Doing Computational Social Science with Python: An Introduction

Damian Trilling

Version 0.99

PDF created January 3, 2017

Contents

Introduction: What and why?

Social scientists are more and more confronted with the analysis of large-scale datasets. Often, these are data from online sources, and often, they contain some form of textual data. Think of data from social media, but also large archives. Often, this development if referred to as a move towards "computational social science" (Kitchin, 2014; Lazer et al., 2009).

There is a certain overlap with the term "Big Data". But although the latter sounds sexy, it is a somewhat problematic term, because people use it for all kind of data. As scientists, we want a clear definition, but it is hard to tell what the term Big Data actually entails. This document is not about *really* Big Data requiring a whole server farm, but it is about data that is too big to handle manually—think of one million tweets for example. It is about data sets that are small enough to be handled by an ordinary desktop computer, but too big to be processed by ordinary programs. Excel does not have an unlimited number of rows, and SPSS and STATA start complaining (or simply stop working) once you have too many cases and variables. If you know R, you are better off, but for some of the tasks we will discuss in this document, Python has much more to offer.[1]

This document introduces you to automated content analysis of data that typically comes in amounts that are too voluminous for traditional point-and-click applications: Tweets, blogposts, articles from RSS-feeds, etc. We will use the programming language Python, which is very flexible and highly suitable for this end. It also scales very nicely—meaning that you can use it now for some smaller projects, but it can also be used on immense data sets. So, if you will find yourself working on some really Big Data that cannot be handled on a single computer any more, the principles are not too different from what we do in this document (yes, I'm simplifying a bit).

You will be guided through the first steps towards automated content analysis with Python. Note that I wrote this document for an audience of students in the social sciences, communication science in particular. From a

---

[1]I do not want to go into the R-vs-Python-for-data-analysis debate here, I use both tools a lot, but for completely different tasks. Google for it if you want some comparison.

computer science point of view, much more would have to be said, and some things we do in this course might actually be considered bad programming style. But that's not our objective here: This course should enable students of the social science to make some first steps with Python, in order to solve their analytical questions.

This also means that this manual is far from complete. It rather serves as a starting point—and from that point onwards, it's up to you. Once you got the basic ideas and concepts, there are enough resources out there to help you further.

Have fun! And, join me in thanking those who contributed with great ideas or served as guinea pig for earlier versions of this document – which basically applies to all students and colleagues working with earlier versions of this document.

<div align="right">

Amsterdam, March 2015 (version 0.1)
Amsterdam, February 2016 (version 0.2)
Amsterdam, November 2016 (version 1.0)

</div>

Damian

<div align="right">

Damian Trilling
d.c.trilling@uva.nl
www.damiantrilling.net
@damian0604

</div>

Chapter 1

Preparing your computer

This chapter describes how to prepare your computer for use in this course. We work with so-called virtual machines, which means that we install a program within which you can install an own operating system. This means that

(a) you can play around without the danger of damaging your own system;

(b) you can make use of a lot of cool tools that might not be available for your own system;

(c) everyone in this course has exactly the same environment running.

Specifically, we are using Oracle VirtualBox, within which we will install Lubuntu (a lightweight version of Ubuntu, a user-friendly Linux distribution). Before you start, make sure that you have plenty of space available on your laptop, preferably more than 10 GB. (You'd better put your music and photos on an external drive for the duration of this course...)

Before we start, let me introduce three conventions that I'll use in this book. Python-code, which you will type into the Python interpreter or use in the programs you'll write, is represented in blue:

```
1  print("Hello world"!)
```

Commands that you have to type into the Linux command line (also referred to as shell, terminal, or bash) are pink:

```
1  echo Hello world!
```

And any output or data file is represented in gray:

```
1  Hello world!
```

Ready to go?

1.1   Install VirtualBox

Download and install VirtualBox from `https://www.virtualbox.org/`. Depending on your computer, download the binary for Windows, MacOS, or Linux.

1.2   Download a Lubuntu-image

Download the Lubuntu image from `http://lubuntu.net`. **We are working with version 16.04.1 LTS (Xenial Xerus). It is important that you do *not* use version 16.10.** The download link for this version should be `http://cdimages.ubuntu.com/lubuntu/releases/xenial/release/` If you have a relatively new computer, download `lubuntu 64-bit (AMD64) desktop CD`, only if you encounter problems later on or have an old computer (up to around 2007), download the 32-bit version. So, in most cases, select the 64-bit version. Save it at a place of your convenience.

1.3   Create a virtual machine

Open VirtualBox, click on "New", give your image a name, and select first Linux, then Ubuntu 64bit from the list (or 32 bit, if that's what you downloaded). On the following screens, just leave the default settings and agree to everything—with one exception: When you are asked how much "base memory" you want to allocate to your machine, you should choose a value of at the *very* least around 1000 MB. More is better, if you you have 4,096 MB of RAM in total, you can safely choose 2,048 MB. The examples in this tutorial are tested on a VM with 2,048 MB RAM.

On the next screens, you are asked about a virtual hard disk to create. I strongly recommend you to use a bit more than the 8 GB that are suggested. If you choose more (say, 12 GB or so), chances are lower that you will run out of space later during the course.

> **!** The virtual machine you just created should show up in the main window now. If it doesn't (I had that on one computer once), restart VirtualBox.

Now click on the name of the virtual machine and on "Settings" (Figure 1.1. Under "System"/"Processors", you should see a check box "Enable PAE/NX". If you can enable this, enable it.[1] If you have more than one

---

[1]In rare cases, you might just to have to disable it. So, if you cannot start your VM, you know what to try.
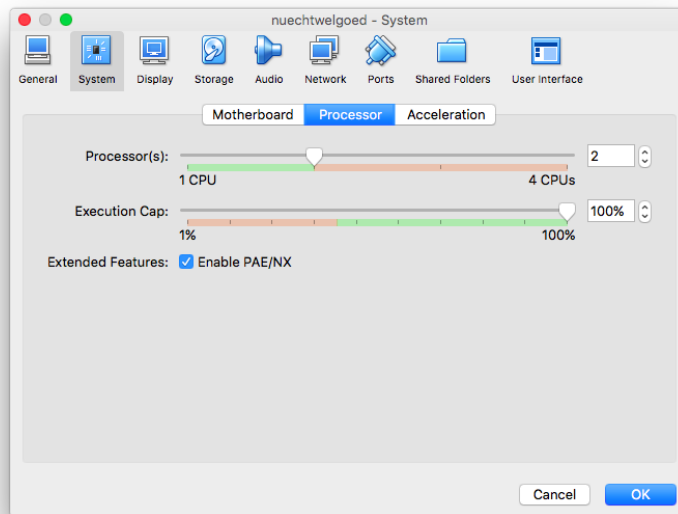
Figure 1.1: Possibly necessary tweaks: Enabling PAE/NX and allowing the use of multiple CPUs in the settings dialogue

processor, also select the maximum (e.g., choose 2 instead of 1). Confirm with OK. Also, check if you can enable "Display"/"Enable 3D acceleation" (Figure 1.2). These changes might not be strictly necessary, but can in some instances prevent problems.

When you are done, start the virtual machine by clicking on the green arrow in the main window. You will be asked for a location from which to boot. Click on the icon to select a file and select the Lubuntu-image you downloaded.

1.4 Install Lubuntu on your virtual machine

Once your virtual machine is running, you can select whether you want to try out Lubuntu without installing or install it. If you feel like, you can now just play around a bit in the try-out mode, but eventually, you'll need to go for installing as we need some specific tools for our course. So you can as well just select Install now, using the arrow keys. When you select "Install", just follow the instructions. At one point in time, you will be asked whether you want to erase all data from your hard drive (yes, seriously). You can safely agree (remember, that's just the point of the VirtualBox, you have a fenced room in which you cannot damage anything. Only your *virtual* hard
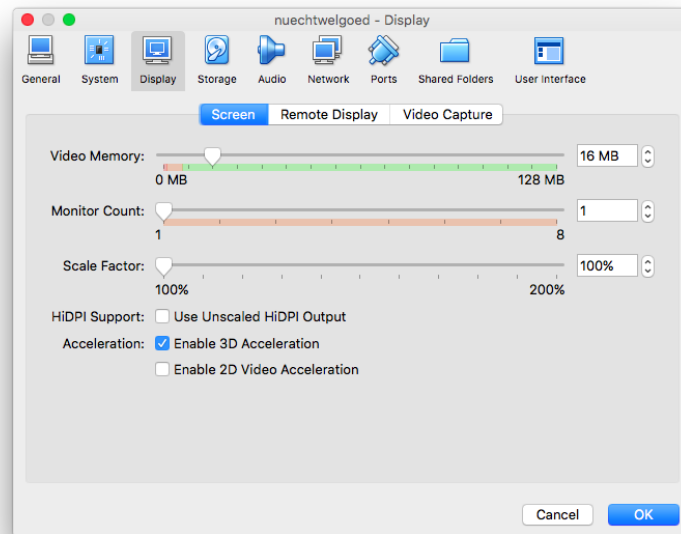
Figure 1.2: Possibly necessary tweaks: Enabling 3D and/or 2D acceleration in the settings dialogue

drive will be erased, not your real one). After finishing the installation, your virtual machine will reboot. After clicking on reboot, you can get a message saying "please remove installation disks". You can just press enter, you don't have to remove anything in your case. Also, it might happen that you see only some lines of text saying something like "Unmounting crypto disks". If this takes longer than a minute or so, try pressing Enter twice.

1.5   Install necessary packages

Once you have installed and rebooted your virtual machine, you will have to install a couple of packages. It might be that your system is running now in a very low resulution, but that will change after the following step. Click on the button on the bottom left (where the traditional Windows start button would be), then on System Tools/LXTerminal. Enter the following command:

```
1  sudo apt-get install virtualbox-guest-dkms virtualbox-guest-utils
       virtualbox-guest-x11
```

You will be asked for your password (the one you defined during the install process). Just type it and confirm with Enter (no, you do not see anything

while you type, that's correct). If you reboot your virtual machine (by clicking on the Start button/logout/reboot or by typing

```
1  sudo reboot
```

it will display in a higher resolution and you can even run it full screen if you want to (by clicking on View/Switch to Full Screen in the VirtualBox menu).

We will now install some other programs we will use with the following commands. Just type them into the LX Terminal, which you already know:

```
1  sudo apt-get install gedit spyder3 python3-dev python3-pip
```

You will be asked for your password and some questions which have to be answered with yes (by typing the letter y) followed by Enter.

Let us also update everything to the latest version:

```
1  sudo apt-get update
2  sudo apt-get upgrade
```

And let us finally install some additional Python modules:

```
1  sudo pip3 install --upgrade pip
2  sudo pip3 install nltk
3  sudo pip3 install scikit-learn
4  sudo pip3 install pandas
5  sudo pip3 install statsmodels
6  sudo pip3 install jupyter
7  sudo pip3 install gensim
```

In the start menu under "Education" you now find a new program, Spyder 3. This is the tool we will be using for doing some Python programming. But of course, you can also start it from the command line by just typing:

```
1  spyder3 &
```

> **!** While working in the VM, you realize that you made a mistake while configuring your keyboard so that some keys don't work as expected? No worries, you can re-configure your keyboard at any time with the following command:
> `sudo dpkg-reconfigure keyboard-configuration`
> If you have a standard laptop purchased in the Netherlands, it is very likely that you have a US keyboard. In that case, the following configuration is correct:
> Model: Algemeen 105 toetsen internationaal
> Oorsprong van het toetsenbord (Engels VS)
> Indeling: Engels (VS)
> standaard
> geen samenstelling
> X-server: nee

1.6 Recap

Make sure you installed *everything* correctly. If you got any error messages, try to find out what went wrong or ask your instructor. In particular, make sure you write down *what* went wrong.

There are so many different system types (and believe me, I never expected *how* many slightly different systems there are until I had dozens of students in these methods courses!), that we cannot cover every eventuality. But once you got your VM is up and running, everything should be the same for everyone in the course.

Chapter 2

The Linux command line

2.1 Getting to know it

This chapter provides you with some basic knowledge about how to use the command line of your operating system. Chances are very high that you will need this later, so before diving into Python, I recommend doing this first.

Figure 2.1: The old MS-DOS command line prompt

Let's get started with a generational divide. It greatly helps if you are of the age of the author (I'm born in 1983) or older, as you probably remember the thing in Figure 2.1. The Linux command line (which is basically the same as what you get when you start the Terminal on MacOS, so Mac users, you can use this on your own computer as well!), however, is much, much, much, much, much more powerful than the old MS-DOS thing. If you look on YouTube for "bash" (that's the specific one we are using) or "linux command line", you'll probably get some good tutorials, but let's explore the most basic things together.

- Open a Terminal by clicking on the Start button/System Tools/LX-Terminal.

- Type the following commands and watch what is happening:

```
1   pwd
2   mkdir test
3   cd test
4   pwd
5   ls
6   echo Hello world > test.txt
7   ls
8   cat test.txt
9   rm test.txt
10  cd ..
```

What happend? Let's discuss it line by line:

1. `pwd` means "print working directory". It asks the computer to tell you where you are, and you probably got some answer like `/home/damian`, which is your so-called home-directory. Directory is just another word for what you might know as a folder.

