf.ols(formula='linvpc_det ~ gprice + linvpc_det_lag1 +'
                               'gprice_lag1',
                       data=hseinv)
results_rational = reg_rational.fit()

# print regression table:
table_rational = pd.DataFrame({'b': round(results_rational.params, 4),
                               'se': round(results_rational.bse, 4),
                               't': round(results_rational.tvalues, 4),
                               'pval': round(results_rational.pvalues, 4)})
print(f'table_rational: \n{table_rational}\n')

# LRP:
lrp_koyck = results_koyck.params['gprice'] / (
        1 - results_koyck.params['linvpc_det_lag1'])
print(f'lrp_koyck: {lrp_koyck}\n')

lrp_rational = (results_rational.params['gprice'] +
                results_rational.params['gprice_lag1']) / (
                       1 - results_rational.params['linvpc_det_lag1'])
print(f'lrp_rational: {lrp_rational}\n')
```

Script 18.2: `Example-18-4.py`

```
import wooldridge as woo
import numpy as np
import pandas as pd
import statsmodels.api as sm

inven = woo.dataWoo('inven')
inven['lgdp'] = np.log(inven['gdp'])

# automated ADF:
res_ADF_aut = sm.tsa.stattools.adfuller(inven['lgdp'], maxlag=1, autolag=None,
                                        regression='ct', regresults=True)
ADF_stat_aut = res_ADF_aut[0]
ADF_pval_aut = res_ADF_aut[1]
table = pd.DataFrame({'names': res_ADF_aut[3].resols.model.exog_names,
                      'b': np.round(res_ADF_aut[3].resols.params, 4),
                      'se': np.round(res_ADF_aut[3].resols.bse, 4),
                      't': np.round(res_ADF_aut[3].resols.tvalues, 4),
                      'pval': np.round(res_ADF_aut[3].resols.pvalues, 4)})
print(f'table: \n{table}\n')
print(f'ADF_stat_aut: {ADF_stat_aut}\n')
print(f'ADF_pval_aut: {ADF_pval_aut}\n')
```

Script 18.3: `Simulate-Spurious-Regression-1.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

# i.i.d. N(0,1) innovations:
n = 51
e = stats.norm.rvs(0, 1, size=n)
e[0] = 0
a = stats.norm.rvs(0, 1, size=n)
a[0] = 0

# independent random walks:
x = np.cumsum(a)
y = np.cumsum(e)
sim_data = pd.DataFrame({'y': y, 'x': x})

# regression:
reg = smf.ols(formula='y ~ x', data=sim_data)
results = reg.fit()

# print regression table:
table = pd.DataFrame({'b': round(results.params, 4),
                      'se': round(results.bse, 4),
                      't': round(results.tvalues, 4),
                      'pval': round(results.pvalues, 4)})
print(f'table: \n{table}\n')

# graph:
plt.plot(x, color='black', marker='', linestyle='-', label='x')
plt.plot(y, color='black', marker='', linestyle='--', label='y')
plt.ylabel('x,y')
plt.legend()
plt.savefig('PyGraphs/Simulate-Spurious-Regression-1.pdf')
```

Script 18.4: `Simulate-Spurious-Regression-2.py`

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf
import scipy.stats as stats

# set the random seed:
np.random.seed(123456)

pvals = np.empty(10000)

# repeat r times:
for i in range(10000):
    # i.i.d. N(0,1) innovations:
    n = 51
    e = stats.norm.rvs(0, 1, size=n)
    e[0] = 0
    a = stats.norm.rvs(0, 1, size=n)
    a[0] = 0
```

```
    # independent random walks:
    x = np.cumsum(a)
    y = np.cumsum(e)
    sim_data = pd.DataFrame({'y': y, 'x': x})

    # regression:
    reg = smf.ols(formula='y ~ x', data=sim_data)
    results = reg.fit()
    pvals[i] = results.pvalues['x']

# how often is p<=5%:
count_pval_smaller = np.count_nonzero(pvals <= 0.05)  # counts True elements
print(f'count_pval_smaller: {count_pval_smaller}\n')

# how often is p>5%:
count_pval_greater = np.count_nonzero(pvals > 0.05)
print(f'count_pval_greater: {count_pval_greater}\n')
```

Script 18.5: Example-18-8.py

```
import wooldridge as woo
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import matplotlib.pyplot as plt

phillips = woo.dataWoo('phillips')

# define yearly time series beginning in 1948:
date_range = pd.date_range(start='1948', periods=len(phillips), freq='Y')
phillips.index = date_range.year

# estimate models:
yt96 = (phillips['year'] <= 1996)
reg_1 = smf.ols(formula='unem ~ unem_1', data=phillips, subset=yt96)
results_1 = reg_1.fit()
reg_2 = smf.ols(formula='unem ~ unem_1 + inf_1', data=phillips, subset=yt96)
results_2 = reg_2.fit()

# predictions for 1997-2003 including 95% forecast intervals:
yf97 = (phillips['year'] > 1996)
pred_1 = results_1.get_prediction(phillips[yf97])
pred_1_FI = pred_1.summary_frame(
    alpha=0.05)[['mean', 'obs_ci_lower', 'obs_ci_upper']]
pred_1_FI.index = date_range.year[yf97]
print(f'pred_1_FI: \n{pred_1_FI}\n')

pred_2 = results_2.get_prediction(phillips[yf97])
pred_2_FI = pred_2.summary_frame(
    alpha=0.05)[['mean', 'obs_ci_lower', 'obs_ci_upper']]
pred_2_FI.index = date_range.year[yf97]
print(f'pred_2_FI: \n{pred_2_FI}\n')