2. `mkdir` means "make directory" and creates a subdirectory.

3. `cd` means "change directory" and let you go to the directory in question.

4. . . . which, if you don't believe that you have really gone there, can confirm again with `pwd`.

5. `ls` stands for "list" and shows you all files in the directory. Obviously, there isn't a file yet, so. . .

6. . . . let's simply create one. The >-sign basically means that the output of a specific command should be saved under the following filename.

7. Now it's there.

8. Let's have a look at the file,

9. remove it again,

10. and finally go the parent directory (i.e., one level up), so that we are back at where we started.

2.2 Some useful commands for inspecting data

You now already learned how to create and change directories, list the files in a given directory, and how to delete them again. And, of course, in Chapter 1, you used it to install and start programs. One of the really useful things you can do in the command line is inspecting (also very large) files. Imagine you have millions of tweets, then you do not want to load all of them in a program, but maybe only see the first few to get a general idea.

Let's do a small exercise. Lets create a new directory for this exercise, download a training dataset, and have a look at it. I assume that you are in your home directory, you can check that with `pwd`.

```
1  mkdir excercise1
2  cd excercise1
3  wget datacollection.followthenews-uva.vm.surfsara.nl/bdaca/mytweets.csv --
      user=USERNAME --password=PASSWORD
```

(Of course, you replace USERNAME and PASSWORD by the username and password you received from your instructor to access all class materials.[1] Also note that they are entered in the same line as the rest of the **wget**-command. **And note the space before the `--`!**).

You could as well just have downloaded this file by typing exactly the same URL in your browser, but `wget` allows you to do the same trick from the command line. If you have a really huge file, you'll appreciate this! Later on, you will learn some really cool stuff you can do with wget.

The general scheme of all following commands is that you type the command followed by the file name that you want to view.

You have four tools at your disposal: `head` (which gives you the first lines of a file), `tail` (last lines), `cat` (all lines), `less` (all lines, but with scrolling). By the way, if something goes wrong, you can always cancel by pressing CTRL-C. If you want to learn more about a command like `tail`, just type `man tail`. By default, `head` and `tail` give you 10 lines, but you can specify a different number, thus, you can produce the output below with `head -3 mytweets.csv`.

```
1  text,to_user_id,from_user,id,from_user_id,iso_language_code,source,
      profile_image_url,geo_type,geo_coordinates_0,geo_coordinates_1,
      created_at,time
2  :-) #Lectrr #wereldleiders #uitspraken #Wikileaks #klimaattop http://t.co/
      Udjpk48EIB,,henklbr,407085917011079169,118374840,nl,web,http://pbs.
      twimg.com/profile_images/378800000673845195/
```

[1]Unfortunately, Twitter does not allow to publish datasets with tweets. Therefore, I cannot make the dataset publicly available to all readers of this book (which luckily *is* available to every interested reader).

```
     b47785b1595e6a1c63b93e463f3d0ccc_normal.jpeg,,0,0,Sun Dec 01 09:57:00
     +0000 2013,1385891820
3  Wat zijn de resulaten vd #klimaattop in #Warschau waard? @EP_Environment
     ontmoet voorzitter klimaattop @MarcinKorolec http://t.co/4Lmiaopf60,,
     Europarl_NL,406058792573730816,37623918,en,<a href="http://www.
     hootsuite.com" rel="nofollow">HootSuite</a>,http://pbs.twimg.com/
     profile_images/2943831271/b6631b23a86502fae808ca3efde23d0d_normal.png
     ,,0,0,Thu Nov 28 13:55:35 +0000 2013,1385646935
```

This is a so-called CSV file, a table in which each column is separated
by a comma. The first column contains the tweet, the second the user that
the tweet is directed at (which obviously can be empty), the third the sender
and so on.

But it gets even cooler: You can also just display a specific column (e.g.,
only the tweets): cut -f1 -d, mytweets.csv. See man cut for more details.
And you can combine these kind of commands through a technique called
"pipe", indicated by the |-sign. The command

```
1  cut -f1 -d, mytweets.csv | tail -5
```

for example sends the output of the cut-command to the tail-command. If
you want to send output to a file instead of the screen, this is done with the
>-sign (remember?). For example, try this cool pipe:

```
1  cut -f1 -d, mytweets.csv > mytweetsonlyfirstcolumn.csv
```

Now, play a bit around with the commands you learned!

2.3   Recap

In this chapter, you learned some basic command line commands. You should
understand how the following commands work:

- pwd

- ls

- cd

- mkdir

- man

- rm

- cat

- head

- tail

- wget

You will encounter these over and over again, so you should have understood the general concept (`cut` was a pretty cool one, but probably too complicated to learn by heart—and hey, that's where `man` is for: you don't have to know all details!).

Chapter 3

A language, not a program

If you come from the SPSS or Stata world, you probably expect to *open SPSS* or *open STATA* and then work within it. There is *one* SPSS, with an icon you can click on.

In contrast, there are many ways to issue Python commands, which is actually a first important lesson: Python is not a program (like SPSS, or Word or Excel are programs), but a language — and there are a bunch of programs one can talk to in this language. Because of this, it is so flexible and can be used in so many contexts.

Therefore, there is no such thing as *opening Python*. Instead, there are several programs in which you can write code in Python and/or execute it. You probably will have your own favorite after a while, but you should be aware of the alternatives, as it is not only partly a matter of taste which one to choose. Especially if you later on want to run a script not on your own computer but, for instance, on a university server, you want to be able to deal with different ways of running Python code.

3.1   A note on different versions

In all examples in this book, we use Python 3. Most likely, it won't matter which exact version you have. The differences between, say, 3.4 and 3.5 are so minor that you are very unlikely to notice.

However, this is *not* true for versions starting with a 2. Python 2 is *not* just an old version of Python 3, and the latest version of Python 2, Python 2.7, is still widely used. This is important to know, because if you look on Google or StackOverflow for help, or if you read some of the books and tutorials I cite, then chances are high you come across code written in Python 2.

13

The differences are not very big. Most notably, in Python 2, you can write `print "Hello"`, while in Python 3, you need to write `print("Hello")`; and in Python 2, it is a bit more complicated to deal with Unicode (i.e., with special characters like umlauts, emoticons and the like).

> **!** So, if you see some code and think "hey, they forgot the brackets after print", then it's Python 2 code. Very often, you can just change such little things to make it work with Python 3. There is even a tool for the command line that does this automatically. It's called, not very surprisingly, `2to3`.

A last thing to know is that it is possible (and also common!) to have different versions of Python installed on one and the same computer. For example, it might very well be the case that someone works with both Python 2 and Python 3. For example, MacOS and most Linux distributions already *have* some version of Python installed by default, because they need it for internal workings. Make sure that you don't accidentally use the wrong one.

3.2   The interactive shell

This is the most simple and most straightforward way to just issue some Python commands. Go to the command line and just enter

```
1  python3
```

You just started a so-called python interpreter (or shell), and you can type Python commands, for example

```
1  print("Hello world!")
```

When you are done and want to go back to your normal Linux command line, just type

```
1  quit()
```

You also installed `ipython3`, which is some kind of luxury version of `python3`. It offers some more help and interactive features. It is, for example, better with graphics, and it is easier to copy-paste code.

For a quick look into a dataset or if you just need a calculator (yes, you can simply type `5+2` and stuff like that), starting an interactive shell can be very useful. But it comes with a big disadvantage: You cannot save your code.

In most cases, this is therefore, you probably want to do something else.

3.3　A text editor plus the command line

You could also write your code in an arbitrary text editor. For example, you could start

```
gedit &
```

(or any other text editor you like) and write the following code:

```
#!/usr/bin/env python3
print("Hello world!")
```

Save it as `/home/damian/hello.py` and go back to your linux shell. The first line is a so-called shebang, it tells your computer how to run the program.

Go to your home directory and tell the shell that the file you just wrote is a program and that you are allowed to run it: You tell it that the (u)ser should get the right to e(x)ecute it. Of course, you don't have to do that again if you want to run the program again next time.

```
cd /home/damian
chmod u+x hello.py
```

You can now run the program by typing

```
./hello.py
```

> ! The advantage of this approach is that on *every* system, there is some form of text editor. You do not even need a graphical interface for it. Imagine a really ressource-consuming program you wrote: the great advantage of being able to run it from the command line like this is that you do not have to open any other program next to it. In addition, if you can start your program with a single command like this, it means that you can use a huge variety of Linux tools to actually run it – for example once a day or every hour.

3.4　Integrated development environments (IDE)

For daily work, though, there are more helpful ways of developing Python code, so called integrated development environments, or IDEs. We will use Spyder, which is very similar to programs you are familiar with. It looks very much like RStudio and a bit like STATA, so you probably feel a bit at home here.

Just start it like this (or by clicking on it in the menu):

```
spyder3 &
```
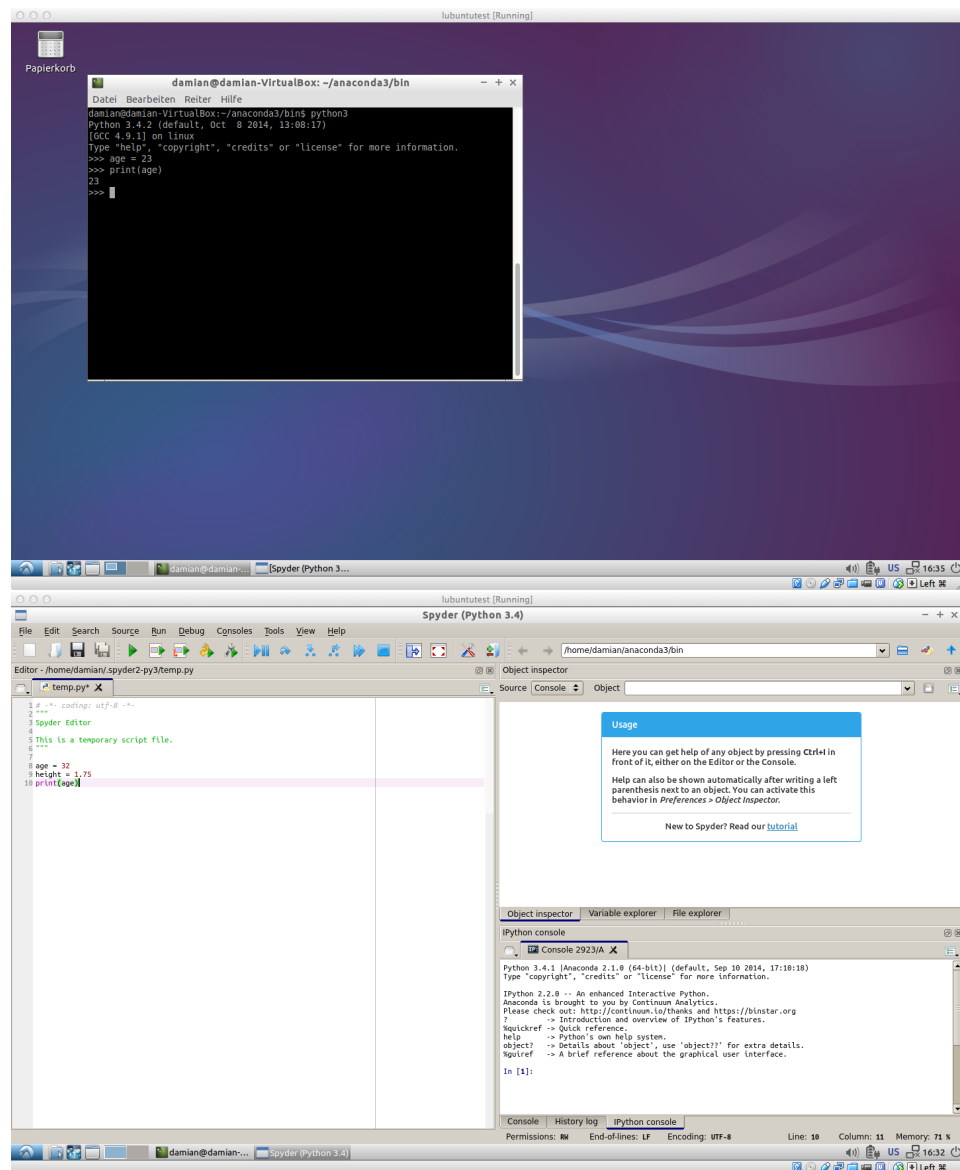
Figure 3.1: Different ways to issue Python commands: By starting the interpreter `python3` in the terminal (top) or by using an integrated development environment (IDE) like spyder (bottom).

It is a good environment to work in, and you will probably do a lot of your work in there. There are other IDEs, most notably one called PyCharm, which I use a lot. It is, however, much more geared towards professional programming and less towards data analysis, so you probably do not want to use it at the start of your Python journey.

3.5 Jupyter Notebook

Especially for data *analysis*, there is a fourth form of issuing Python commands: Jupyter Notebook, formerly known as iPython notebook. It uses the aforementioned iPython, but lets you run it in your web browser. The great advantage is that you can interactively play with the data, and save code *and* output as well as own annotations in one place. You can find some examples for such notebooks here: `https://github.com/damian0604/bdaca/`

You can start Jupyter Notebook from the command line:

```
1  jupyter-notebook
```

It should automatically open a web browser, and lets you do everything from there. For example, in Figure 3.2, you see part of a VAR time-series model conducted in Jupyter notebook.

While this is great for *interactively analyzing* data and keeping track of the output, you might already guess that using Jupyter Notebook is not such a smart way for running programs that take a while to run. For example, if you write a program that downloads a lot of data from websites, maybe running for half an hour or so (as we will do in Chapter 8), you don't want to do that in your browser. But for statistics stuff like we'll discuss in Chapter 11, it is really great. For more examples, have a look at the books by McKinney (2012) and Russel (2013).

3.6 Recap

You should have understand why there is no *one* way of "running Python" and know what the pros and cons of different ways of running Python code are.

For you, the most important ones are:

- Spyder as an allround solution for daily work

- Jupyter Notebook as a useful tool specialized in interactive exploration and analysis of data

And next to that, you should keep in mind that one can also run programs from the command line or starting an interactive Python interpreter on the command line. There will be a point in time when you want to do that.
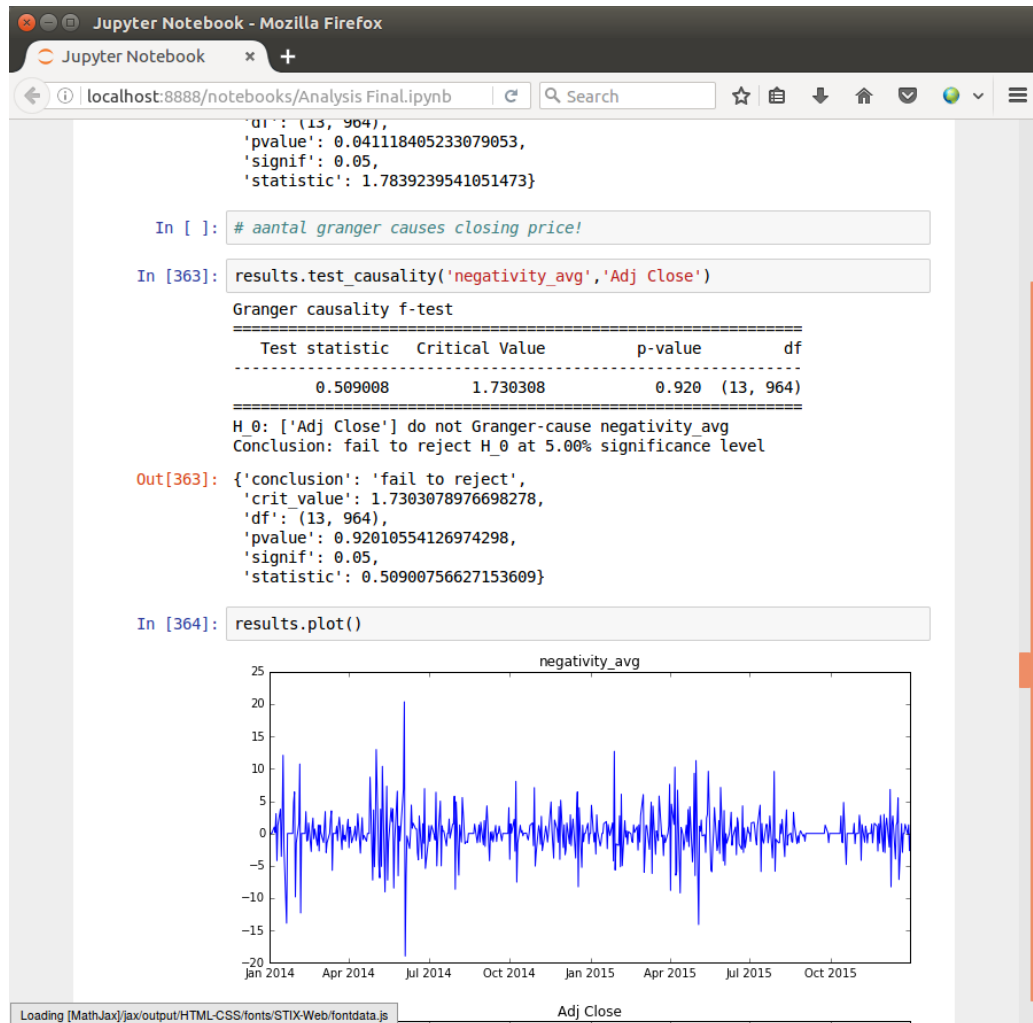
Figure 3.2: Jupyter Notebook lets you run Python code from within your web browser and lets you save it together with the output and your own comments.

Chapter 4

The very, very basics of programming in Python

Before we start writing our first own program, we need to clarify a few concepts. Luckily, they are pretty simple—but nevertheless, it is crucial to completely understand them.

> **!** Of course, what I present here is far from complete. I'll mention only the most essential parts that you need to understand to get started. Everything that is not strictly necessary right now or that is pretty intuitive (I guess I don't have to say that mathematical operators like $+$, $-$, $*$, or $/$ work exactly as you'd expect?) will be left out, and a lot is pretty much simplified. That's because our main objective is to get started as soon as possible with some real life data, not to get every detail right as you would do in a computer science course. After all, we are no computer scientists.

So, start an environment of your choice (see Chapter 3) and follow me along with some examples.

4.1 Datatypes

4.1.1 Basic types

When thinking of a *variable*, most social scientists immediately think along the lines of how the term 'variable' is understood in SPSS or STATA datasets (or in methods sections in social science papers, for that matter): Something like age that has one value for each case (each person, each newspaper article, each unit of analysis). That's *not* how we define a variable here. (R users might suffer less from this misunderstanding.)

Instead, we distinguish several types of variables (datatypes), and the basic ones take——in contrast to what you might expect from the SPSS/STATA world—*one and only one* value:

```
1  age = 33
2  height = 1.75
3  male = True
4  name = "Damian"
```

It is of crucial importance to understand that each of these four variables is of a different type. The first one, age, is an *integer*, or, shorter, an *int*: A whole number without anything behind the comma. If age is an int, then someone cannot be aged 32.3.

The second one, height, is a *floating point number*, or, shorter, a *float*. In contrast to an int, a float can have something behind the comma. Unlike in SPSS, you do not have to specify a number of decimal places, $height = 1.7527647326573265743648$ would be perfectly acceptable.

The third one, a *Boolean value*, or, shorter, a *bool*, can only take two values: True or False. Note that there are no " " around True and that True starts with a capital latter. This is how Python recognizes it is not a . . .

. . . *string*, the fourth data type. A string contains some arbitrary text, which is indicated by surrounding " " or ' '.

4.1.2 Type conversion

Python automatically converts some types for you (for example, luckily, you can divide a float by an int without thinking about their types). But still, you have to be very careful in using the right datatype. For example, you cannot write code like this:

```
1  numberofguests = "20"
2  bottlesofbeer = 100
3  bottlesperperson = bottlesofbeer / numberofguests
```

This does not work, because you cannot divide an int by a string. However, one can convert a string to an int (if the string contains a number):

```
1  bottlesperperson = bottlesofbeer / int(numberofguests)
2  print(bottlesperperson)
```

This code would work! By the way, if you want to have it the other way around and make a string from an int, the function is called `str()`.

> **!** You might wonder why one would have to convert types at all. Couldn't one just use the right type from the start? That's because a lot of data that we will analyze does not come in the right format. For example, if we have a file with some tweets followed by the number of retweets they

received, then data that we read from the file in principle is text, thus, a string. If we want to do some calculations with the number of retweets, we have to convert it to an int.

4.1.3 Lists and dicitonaries

Imagine we want to calculate the mean age of a group. It would be very inefficient to have a variable for every single person:

```
1  age1 = 22
2  age2 = 25
3  age3 = 23
4  age4 = 28
5  age5 = 26
6  meanage = (age1 + age2 + age3 + age4 + age5) /5
7  print(meanage)
```

In fact, we could almost do it more efficiently by hand. Thus, we might want to have something that is more like what is called a variable in SPSS or STATA: something named age that can contain multiple values. Such a datatype is called a *list*. You can have lists of strings, lists of floats, lists of ints, and even lists of lists (and lists of lists of lists...). A list is denoted by [ ], and the entries are seperated by commas. Let's create a list of ints:

```
1  age = [22, 25, 23, 28, 26]
```

To calculate the mean age, we could now divide the sum of the elements by the number of elements (or, technically speaking, by the *length* of the list). We'll discuss such functions in the next section, but for those who cannot wait, this is how it works:

```
1  age = [22, 25, 23, 28, 26]
2  meanage = sum(age)/len(age)
3  print(meanage)
```

> **?** What do you think, of what type are `age`, `sum(age)`, `len(age)`, and `meanage`? If you want to know if you got it right, you can get the answer with `type(meanage)`, `type(sum(age))` and so on.

We can also access a specific item from a list: If we want to know the age of the third person, then we can get it with

```
1  age[2]
```

Yes, 2, because we start counting with 0! The drawback of our current approach is that you have no idea who the person is that is 23 years old. You could have a second list in the same order:

```
1  name=["John","Bas","Anne","Sheila","Mark"]
```

Note that this is a list of strings, as we can see from the "-signs. Having multiple lists in a corresponding order is actually a strategy we will work with pretty often, and it is pretty much the same as having a dataset with multiple variables in SPSS or STATA. If we are interested in the name and age of the ith person, we could now find out:

```
1  i=3
2  print(name[i])
3  print(age[i])
```

Another option would be the last data type that we will discuss: a *dictionary*. As the name says, a dictionary is something where you can look something up——in Python terms, you search for a *key* and get its *value*. A dictionary is denoted by { }, with entries separated by commas, and keys and values by a colon:

```
1  nameage={"John": 22, "Bas": 25, "Anne": 23,"Sheila": 28 ,"Mark": 26}
```

Note that in this examples, the keys are strings and the values are ints, but they could also have any other data type. If we want to know how old Sheila is, we can retrieve that information:

```
1  print(nameage["Sheila"])
```

4.2 Functions

We have played around a bit with variables. To do some useful things with them, however, we need something called *functions.* In fact, we already used some: `print()` for example, or `len()` and `int()`. You can imagine a function as a command that in most cases takes some input (or, as we will call it, one or more *arguments*) and does something with it. The argument is supplied between (). For example, the function `print()` takes some variable as argument and show its content on the screen. This works for all types you know until now:

```
1  answer = 42
2  question = "What's the answer?"
3  print(question)
4  print(answer)
```

Note that you can use single quotes inside a string if you use double quotes outside the string (or vice versa!), but don't mix them up, otherwise it is unclear where the string ends. Of course, you do not have to define the variable in advance but also can do so on the fly:

```
1  print("Hello world")
```

Pay attention to the " ", which indicate that you do not supply a variable name that refers to a string, but the string itself. You could as well write:

```
hi="Hello world"
print(hi)
```

This gives exactly the same result. Another function that we already used is `len()`. It gives the length of an object, for example, the number of elements in a list or the number of characters in a string. You can actually also use functions within a function:

```
hi="Hello world"
print("The string",hi,"has",len(hi),"characters")
opdetap=["Estaminet","Steenbrugge Blond","Palm"]
print("They serve",len(opdetap),"different beers")
```

Make sure you understand where a " is placed and why!

4.2.1 Writing your own functions

There are a lot of useful functions available, either directly or from one of the numerous additional modules that can be loaded into Python. However, you can also write your own functions. This is something you might not need right now, but it can come in very handy later on. Imagine you would want to write a function to calculate the mean age, given a list of integers with ages. Then you could define a function to do this:

```
def averageage(agelist):
    y = sum(agelist)/len(agelist)
    return y
```

We basically define that our function takes some list as input (we called it `agelist`, but we could have chosen any other name). Then, it calculates some variable `y` (again, an arbitrary name) and returns that value.

So, if we have a list

```
ages = [22, 25, 23, 28, 26]
```

we can now simply write

```
averageage(ages)
```

to get the mean age, because we have defined the new function `averageage` before.

Obviously, for such a simple thing, this is not really necessary, because we could have just calculated

```
sum(ages)/len(ages)
```

directly. But if the calculation is more complicated and if we have to do it repeatedly, this can save us a lot of copy-pasting and makes our code more readable.

> **!** Once I was teaching this course, a student had the great idea of writing a function she named `translatetoenglish()`. It took a Dutch string as argument, passed that one to to Google Translate, and returned the English translation. So, once the function was defined, you could write something like `print(translatetoenglish("Wat een mooie dag!"))`, which would produce the following output: `What a beautiful day!`. She used it for more serious business, namely to loop over a dataset of multilingual tweets to produce a new dataset in English. What a loop is, is explained in Section 4.4.