# forecast errors:
e1 = phillips[yf97]['unem'] - pred_1_FI['mean']
e2 = phillips[yf97]['unem'] - pred_2_FI['mean']

# RMSE and MAE:
rmse1 = np.sqrt(np.mean(e1 ** 2))
```

```
print(f'rmse1: {rmse1}\n')
rmse2 = np.sqrt(np.mean(e2 ** 2))
print(f'rmse2: {rmse2}\n')
mae1 = np.mean(abs(e1))
print(f'mae1: {mae1}\n')
mae2 = np.mean(abs(e2))
print(f'mae2: {mae2}\n')

# graph:
plt.plot(phillips[yf97]['unem'], color='black', marker='', label='unem')
plt.plot(pred_1_FI['mean'], color='black',
         marker='', linestyle='--', label='forecast without inflation')
plt.plot(pred_2_FI['mean'], color='black',
         marker='', linestyle='-.', label='forecast with inflation')
plt.ylabel('unemployment')
plt.xlabel('time')
plt.legend()
plt.savefig('PyGraphs/Example-18-8.pdf')
```

19. Scripts Used in Chapter 19

Script 19.1: `ultimate-calcs.py`

```
########################################################################
# Project X:
# "The Ultimate Question of Life, the Universe, and Everything"
# Project Collaborators: Mr. X, Mrs. Y
#
# Python Script "ultimate-calcs"
# by: F Heiss
# Date of this version: February 18, 2019
########################################################################
# external modules:
import numpy as np
import datetime as dt

# create a time stamp:
ts = dt.datetime.now()

# print to logfile.txt ('w' resets the logfile before writing output)
# in the provided path (make sure that the folder structure
# you may provide already exists):
print(f'This is a log file from: \n{ts}\n',
      file=open('Pyout/19/logfile.txt', 'w'))

# the first calculation using the function "square root" from numpy:
result1 = np.sqrt(1764)

# print to logfile.txt but with keeping the previous results ('a'):
print(f'result1: {result1}\n',
      file=open('Pyout/19/logfile.txt', 'a'))

# the second calculation reverses the first one:
result2 = result1 ** 2

# print to logfile.txt but with keeping the previous results ('a'):
```

```
print(f'result2: {result2}',
      file=open('Pyout/19/logfile.txt', 'a'))
```

Script 19.2: `ultimate-calcs2.py`

```
# external modules:
import numpy as np
import datetime as dt
import sys

# make sure that the folder structure you may provide already exists:
sys.stdout = open('Pyout/19/logfile2.txt', 'w')

# create a time stamp:
ts = dt.datetime.now()

# print to logfile2.txt:
print(f'This is a log file from: \n{ts}\n')

# the first calculation using the function "square root" from numpy:
result1 = np.sqrt(1764)

# print to logfile2.txt:
print(f'result1: {result1}\n')

# the second calculation reverses the first one:
result2 = result1 ** 2

# print to logfile2.txt:
print(f'result2: {result2}')
```

Bibliography

BARRY, P. (2016): *Head First Python: A Brain-Friendly Guide*, O'Reilly Media.

DOWNEY, A. (2015): *Think Python: How to Think Like a Computer Scientist*, O'Reilly UK.

GUIDO, S. AND A. MUELLER (2016): *Introduction to Machine Learning with Python: A Guide for Data Scientists*, O'Reilly UK.

HEISS, F. (2020): *Using R for Introductory Econometrics*, CreateSpace Independent Publishing Platform, 2 ed.

HUNTER, J. D. (2007): "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, 9, 90–95.

KLUYVER, T., B. RAGAN-KELLEY, F. PÉREZ, B. GRANGER, M. BUSSONNIER, J. FREDERIC, K. KELLEY, J. HAMRICK, J. GROUT, S. CORLAY, P. IVANOV, D. AVILA, S. ABDALLA, C. WILLING, AND J. DEVELOPMENT TEAM (2016): "Jupyter Notebooks ? a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, ed. by F. Loizides and B. Scmidt, IOS Press, 87–90.

MATTHES, E. (2015): *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, No Starch Press.

MCKINNEY, W. (2011): "pandas: a foundational Python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, 14.

OLIPHANT, T. E. (2007): "Python for Scientific Computing," *Computing in Science & Engineering*, 9, 10–20.

SEABOLD, S. AND J. PERKTOLD (2010): "Statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*.

SILVERMAN, B. W. (1986): *Density Estimation for Statistics and Data Analysis*, Chapman & Hall.

VIRTANEN, P., R. GOMMERS, T. E. OLIPHANT, M. HABERLAND, T. REDDY, D. COURNAPEAU, E. BUROVSKI, P. PETERSON, W. WECKESSER, J. BRIGHT, S. J. VAN DER WALT, M. BRETT, J. WILSON, K. JARROD MILLMAN, N. MAYOROV, A. R. J. NELSON, E. JONES, R. KERN, E. LARSON, C. CAREY, İ. POLAT, Y. FENG, E. W. MOORE, J. VAND ERPLAS, D. LAXALDE, J. PERKTOLD, R. CIMRMAN, I. HENRIKSEN, E. A. QUINTERO, C. R. HARRIS, A. M. ARCHIBALD, A. H. RIBEIRO, F. PEDREGOSA, P. VAN MULBREGT, AND CONTRIBUTORS (2020): "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, 17, 261–272.

WALT, S. V. D., S. C. COLBERT, AND G. VAROQUAUX (2011): "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, 13, 22–30.

WOOLDRIDGE, J. M. (2010): *Econometric Analysis of Cross Section and Panel Data*, MIT Press.

——— (2014): *Introduction to Econometrics*, Cengage Learning.

——— (2019): *Introductory Econometrics: A Modern Approach*, Cengage Learning, 7th ed.

List of Wooldridge (2019) Examples

Index