4.3 Methods

We have learned that a function is called[1] by simply writing its name and passing some variables as an argument between `()`. Another way of doing something with variables is using a *method*[2]. In contrast to a function, a method is directly associated with a variable. For example, each string has a built-in method with the name `.lower()`. If you want to change all Capital Letters to lowercase, you can simply call the method by attaching it to the string. See these examples:

```
1  hi="Hello world OUT THERE"
2  print(hi.lower())
3  print("TESTtttt".lower())
```

As you see, methods are very similar to functions, but the way they are called is different. If `lower()` was a function, you would write `lower("TEST")` — but it is a built-in method of each string rather than a function, so you have to call it by writing `"TEST".lower()` instead. Obviously, `.lower()` does not need an additional argument to convert the string to lowercase, as it a method *of the very string it is supposed to work on*, so there is simply nothing between the `()`. Nevertheless, you cannot leave away the `()`, because every function and every method has to end with `()`, otherwise, it would not be executed.

For now, it is not important to know why some things are a method and

---

[1]To 'call a function' means to execute it.

[2]Strictly speaking, a method is also a function, but hey, this is no computer science course, so we'll allow ourselves to be a bit sloppy (and it relieves me from the burden to explain to you what a *class* is, what I would have to do to tell you the complete story).

some are a function.[3] It is important to know, however, that both ways of doing something with your variables exist and how you use them.

> ⚠ Note that functions and methods usually do not change the value of an argument itself but rather return a new object as result. So, if you have a string `mystring = 'HELLO'`, then calling `mystring.lower()` does *not* change mystring itself – it only *returns* the lowercased version of it. You therefore have to assign the result to a new variable (`mylowerstring = mystring.lower()`) or to the old one (`mystring = mystring.lower()`).

4.4   For loops

All these functions and methods are mainly interesting if you do not apply them one time, to one string, but hundreds, thousands, or millions of times. In fact, this is the very essence of what we are doing in this course: Splitting up a task in small functions or methods that we then apply repeatedly.

To tell the computer to repeat something, we use a construction called a loop, or more specifically, a *for-loop*. Let's start with an example (when typing it over, start line 3 by pressing SPACE four times):

```
1  alltweets = ["Great lecture at the UvA", "I HATE YOU!", "I want BEER"]
2  for tweet in alltweets:
3      print(tweet.lower())
```

> ⚠ After a line ending with a colon, Python expects an indented block. If you do not do this in a consequent way (for example, you sometimes press TAB, sometimes use 4 spaces, sometimes 3, or none at all, Python will complain and give you an "indention error". Some IDEs and some editors automatically convert TABs to spaces, but to avoid any confusion, the best convention is to simply never press TAB and always use spaces.

What happens? In line 1, we define the list `alltweets`. In line 2, we instruct the computer to take the first element of the list `alltweets` and give that element the name `tweet`. This is done by using the `for`-command. In human language, it might read as "Please repeat the task I will explain in the following indented lines `for` each element `in` the given list. In the task description given in the following block of indented lines, I will refer to the element that you are working on at that time as `tweet`".

Note that the name `tweet` is completely arbitrary chosen, while `for` and `in` are mandatory statements.

---

[3]In fact, sometimes this is kind of an arbitrary choice made by those who invented Python. Some operations have even been implemented as both function and method.

The indented line, line 3, is now executed. `tweet` now has the value of the first tweet of the first element of `alltweets`. After the indented block is finished, the program goes back to line 2, takes the next element from the list, assigns the name `tweet` to that next element, executes the indented block, and repeats until all elements have been used.

We could extend our code to produce output that looks a bit more fancy (and do not forget to indent lines 4 to 7 by using four spaces at the beginning of each line):

```
1  i=0
2  alltweets = ["Great lecture at the UvA", "I HATE YOU!", "I want BEER"]
3  for tweet in alltweets:
4      print("Printing tweet number",i,"in lowercase:")
5      print(tweet.lower())
6      print("\n")
7      i = i +1
```

`\n` denotes a line ending, so we get an empty line. In the last line, we add one to our counter variable `i`. A short form of doing this is `i+=1`, and it's pretty likely that I'll use that shorthand in the lectures.

> **?** Can you write your own loop based on the functions and data structures you learned until now?

4.5 If statements

Another important way of structuring your code is to define that it only has to be executed under specific conditions. Again, the block of code that has to be executed under a condition is indented and introduced by a colon.

```
1  age = 23
2  if age > 25:
3          print("Older than 25")
```

Of course, this does not print anything, as 23 is less than 25. You can also specify different conditions using the `elif` statement:

```
1  age = 23
2  if age >23:
3          print("Older than 23")
4  elif age <23:
5          print("Younger than 23")
6  elif age==23:
7          print("Exactly 23")
```

If you want to have a condition that is met if none of all others are met, you can use `else:`. Note the double == in line 6. This is very important: a

single = assigns a value to a variable, a double == compares two variables. Maybe it is obvious, but also note that you can nest such structures and have a condition within a for-loop (and maybe have another loop within the condition).

> ? Can you write a program that takes each element from a list (using a for-loop), and then prints it if it if satisfies a specific condition?

4.6 Recap

You should have no problems explaining what the following datatypes are:

- int

- float

- bool

- string

- list

- dictionary

You should know what functions and methods are and how to use some basic ones like `print()`, `len()` or `.lower()`.

You should have understood that you can write such functions yourself.

And, very importantly, you have to understand for-loops and if-conditions, including the role of indentation to structure your code.

Chapter 5

Basic ways to retrieve and store data

We already learned in section 4.1 (Datatypes) how we can internally create some data structures that we can use to further do some analyses. For example, if we wanted to do some analyses with names, ages, and height, we could create three lists like this:

```
1  age = [22, 25, 23, 28, 26]
2  name = ["John","Bas","Anne","Sheila","Mark"]
3  height = [1.83, 1.77, 1.55, 1.76, 1.74]
```

But of course, if we had such a limited amount of data, we wouldn't need to write a program to analyze it. So, we probably want to import the data from an external source. In this chapter, we will describe some ways to to this.

5.1 CSV files

In the easiest secenario, we already have a *table* in which each row contains a case and each column the value for age, name, and height, respectively. The universally acceptable format for such a table are CSV files: Text files in which columns are seperated by commas. Sometimes, a semicolon is used instead, conventions differ. Others use a TAB character instead, which is essentially the same, although one usually calls it a TAB-seperated file then.

You actually already saw a CSV file on page 9. Have a look back there if you forgot. Virtually all programs (including SPSS, Stata, Excel, . . .) can read and write CSV files, that's why we will use them very frequently. When creating data with such a program, make sure that you know how they are formatted exactly (e.g., are columns separated by `,` or `;`, are strings

encapsulated by "" or not, ... You can use the tools described on page 9 to inspect the files.

Lets create a text file with the following content:

```
1  John,22,1.83
2  Bas,25,1.77
3  Anne,23,1.55
4  Sheila,28,1.76
5  Mark,26,1.74
```

You can use gedit for this, an editor you find somewhere in the menu, but which you can—surprise, surprise!—also start from the command line with `gedit &`. Let's assume we saved this file as `/home/damian/mensen.csv`.

Now we can write a simple Python program to read this data:

```
1  import csv
2  name=[]
3  age=[]
4  height=[]
5  with open('/home/damian/mensen.csv', encoding='utf-8',mode='r',newline='')
       as csvfile:
6     reader = csv.reader(csvfile, delimiter=',')
7     for row in reader:
8         name.append(row[0])
9         age.append(row[1])
10        height.append(row[2])
11 print("Done!")
```

We see a lot of familiar structures: for example, the construction of (empty) lists in lines 2 to 4, the `for`-loop in line 7 to 10. So, let's first focus on the new parts:

- The `import` statement. It simply says that we want to load an external module, in this case one that knows how to read csv files. Import statements are usually written in the very first lines of a python scripts.

- The `with open(...)  as:` construction. It indicates that we want to open a file and assigns a name to that so-called *file object*, indicated by `as` (in this case, we decided to call it `csvfile`, which is a completely arbitrary name). The statement ends with a colon, so an indented block has to follow (remember?). When the indented block is over (thus, after line 10), the file is automatically closed again. That's exactly what we want, because by then, we have read all the information and stored it in lists; we don't need the file any more. The arguments passed to the `open()` function specify the file name, but also some other details of how to open it. We'll skip the details for now.

- The `csv.reader()` function. It returns an object[1] (which I decided to call, not very creatively, `reader`) that we can use to actually read the file. Note that in line 5, we only *opened* the file, we have not *read* a single byte from it yet. In line 5, we actually could have opened any arbitrary file—the `open()` command does not imply a specific data structure. Therefore now, in line 6, we define how to read from the file: Namely, by using the `csv` module. As an argument, we can give more specific information, for example, as done here, that the columns are separated by a `,`.

- The `.append()` method. It adds a new element to a list.

With your knowledge gained so far, you probably already can explain what in the for-loop in line 7–10 exactly happens. Think about it for a moment before reading further.

OK, the solution.

In line 7, we take the first row from the csv table, as provided by the `reader` object. We call it row. `row` simply is a list of all columns in that specific row. So, when we start, it is the list `["John",22,1.83]`. This means that we can get the value in the first column, `"John"`, by asking for the first element of the list, namely `row[0]`. That's what we do in line 8, where we put this value into the (until now empty) list of `name`s. The same is done with age and height, before we return to line 7 and get the next row from the `reader`.

Now, `row` has the value `"Bas",25,1.77`, and we again append `row[0]` to the list `name`, `row[1]` to the list `age`, and `row[2]` to the list `height`. We continue with this procedure until nothing is left to read from the file.

Let's also check if the data actually looks like what we expected:

```
1  print(name)
2  print(age)
3  print(height)
4  i=2
5  print(name[i],'is',age[i],'years old and',height[i],'meter tall')
```

Great!

Let's now save these columns in to a new file, `test.csv`, but in a different order. To this end, we use the `zip` command that basically glues several lists together. It goes like this:

---

[1]For those who are interested: More specifically, it returns a so-called generator. You can loop over a generator, but you can do so only once. In every iteration of the loop, it gives you one next element (one more row from the file in our case), and once there are no elements left, it stops. The great advantage of this, especially when dealing with large files, is that you don't have to load the whole file into memory at once.

```
1  output=zip(age,height,name)
2  with open("test.csv",mode="w",encoding="utf-8") as fo:
3      writer=csv.writer(fo)
4      writer.writerows(output)
```

That's all! Have a look at the file you just created and check if everything went right.

5.2   JSON files

OK, let's do this quickly, because what we will do in the next section is way cooler: We will retrieve JSON-data directly from an Google or Twitter. JSON is a data structure that is very similar to (if not identical to the one of) a Python dict, You already learned on page 22 what a dict is. Have a look back if you do not recall exactly.

Because a JSON file has the same data structure, we can open a JSON file and directly import it into a dict. This is really handy, as a lot of online data is available in that format.

Lets again (with an editor like gedit) create a file with the following content and call it `/home/damian/mensen.json`.

```
1  {"Sheila": 28, "Anne": 23, "John": 22, "Bas": 25, "Mark": 26}
```

To read this file into a Python dict, we only need to do the following:

```
1  import json
2  with open("/home/damian/mensen.json", mode="r", encoding="utf-8") as fi:
3          mydict = json.load(fi)
4  print(mydict)
5  print("Sheila is",mydict["Sheila"])
```

Note that both the dictionary name `mydict` and the name of the file object `fi` are completely arbitrary. I often use `fi` (file in) for files I read from and `fo` (file out) for files I want to write to, so that I don't confuse the two.

As you might expect, saving is as easy as opening. We only need to do call the function `json.dump()`, which takes two arguments: the dictionary that we want to save and a file object:

```
1  import json
2  ages = {"Sheila": 28, "Anne": 23, "John": 22, "Bas": 25, "Mark": 26}
3  with open("/home/damian/mensen.json", mode="w", encoding="utf-8") as fo:
4          json.dump(ages,fo)
```

Note that in the `open()` function, we now specify `mode="w"` instead of `"r"` — we open a file for writing, not for reading.

> ! Note that opening a file for writing (i.e., with `mode="w"`) creates a new file if it does not exist, **but if it already exists, it immediately deletes all content** that might have been in there.

But as I promised, it get's cooler, because in the next section, we will retrieve JSON objects not from a file, but directly via an API.

5.3   APIs

5.3.1   How does it work?——A first example
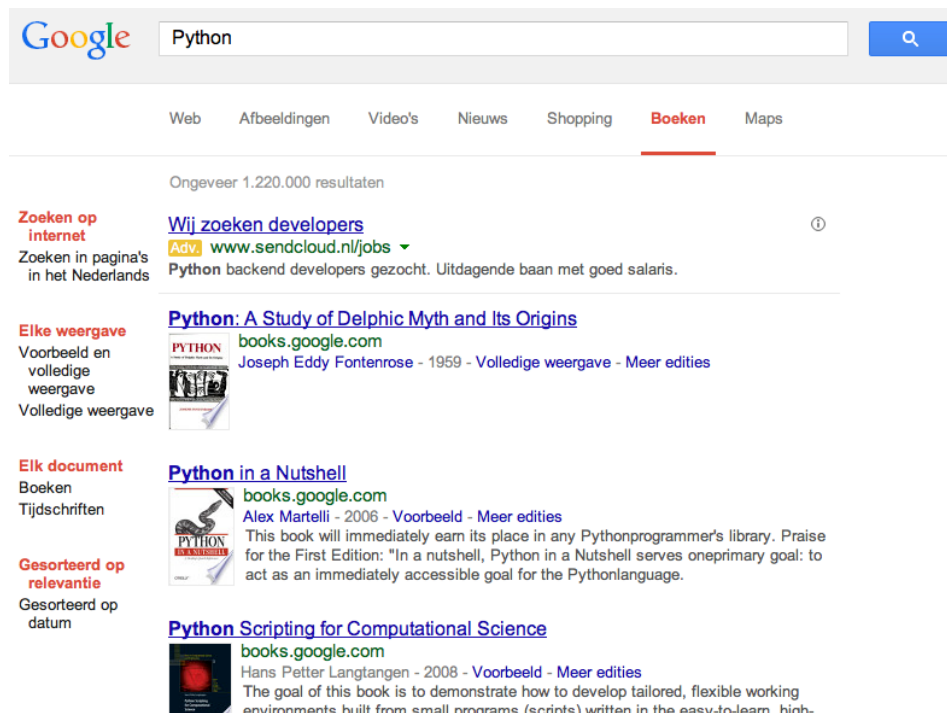
Figure 5.1: The GoogleBooks web interface

Let's assume we want to retrieve data from some online service, for example book descriptions from GoogleBooks. Of course, we could surf to their website, enter a search query, and somehow save the result. This would result in a lot of impracticalities, most notably that the website is perfectly readable and understandable for humans, but may have no meaning for a computer program. As humans, we have no problem understanding which parts of Figure 5.1 refer to the author of a book, what the numbers "2006" and "2008" mean, and on. But it is not trivial to think of a way to explain

to a computer program how to identify variables like `author`, `title`, or `year` in the output.

> We will learn how to do exactly that in Section 8 (Web scraping). Writing such a parser can be really useful, but it is also error-prone and some kind of a detour, as we are trying to bring some information that has been optimized for human reading *back* to a more structured data structure. Thus, if an API exists, it saves *a lot* of work.

Luckily, however, many online services do not only have web interfaces, but also offer another possibility to access the data they provide: an API (Application Programming Interface). In our example, we submit our request to GoogleBooks, and the API returns a JSON object with all the relevant data in it, in a neat and organized structure.

Consider the case of GoogleBooks. It offers an extremely simple API, as you do not have to log on (everyone can just use it), and because you call it in a very straightforward way: Assiming that you want to retrieve books that match the search string "python", you simply open the url `https://www.googleapis.com/books/v1/volumes?q=python` — and get a JSON object with all relevant information.

```
1  from urllib.request import urlopen
2  import json
3  from pprint import pprint
4  antwoord=urlopen("https://www.googleapis.com/books/v1/volumes?q=python").
       read()
5  data=json.loads(antwoord.decode("utf-8"))
6  pprint(data)
```

What does the code do? In lines 1 to 3, we import a module to download data from a URL, a module to read JSON data, and (to have some luxury) a module that prints json objects in a more readable way. Nothing spectacular so far.

In line 4, we access the GoogleBooks API and store the response in a variable we gave the arbitrary name `antwoord`. The function `urlopen` opens the URL, and its `.read()` method actually reads the content one gets when accessing the URL. Note that this is very much the same like opening a file and subsequently reading its content.

Until now, `antwoord` is just a bunch of bytes, and we have not made any attempt to somehow interpret it. That's what we do in line 5: `antwoord.decode("utf-8")` transforms the bunch of bytes into a string. However, a string is not really what we want, because we know that the string contains in fact JSON data[2]. And, as we saw in Section 5.2 (JSON files), a JSON string

---

[2]We know these things (that the bytes are a UTF-8 encoded string and that the string

can be directly translated to a Python dict. This is done by the function
`json.loads()`. Thus, `data` contains a Python dict with all information
provided by the GoogleBooks API based on our search query. In line 6, we
simply print that information.

> **!** Why `json.loads()` in line 5 and not `json.load()`, as we did in Section
> 5.2 (JSON files)? Because `json.load()` takes a file object (which we
> don't have here) as argument, and `json.loads()` a string.

Let's have a look at the output `pprint(data)` produced to see what is
actually stored in our dict `data`. As you can see, it is a lot, which is why I
removed some lines from the following listing to make the picture clearer.

```
1  {'items': [{'accessInfo': {'accessViewStatus': 'SAMPLE',
2
3  <<I REMOVED SOME LINES HERE>>
4
5    'volumeInfo': {'authors': ['Niklas Luhmann'],
6        'canonicalVolumeLink': 'http://books.google.nl/books/about/Love.
             html?hl=&id=Hh-kncGf4QkC',
7        'categories': ['Social Science'],
8        'contentVersion': '0.0.2.0.preview.3',
9        'description': 'This short text, originally '
10
11 <<I REMOVED SOME LINES HERE>>
12
13       'language': 'en',
14       'pageCount': 96,
15       'previewLink': 'http://books.google.nl/books?id=Hh-kncGf4QkC&
             printsec=frontcover&dq=inauthor:%22Niklas+Luhmann%22&hl=&cd=1&
             source=gbs_api',
16       'printType': 'BOOK',
17       'publishedDate': '2010-12-06',
18       'publisher': 'Polity',
19       'readingModes': {'image': True, 'text': True},
20       'subtitle': 'A Sketch',
21       'title': 'Love'}},
```

First of all, we see at the very beginning that `data` contains an entry
`items`, which is is probably what we need. We could access this part of the
dict by typing `data["items"]`.

Now let's see what `data["items"]` consists of. First of all, it seems to
be a list of different items (book records, in our case), as `[` denotes the
beginning of a list here (in contrast to `{`, which would imply a dict, see

---

contains JSON data) because it is defined like that somewhere in the documentation of
the GoogleBooks API. Nevertheless, both things are very common for many APIs, and
one could therefore just try it out as well.

Section 4.1 (Datatypes)). This implies that we could get the ith book by typing `data["items"][i]`.

What information is provided for each item? First of all, we find another dict called `'accessViewStatus'`. This doesn't look very interesting, so let's skip it. Some lines later, however, we see that our first book (and presumably also all the following books) have a dictionary called `'volumeInfo'`. That looks promising!

Now we're there! The dict `data['items'][i]['volumeInfo']` consists of several key-value pairs, so that we could get the language in which the 3rd book was written by typing: `data['items'][2]['volumeInfo']['language']` (remember, we start counting at 0).

> ! If you want to see all keys of a given dictionary `mydict` without having to wade through all the output, then you can get them by calling the method `mydict.keys()`.

Let's put all this into practice and write a small program that gives us some information on books written by a specific author. What about some information on books written by Niklas Luhmann, one of the most important social theorists of the 20th century[3]? We just saw how to access the data we retrieve from the API, but we still have to see how we can transmit a more complicated search string (namely, one specifying that we are interested in one author only).

Let's do some reverse engineering. By playing around with the Google-Books web interface and paying attention to the URLs that appear in your browser's address bar, you can actually infer how the part after `?q=` is constructed for more complicated search strings. I basically surfed to `http://books.google.com`, entered Luhmann as search string, got some results, clicked on the link of one of the results that were actually indeed written by Luhmann, which then gave me the page showed in Figure 5.2. In fact, I saw that entering `inauthor:"Niklas Luhmann"` would have been the correct search string for what I intended, and I also saw that this indeed becomes part of the URL.

Putting all this information together makes it possible to write the following code to retrieve information on some books written by Luhmann:

```
1 luhmann=urlopen('https://www.googleapis.com/books/v1/volumes?q=inauthor:"
     Niklas+Luhmann"').read()
2 niklas=json.loads(luhmann.decode("utf-8"))
```

---

[3]Accoring to Wikipedia. Actually, we could also access Wikipedia's API as well to directly retrieve this information. But, to be honest, I just used their web interface. But if I were to collect biographical information on more than 20 sociologists, it would probably already worth learning how to use Wikipedia's API.
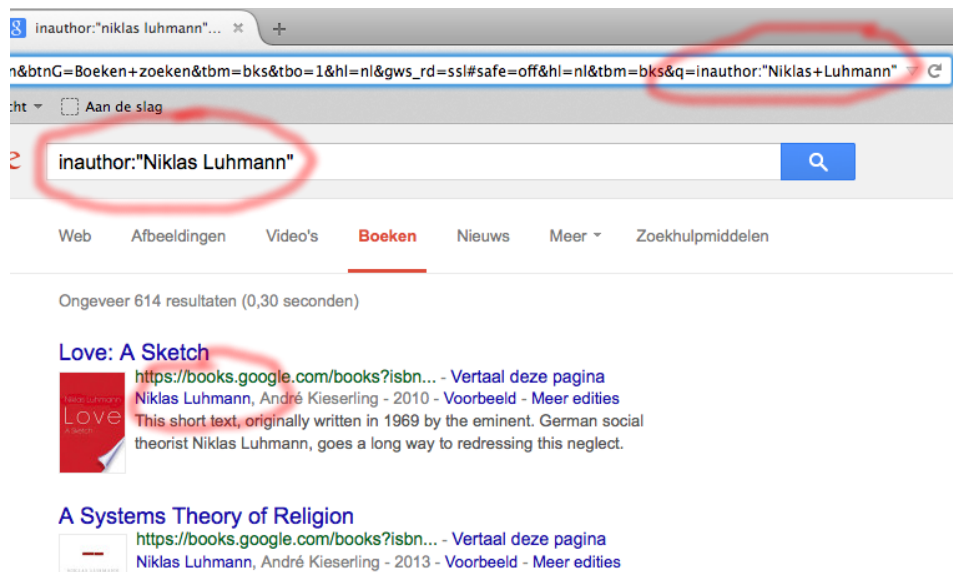
Figure 5.2: Reverse-engineering the GoogleBooks API

```
3  for boek in niklas["items"]:
4      print (boek["volumeInfo"]["authors"],": ",boek["volumeInfo"]["title"])
```

And why not calculate how long they actually are?

```
1  totalpages=0
2  numberofbooks=0
3  for boek in niklas["items"]:
4      pages=int(boek["volumeInfo"]["pageCount"])
5      print(pages)
6      totalpages=totalpages+pages
7      numberofbooks=numberofbooks+1
8  print ("The average length of a book by Luhmann is", totalpages/
       numberofbooks, "pages")
```

? An alternative way of solving the same problem would be the following program. Can you see advantages and disadvantages of both approaches? Which approach would you choose?

```
1  pagelist=[]
2  for boek in niklas["items"]:
3      pagelist.append(int(boek["volumeInfo"]["pageCount"]))
4  print ("The average length of a book by Luhmann is", sum(
       pagelist)/len(pagelist), "pages")
```

5.3.2   The Twitter API

For this exercise, we need to import a module named `twitter`.[4] You can easily install it from the command line with

```
1  sudo pip3 install twitter
```

Do this before you read any further.

There is a lot of information on the Web, an extensive documentation on `https://dev.twitter.com/`, and we will have a number of exercises on the Twitter API in class, so there is no need to go too much into detail here. There are two main differences with the GoogleBooks-example: (1) You do not have to access the API directly via `urlopen()`, but some nice person already wrote a so-called wrapper which does that for you; and (2) you need an account and log on.

To create an account, go to `https://apps.twitter.com/app/new`, register as a developer, create a new app and note down your consumer key, consumer secret, access key and access secret (the latter two are also referred to as access token). See Figure 5.3 for screenshots. Most things should be rather self-explanatory. You can leave the field for a "callback URL" empty, and for "website", just make something up like `idontknowyet.com`.

In the following example, replace ACCESS_KEY etc. with your data (don't forget to put them between `"  "`). If you want to retrieve my last 20 tweets as a JSON object (who wouldn't want to do that?), you can simpy do so with a three-line program:

```
1  from twitter import *
2  twitter = Twitter(auth = OAuth(ACCESS_KEY, ACCESS_SECRET, CONSUMER_KEY,
       CONSUMER_SECRET))
3  posts=twitter.statuses.user_timeline(screen_name="damian0604", count=20)
```

If you do not care about my tweets, but rather about more general information about me (like my bio statement or the number of followers I have), you can do so as well:

```
1  tweep="damian0604"
2  info=twitter.users.show(screen_name=tweep)
3  print(tweep,"has",info["followers_count"],"followers.")
4  print("He is following",info["friends_count"], "people.")
5  print("This means that his follower-to-following-ratio is",int(info["
       followers_count"])/int(info["friends_count"]))
6  print("\nThis is what he says about himself:\n")
7  print(info["description"])
```

---

[4]Another popular module for accessing the Twitter API is called `tweepy`. If you want to dig deeper into using the Twitter API, you might want to check that one out as well and find out which one fits your purposes best.

To see what type of elements `info` contains, you can either read the documentation on `https://dev.twitter.com/` or use `print` or `pprint` to inspect its structure.

5.4 Saving the retrieved data

Once you have retrieved the data from an API, there are several ways to go. You could directly analyze them in some way, but maybe you just want to save them first (in fact, it might be a good idea to sve them anyway for archiving purposes). Combining our knowledge we gained up til know, we can conceive of multiple ways of storing the data. The first possibility, as we have learned, would be to simply dump the Twitter dataset we aquired into a JSON file.

```
1  with open("/home/damian/someone.json", mode="w", encoding="utf-8") as fo:
2      json.dump(info,fo)
```

We could also think of writing a CSV table, in which we store only the relevant information. This is especially useful if we want to further anayze the data using a different program later on. To this end, we could first store all relevant information in lists and then save these lists to a CSV file.

```
1  import csv
2  tweets=[]
3  dates=[]
4  for p in posts:
5      tweets.append(p["text"])
6      dates.append(p["created_at"])
7  output=zip(dates,tweets)
8  with open("tweetswithdate.csv",mode="w",encoding="utf-8") as fo:
9      writer=csv.writer(fo)
10     writer.writerows(output)
```

5.5 Another example: The AmCAT API

Some of you might use AmCAT (Van Atteveldt, 2008), a framework for content analysis. If you have no idea what AmCAT is, just skip this section. If you do, however, you will be delighted to hear that you can access your AmCAT projects via an API and retrieve all your data for further analysis with Python.

First, you have to install the AmCAT Client:

```
1  sudo pip install amcatclient
```

You need your username, your password, the project number, and the article set number. Then, you can simply adapt the following sample script to your own needs.

```
1  import json
2  import csv
3  from amcatclient import AmcatAPI
4
5  api = AmcatAPI('https://amcat.nl', 'USERNAME','PASSWORD')
6  articles = api.list_articles(project='XXXX', articleset='XXXX')
7
8  # articles is a generator, meaning you can read it only once
9  # that's impractical for us, so we turn it into a list:
10 articles = [art for art in articles]
11
12 #get an idea about the data by printing info about the first item
13 print(articles[0].keys())
14 print(articles[0])
15
16 #%% save as json
17 with open("articles.json",mode="w",encoding="utf-8") as fo:
18         json.dump(articles,fo)
19
20 #%% additionally, make a csv table with some of the data fields
21 with open("articles.csv",mode="w",encoding="utf-8") as fo:
22     writer=csv.writer(fo)
23     for art in articles:
24         # while saving, make sure that the text does not contain
25         # linebreaks ("\n" or "\r")
26         writer.writerow([art["medium"],art["date"],art["text"].replace("\n
               "," ").replace("\r"," ")])
```

5.6   Recap

You should have understood

- how a CSV file looks like and how it can be read;

- how a JSON file looks like and how it can be read;

- how JSON data can be retrieved via an API;

- how to write the retrieved data to a JSON file;

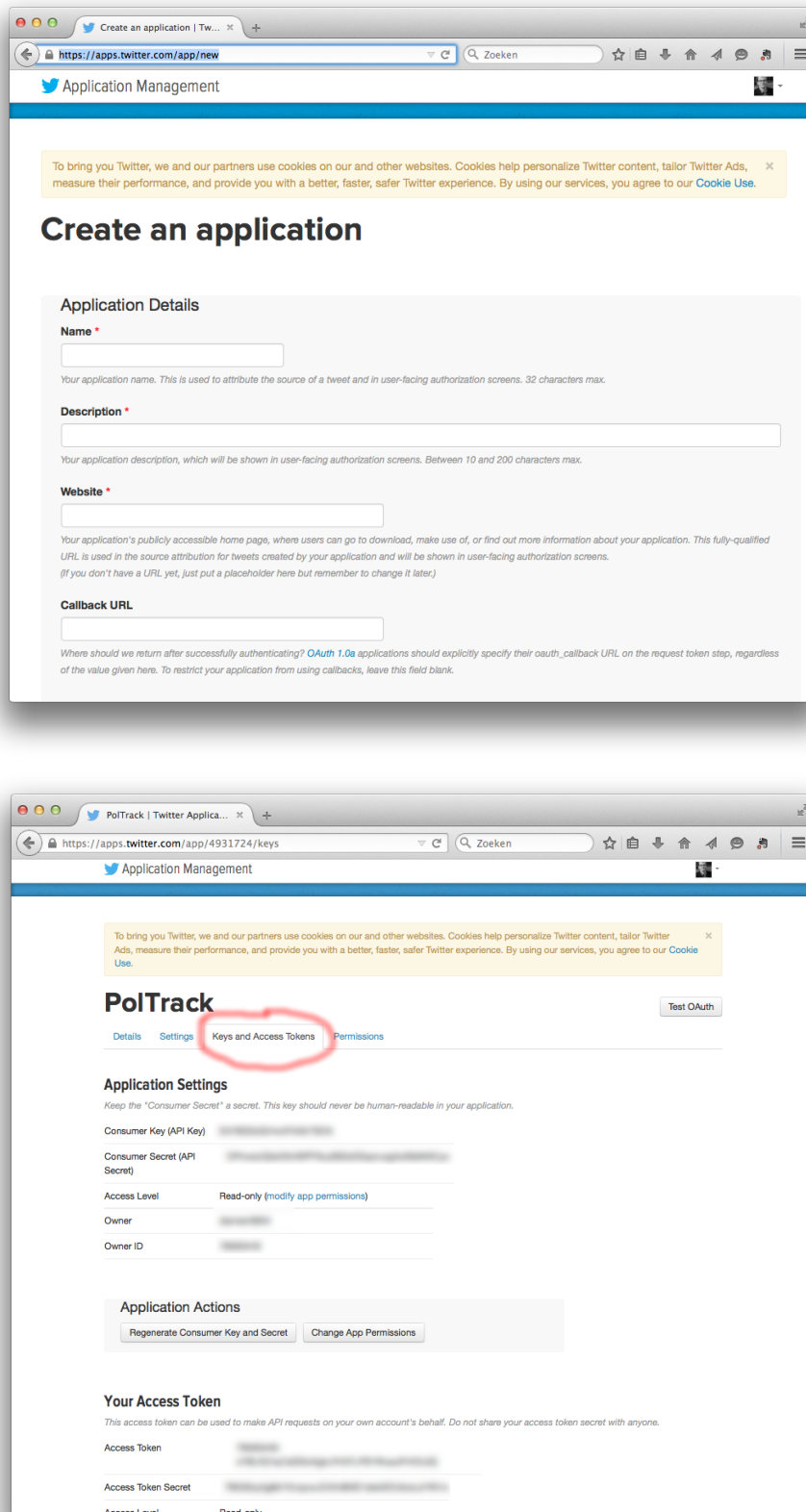- how to write the retrieved data to a CSV file.

Figure 5.3: Creating a Twitter app and requesting tokens

Chapter 6

Sentiment analysis

6.1 Preparation

In this chapter, we are going to conduct a sentiment analysis. You can use your own data for this, you only have to modify the proposed code accordingly. For instance, you can use the examples from earlier chapters about how to use CSV files, JSON files, or APIs and add the sentiment analysis code from the examples below to them.

We will discuss several approaches to sentiment analysis below and use several example datasets. In the end, you should be able to transfer what you learned to own datasets and make an informed decision on the approach to use.

You can also download the whole code for the first example if you want to:

```
1 wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/ex
    -senti-simple.py
```

6.2 A simple dictionary-based approach

The simplest way to do a sentiment analysis is to simply count the number of positive and negative words in a text. While this has obvious drawbacks (for example, because we do not care about word order, we cannot distinguish between 'good' and 'not good'), it is a good starting point to understand how sentiment analysis works. Moreover, because the approach is so simple, you can easily modify it to measure other things.

First of all, we need a list of negative words and a list of positive words. Let's use the command line to make a new directory and download two files

with such lists from the internet[1]:

```
1  cd /home/damian
2  mkdir sentiment
3  cd sentiment
4  wget http://www.unc.edu/~ncaren/haphazard/negative.txt
5  wget http://www.unc.edu/~ncaren/haphazard/positive.txt
```

Let us use some Twitter data to play with. Unfortunately, Twitter does not allow to share datasets with tweets. It does allow, however, to share lists with tweet ids. Via the Twitter API, anyone can then download the whole tweets and reconstruct the dataset.

I prepared a list of tweet ids with tweets about Bernie Sanders in the pre-election period of the US elections before a candidate was nominated. I also prepared a script that automatically downloads the associated tweets.

Download them. . .

```
1  wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/
      sanders.txt
2  wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/
      retrievetweets.py
```

. . . and then open `retrievetweets.py`, where you have to change lines 5 to 8 by entering your API credentials (see Subsection 5.3.2). Now you can run the file, and it will retrieve a set of approximately 900 tweets and save them to a TAB-separated file.

First of all, you should carefully inspect the file to get an understanding of the data structure. For example, you could use the `head` command on the bash command line (see Chapter 2.2):

```
1  head -5 sanders.tsv
```

Which columns are interesting? Is there a header line (with column names), or do all lines contain data?

Once we have this information, we can make a plan on how to proceed.

1. We can skip the header row with the `next()` function (you didn't know that yet, but now you do).

2. We need a `list` of all tweets. As we know we can get this by starting with an empty list, followed by looping over all rows in the file and appending the value of the appropriate column to that list.

---

[1]The example presented here is based on the following tutorial: `http://nealcaren.web.unc.edu/an-introduction-to-text-analysis-with-python-part-1/`, and the word lists are originally developed by Wilson, Wiebe, and Hoffmann (2005)

3. We are only interested in the words, not in punctuation. So, we could make a second list where we replace them by a space. This can be done using the `.replace()` method.

```
1  import csv
2  tweetlist=[]
3  processedlist=[]
4  with open("/home/damian/sentiment/sanders.tsv") as fi:
5      reader=csv.reader(fi,delimiter='\'})
6      next(reader)
7      for row in reader:
8          tweet=row[0]
9          tweet_processed=tweet.lower().replace("!"," ").replace("."," ").
               replace("?"," ").replace(u"\u201A"," ").replace("'"," ").
               replace('"'," ").replace('#'," ").replace(':'," ")
10         tweetlist.append(tweet)
11         processedlist.append(tweet_processed)
```

Ok, we have the data and already did the necessary preprocessing (removing unnecessary characters in this case). BTW: You see why we access the element with the index 0 of the list `row` in line 7?

You might actually want to have a look at how the data look like to see that you did not make any mistakes. You can do so with some `print()` functions, or by clicking on the values in the variable explorer (Figure 6.1).
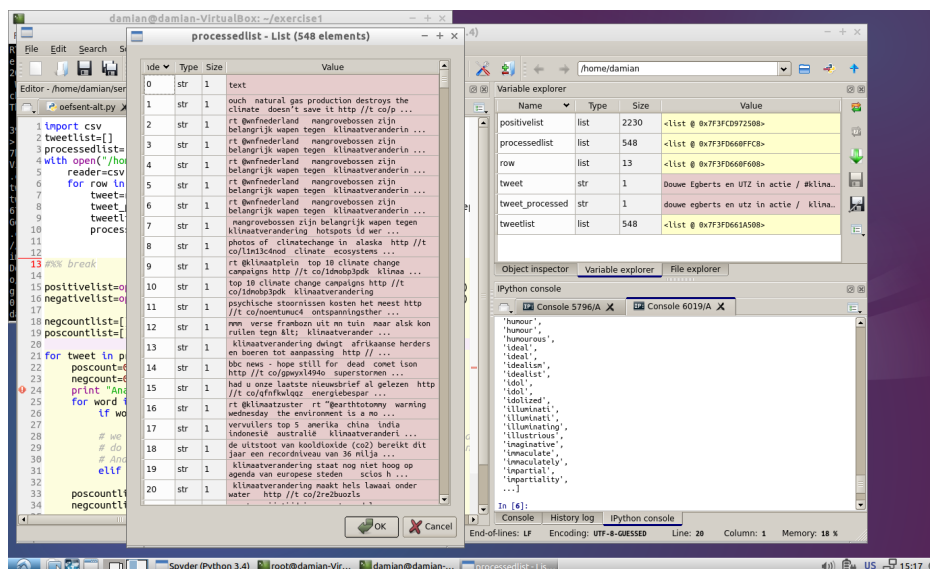

Figure 6.1: Inspecting the data with Spyder's variable explorer

We can now plan our next steps:

4. We need to read a list of positive words

5. We need to read a list of negative words

We already downloaded two of such lists, `positive.txt` and `negative.txt`. Use the bash commands `head` or `cat` to find out how they look like!

Luckily, Python makes it very easy to read a text file in which each line should be treated as a single string into a list of strings:

```
1  positivelist=open("/home/damian/sentiment/positive.txt").read().splitlines
       ()
2  negativelist=open("/home/damian/sentiment/negative.txt").read().splitlines
       ()
```

Print the lists and check if they look like you would expect them to look like!

Now we can start with the real analysis. The algorithm is straightforward:

6. Loop over `processedlist`.

7. Within the loop: Set a variable `poscount` that indicates the number of positive words in this specific tweet is 0.

8. Within the loop: Set a variable `negcount` that indicates the number of negative words in this specific tweet is 0.

9. Within the loop: Split each element from that list (thus, each processed tweet) into words and loop over these words

10. Within this inner loop: check wether the word is an element of `positivelist`, and if so, increase `poscount` by one

11. Same for negative words

12. Add counts to some lists

13. Optionally: weigh by the length of the tweet

Translated to Python code, it looks like this:

```
1  negcountlist=[]
2  poscountlist=[]
3  sentilist=[]
4
5  for tweet in processedlist:
6      poscount=0
7      negcount=0
8      print ("Analyzing this one:",tweet)
```

```
9      for word in tweet.split():
10          if word in positivelist:
11              poscount+=1
12          elif word in negativelist:
13              negcount+=1
14      print("It contains",poscount,"positive words and",negcount,"negative
           words.")
15      poscountlist.append(poscount)
16      negcountlist.append(negcount)
17      sentilist.append((poscount-negcount)/len(tweet))
18
19  print("Average sentiment:",sum(sentilist)/len(sentilist))
```

The only thing we still want to do is to save a dataset for further analysis — maybe with Python, maybe with some statistics package:

```
1  output=zip(tweetlist,processedlist,negcountlist,poscountlist,sentilist)
2  with open("/home/damian/sentiment/sanders-analyzed.csv",mode="w",encoding
       ="utf-8", newline="") as fo:
3      writer=csv.writer(fo)
4      writer.writerows(output)
```

6.3 Vader

There are several nice things to the simple approach demonstrated above: First, it is insightful from a didactical point of view, as it illustrates the basic principles of bag-of-words analyses. Second, it is very flexible in that it can be modified to measure a lot of other things, not only positivity or negativity. Third, it is transparent—everyone can understand the basic algorithm.

However, in a real-world setting, it might be too simplistic. Instead of writing a sentiment analysis algorithm yourself, you can therefore invoke some algorithm that others have developed. For instance, the sentiment analysis module Vader by Hutto and Gilbert (2014) deals with negation, punctuation, intensifiers ("very") and dampeners ("kind of") as well as emoticons (see `https://github.com/cjhutto/vaderSentiment`).

Vader is integrated in NLTK, a Python module that we will use for a number of text processing tasks in this course (see Section 7.2). You already installed NLTK when setting up your VM, but we need to additionally install the Vader sentiment lexicon. We only have to do this once, and we can do it within Python:

```
1  import nltk
2  nltk.download('vader_sentiment')
```

It is very easy to use:

```
1  from nltk.sentiment import vader
2  senti=vader.SentimentIntensityAnalyzer()
3  senti.polarity_scores('This is a great day!')
4  senti.polarity_scores("I don't like this food")
5  senti.polarity_scores("I love her, but I hate him")
```

The `.polarity_scores` method returns a dictionary with four values. In our example, we get the following results:

```
1  {'neu': 0.406, 'pos': 0.594, 'neg': 0.0, 'compound': 0.6588}
2  {'neu': 0.587, 'pos': 0.0, 'neg': 0.413, 'compound': -0.2755}
3  {'neu': 0.282, 'pos': 0.244, 'neg': 0.474, 'compound': -0.5346}
```

We see that we get for each of the sentences an estimate of its positivity, negativity, neutrality, and a combined measure of those. For detailed information on the workings, have a look at Hutto and Gilbert (2014)!

We see that we could easily implement this function into the script we developed in the previous section. For example, we could do:

```
1  compoundscores=[]
2  for tweet in tweetlist:
3      compoundscores.append(senti.polarity_scores(tweet)['compound'])
```

> ? Can you write a sentiment analysis script that applies Vader to your own data?

6.4   Sentistrength

Another very frequently used algorithm is SentiStrength (Thelwall, Buckley, & Paltoglou, 2012). Unfortunately, is is – in contrast to all software that we use in this course – not open-source. SentiStrength is free for academic use, but you have to pay for it if you want to use it commercially. The Java version, which we would need to use it, can be requested from the authors (see `http://sentistrength.wlv.ac.uk/`), but – in contrast to a Windows version – it cannot be downloaded from their site.

For the sake of completeness, I will describe anyhow how to use Sentistrength together with Python – assuming that you are in the possession of a copy of the Java version of Sentistrength.

First of all, you have to install Java:

```
1  sudo apt-get install default-jre
```

Then, download the following module that I wrote to access Sentistrenght from within Python:

```
1 cd /home/damian
2 mkdir goodies
3 cd goodies
4 wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/
      sentipy.py
```

Then, copy your Java version of sentistrength to a folder called `/home/damian/goodies/sentistrength`.

When finished, you have to tell Spyder where to find the new modules. You can do this via the menu Tools/PYTHONPATH Manager (see Figure 6.2 for the two necessary entries). After that, you have to select "Update module names list" from the same menu, and close the Python consoles that are still open (or restart Spyder).
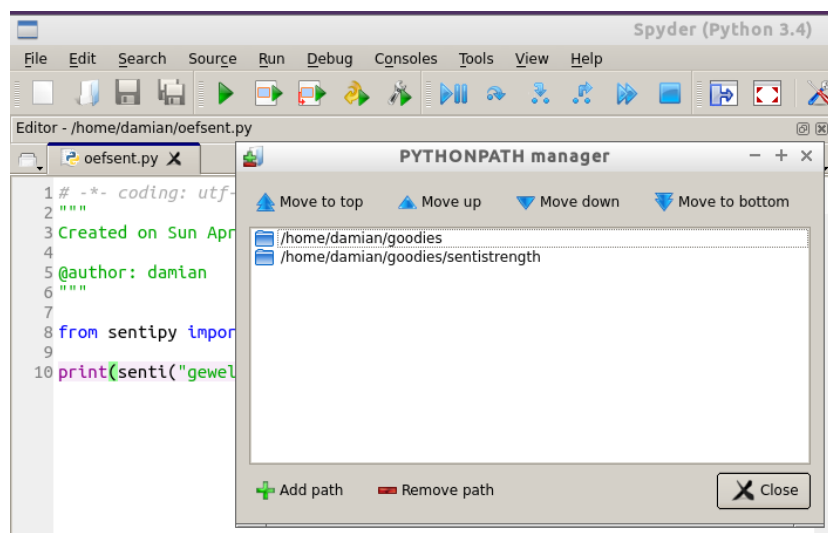

Figure 6.2: Telling spyder where to find additional modules

Now you can test wether everything works:

```
1 from sentipy import senti
2 print(senti("geweldig nieuws","dutch"))
```

You see that it returns a list with two values, the first one being the positivity (on a scale from 1 to 5), the second one being the negativity (on a scale from -1 to -5).

So, let's run it on a real dataset. Why not open a JSON file with some paragraphs from State-of-the-Union speeches (which I exported from AmCAT (Van Atteveldt, 2008)) and write the text, the positivity score, and the negativity score to a csv file?

You can get the speeches here:

```
1  wget https://raw.githubusercontent.com/damian0604/bdaca/master/examples/
        stateoftheunion.json
```

And this is the Python code:

```python
1  from sentipy import senti
2  import csv
3  import json
4  with open("stateoftheunion.json",mode="r",encoding="utf-8") as fi:
5          articles=json.load(fi)
6  with open("articles.csv",mode="w",encoding="utf-8") as fo:
7      writer=csv.writer(fo)
8      for art in articles:
9          s=senti(art["text"],"english")
10         print(s)
11         writer.writerow([art["medium"],art["date"],art["text"],s[0],s[1]])
```

You will surely see that there is a problem, though: It takes almost a second to call `senti()`. This is because of the not-so-good implementation in this case (we call an external java program each time we want to get the sentiment). We wouldn't have to care about time, but as we have > 20 000 paragraphs to analyze, we can either wait six hours or interrupt with CTRL-C . You might have noticed that this time, we did not first create lists that we combined with `zip`, but that we write each line immediately to the file. This has two advantages: It saves memory, and if we abort, everything we did until that moment has already been written to the output file.

> **?** Can you extend the algorithm in Chapter 6.2, so that additional columns based on the Sentistrength-algorithm are saved?
> Can you also implement some way of comparing the outcomes of both approaches?

The reason why the above approach is so slow, however, is not the sentiment analysis *itself*, it is the the fact that the external Java program has to be *started* 20 000 times. It would be more efficient to do this only once (MUUUUUCH more efficient, in fact). The approach below takes *less than a minute* to run. It saves the 20 000 texts to analyze in *one* text file, one per row (thus, all line breaks within each text are removed before) and then tells sentistrength to calculate the sentiment per line. I wrote a function `sentifile` to achieve this. The `sentifile` takes two or three arguments: the name of an input file, the language, and optionally a boolean value that indicates wether you want the full text to be returned or not. If False (or ommitted), the function returns a list of (positive, negative)-tuples, if True, it returns (postive, negative, text).

The following sample code illustrates its working¿

```
1  from sentipy import sentifile
2  import csv
3  import json
4
5  with open("stateoftheunion.json",mode="r",encoding="utf-8") as fi:
6      articles=json.load(fi)
7
8  with open("textonly.txt",mode="w",encoding="utf-8") as fo:
9    for art in articles:
10       fo.write(art["text"].replace("\n"," ").replace("\r"," ").replace("\
           t", " ") + "\n")
11
12 # without text:
13 with open("scoresonly.csv",mode="w",encoding="utf-8") as fo:
14    writer=csv.writer(fo)
15    sents=sentifile("textonly.txt","english")
16    for s in sents:
17       writer.writerow([s[0],s[1]])
18
19 # with text:
20 with open("scoreswithtext.csv",mode="w",encoding="utf-8") as fo:
21    writer=csv.writer(fo)
22    sents=sentifile("textonly.txt","english",True)
23    for s in sents:
24       writer.writerow([s[2],s[0],s[1]])
```

6.5 Alternatives

Let us take a step back and compare the approaches we have seen so far. The first one, the simple dictionary approach, was easy to understand and easy to modify, but not very powerful. The second, Vader, was much more powerful. One disadvantage, however, is that it is only available in English. The third one, Sentistrength, is comparable to Vader, but has as big advantage the support of a number of languages, including Dutch. However, this comes at a price: It is not available for free, making it more difficult to share code with others; and it is written in Java, which requires us to use some tricks to actually use it in Python.

But there are more alternatives. For instance, pattern.nl (De Smedt & Daelemans, 2012) is very easy to use, open source, and it supports multiple languages. However, it is only available for Python 2 and not compatible with Python 3.

A last, but increasingly popular method for doing sentiment analysis is using supervised machine learning. How to do that will be discussed in Chapter 10.

6.6   Recap

You should have understood the general working of different approaches to sentiment analysis. You should know about their pros and cons. Practically speaking, you should be able to

- implement a simple, dictionary-based sentiment analysis tool yourself

- integrate a tool like Vader or SentiStrength in your own analyses.

Chapter 7

Automated content analysis

Automated content analysis covers a wide range of techiques, and not all of them can be covered within this tutorial (for an overview and more background information, see Boumans and Trilling (2016)). We will start with a first, powerful yet easy deductive approach, in which we basically count how often a pre-defined pattern (a search string, for example) occurs. This technique is known as using *regular expressions*. We will then learn how to take characteristics of human language into account to get closer to the real meaning. The technique we use for this is called *Natural Language Processing (NLP)*. The other approaches to automated content analysis mentioned by Boumans and Trilling (2016) are discussed in Chapters 9 and 10.

7.1 Regular expressions

A regular expression is a *very* widespread way to describe patterns in strings. You probably know wildcards like * or operators like `OR`, `AND` or `NOT` that you can use in search strings in a lot of applications. But of course this is rather limited: Maybe you want to say somethhing like "I want all words starting with a letter or a number (but no special character), then a -sign, then again some letters, and after the last dot two or three letters (but not more!) letters. You probably saw that this would be some way of describing an email-adress.

A universal language to formulate such patterns is *regular expressions*. There are some dialects (but the differences are minimal), and you can use them in many editors (!), in the Terminal, in STATA . . . and in Python.

In its most simple form, a regular expression is just an ordinary search string:

```
1  python
```

finds ("matches") every occurance of the word "python". But we can also indicate that two different characters are allowed:

```
1  [pP]ython
```

matches both "python" and "Python". Some other useful things:

- . matches *any* character

- [0-9] matches a digit

- [a-z] matches all lowercase letters

- [a-zA-Z] matches all upper- and lowercase letters

- ([tT]witter|[fF]acebook) matches both Twitter and Facebook and is OK with misspelling them as twitter or facebook.

You probably got the idea. You can also specify how often sth has to occur:

The item *before* has to occur

- ? 0 or 1 times

- * 0 or more times

- + 1 or more times

For example,

```
1  personali[zs]ed-?communication
```

matches

```
1  personalizedcommunication
2  personalized-communication
3  personalisedcommunication
4  personalised-communication
```

There are actually much more possibilities, so download a regular expression cheat sheet (google or try this one: `http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/`) to build your own one. Also the wikipedia page on regular expressions is pretty good. You can also play around at `http://www.pyregex.com/`.

So, let's try to use regular expressions in Python. There is a whole module for this (its called `re`) that provides a lot of interesting functions for finding and replacing stuff based on regular expressions.

- `re.findall("[Tt]witter|[Ff]acebook",testo)` returns a list with all occurances of Twitter or Facebook in the string called `testo`

- `re.findall("[0-9]+[a-zA-Z]+",testo)` returns a list with all words that start with one or more numbers followed by one or more letters in the string called `testo`

- `re.sub("[Tt]witter|[Ff]acebook","a social medium",testo)` returns a string in which all all occurances of Twitter or Facebook are replaced by "a social medium"

- `re.match(" +([0-9]+) of ([0-9]+) points",line)` returns `None` unless it *exactly* matches the string `line`. If it does, you can access the part between () with the `.group()` method.

The difference between `findall` and `match` is thus that `findall` does not care about what it finds before and after the pattern, while `match` requires the whole line to be matched.

An example to illustrate the use:

```
line="           2 of 25 points"
result=re.match(" +([0-9]+) of ([0-9]+) points",line)
if result:
   print ("Your points:",result.group(1))
   print ("Maximum points:",result.group(2))
```

would produce the following output:

```
Your points: 2
Maximum points: 25
```

This is cool, isn't it? We now know how to extract information from a semi-structured string so that we can store it in variables for further analysis!

On page 62, you can find some additional information about some regular expression features, especially about the difference between lazy matching and greedy matching, which basically specifies where to "stop" when a regular expression contains something like `.*` (thus, matching an arbitrary number of arbitrary characters, which is in fact much more useful than it sounds).

? Can you write a program (a so-called parser) that takes a semi-structured input file of your choice (maybe something you downloaded somewhere) as input and uses regular expressions to store the data in a structured way in a CSV table?

7.2   Natural language processing

To prepare, we have to install some more ressources. Run the following Python lines:

```
1  import nltk
2  nltk.download()
```

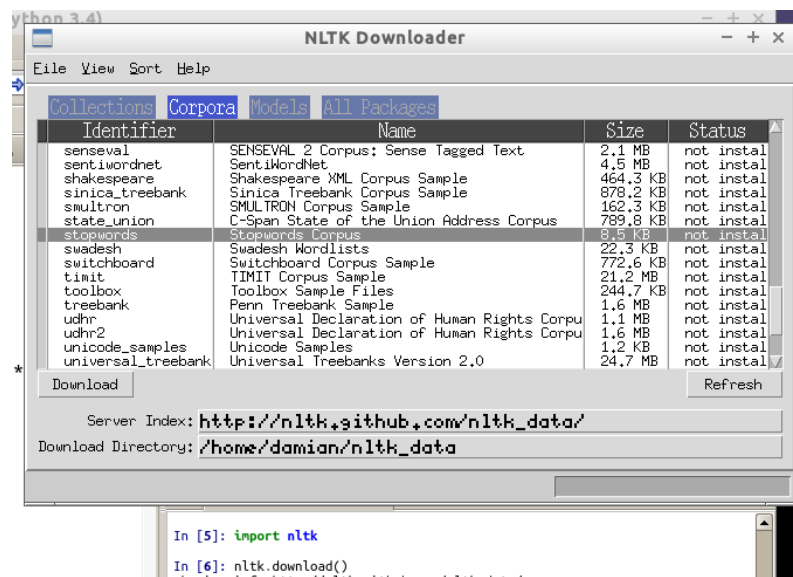A graphical interface is opened (Figure 7.1).

Figure 7.1: Loading additional NLTK-packages

Do *not* download all packages, that will take too much space and you do not need it. Instead, download only the packages "stopwords" , "punkt", and "maxent_treebank_pos_tagger" from the "all packages" tab. Close the window when done.

7.2.1   Stopword removal

Natural language processing is a term that is used to describe a set of techniques used to analyze language written by humans. This comprises simple tasks like the removal of stopwords in order to identify "relevant" words, but also much more complicated things like parsing of a sentence – which means to automatically identify, for instance, subject, verb and object in a sentence. The book by Bird, Loper, and Klein (2009) gives a comprehensive introduction into NLP with Python. The author also provide the Python package `nltk`, which we use here.

To find out all the possibilities, have a look at `nltk.org`, but here are some examples.

Let us first consider stopword removal. These are words without a meaning that is specific to a text, such as "a", "the", ... They are usually more disturbing than useful, because they hinder us from seeing the really interesting words. Keeping them in can lead to misleading results if we want to identify key terms (e.g., by means of a word count), if we want to calculate document similarities, or if we want to make a word co-occurance graphs. A very straightforward way to do so would be the following algorithm, which you – with the knowledge you gained so far – actually could have written yourself:

```
1  t = 'let us clean this up for her'
2  tn = ""
3  stopwords=['and','the','this','a','or','he','she','him','her','us']
4  for w in t.split():
5      if w not in stopwords:
6          tn=tn+w+" "
```

Instead of defining the list in the script itself, we could also read them from a file. Imagine we have a file called `stopwords.txt` in which we put one stopword per line (we could easily create such a file in `gedit` or Leafpad or any other text editor):

```
1  he
2  she
3  it
4  we
5  they
```

We could read this file to a list with

```
1  stopwords=[w.strip() for w in open("stopwords.txt",encoding="utf-8").
     readlines()]
```

Especially if we have a very long list, this makes our code much easier to read and to maintain. We could also use one of the pre-defined lists provided by `nltk`, but often, we might want to tweak it to our research interests anyway.

```
1  from nltk.corpus import stopwords
2  stopwords=stopwords.words('dutch')
```

7.2.2 Stemming

Another interesting step is *stemming*. When interpreting text, we usually do not want to distinguish between smoke, smoked, smoking. Stemming reduces

all of these forms to their stem `smoke`. Therefore, it is a typical preprocessing step (like stopword removal).

So, we could split a given string into words, then stem each word, and combine them again.

```
1  from nltk.stem.snowball import SnowballStemmer
2  stemmer=SnowballStemmer("english")
3  s="I am running while generously greeting my neighbors"
4  stems=""
5  for w in s.split():
6      stems=stems + stemmer.stem(w) + " "
```

We can now `print(stems)` to see how the stemmed string looks like.

> ! The last three lines can be replaced by a single line:
> `stems=" ".join([stemmer.stem(w) for w in s.split()])`
> The technique used for it is called *list comprehension* and a really cool feature, google it if you want to know more. In addition, the `.join()` method allows to join elements from a list to a single string. These two things are considered very pythonic and good coding style, so if you do this kind of stuff more often, it is useful to understand the working (but you can as well use the more elaborate way used in the example). In fact, also the stopword removal could be done in this way: Assuming we have a list with stopwords called `stopwords` and a sentence `s` from which we want to remove all stopwords, we can do so with
> `s2=" ".join([w for w in s.split() if w not in stopwords])`
> In doing so, we can create a new string `s2` with one single line of Python code in which we first split s into a list of words, then make a new list out of these words in which only those words are included that are not in the stopword list, and finally join this list into a single string seperated by spaces.

7.2.3   Part-of-speech tagging

By parsing sentences, we can find out what grammatical function the words within each sentence has. To this end, NLTK first splits sentences into words (it is called *tokenizing*). But, in addition to what we have done before with the `.split()` method, it attaches a label to each token that identifies the grammatical characteristics of each token.

```
1  import nltk
2  sentence = "At eight o'clock on Thursday morning, Arthur didn't feel very
       good."
3  tokens = nltk.word_tokenize(sentence)
4  print (tokens)
5  tagged = nltk.pos_tag(tokens)
6  print(tagged)
```

We see that we get a list of tuples (pairs of two values):

```
1  [('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'), ('
       Thursday', 'NNP'), ('morning', 'NN')]
```

This is pretty useful, because we can address the items with *slicing*. For example, to get the fifth tuple, we can write `tagged[5]`. Of course, we can also slice the tuple. So, to get only the grammatical function of the word "morning", we can use `tagged[5][1]`. Look up the meaning of the abbreviations like 'NN' up in the documentation. Chapter 5 in Russel (2013) also provides a lot of examples.

7.3 Recap

You should have understood

- How to remove stopwords

- How to stem a sentence

- How to parse a sentence and employ POS tagging

- How to use NLTK

Chapter 8

Web scraping

8.1 Overview

When scraping data from the web, we can distinguish two different stages:

1. Downloading a (possibly large) number of webpages

2. Parsing the content of the webpages

Let us start with the first point. Sometimes, we have a list with all URLs of all Dutch news items of the last year, for example, because we retrieved them from a RSS feed. In this case, downloading the data is trivial. We could for exampe use a for-loop in Python, or use the command `wget` from the bash command line. If we have a file `urls.txt` that contains one URL per row, we can download all urls with the command

```
1  wget -i urls.txt
```

We could use a number of additional options, in particular waiting between the URLs to avoid gettig blocked, or pretend that we use a specific web browser. `wget --help` or Google give you more information.

If we do not have such a list, but rather want to start from one page (e.g., the homepage of a specific website) and simply follow all links that we find on that page, we have to resort to a technique called *crawling*. If we for example want to download the page `http://gsc.uva.nl/`, including all links, plus all links encountered on the pages linked to (but stop after this second level of recursion), we can simply type

```
1  wget -l2 http://gsc.uva.nl/
```

The internet is full of examples (like here: `http://www.thegeekstuff.com/2009/09/the-ultimate-wget-download-guide-with-15-awesome-examples/`)

At the end of this chapter, I will point you to a framework that allows crawling with Python.

But first, let us turn to the second point: Parsing the content of the pages we downloaded. In virtually all cases, we do not care about the webpage as we downloaded it, but we rather want to extract *some* information in a meaningful way. For example, on a page with product reviews, we want to get rid of all advertisements, navigation elements, ... — in fact, everything except the reviews themselves, which we probably want to store in a list or a similar data structure.

In the next section, we will do so with an approach you are already familiar with: regular expressions. This is useful to show the general idea, and you could use a similar approach a wide range of input data, not only web pages (think of the LexisNexis example). However, especially with complicated HTML pages, this approach can become cumbersome. We will therefore learn how to use XPath descriptions to parse a webpage in another section.

8.2   A simple regular-expression based approach

First of all, have a look again at the materials about regular expressions. In particular, have another loot at how the `re.findall()` function can be used to get a list of all strings that match a regular expression within another string.

Then, open Firefox and go to `http://www.geenstijl.nl/mt/archieven/` `2014/05/das_toch_niet_normaal.html`. Even if you do not understand a word (which actually might help you to look at the structure rather than the content!), you will see that the article is followed by a whole bunch of comments. Already think about a way how one could identify the comments

Click on Tools/Web Developer/Page source (or press CTRL-U) to inspect the source code (Figure 8.1).

You should see that the whole block of comments starts with the tag `<div class="commentlist">` (and, much later, ends with `</div>`). Also, you should see that each comment starts with a tag `<article id="xxxxxxxx">` and ens with `</article>`.

This looks like a regular pattern, therefore, we can match it with a regular expression. Assuming that the whole page is called `tekst`, we can get the whole comment section with:

```
1   commentsection=re.findall(r'<div class="commentlist">.*?</div>',tekst)
```

Pay attention that we write `.*?` instead of `.*` only. The `.` matches every character, and `*` means an arbitrary number of times, therefore `.*` *also*
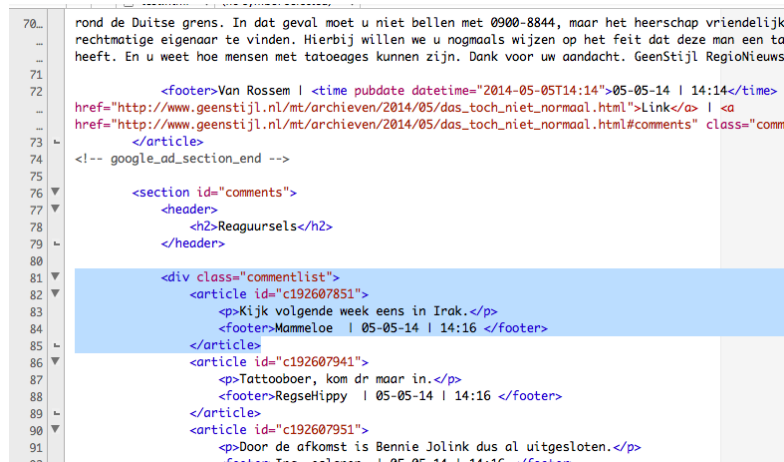
Figure 8.1: Examining the source code of a HTML page

matches `</div>`. This means that our regular expression would only stop at the *last* occurrence of `</div>` if there are several ones instead of the *first* one. This is referred to as *greedy matching*. Using `.*?` instead of `.*` means that we want to perform *lazy matching* instead, where we do stop as soon as the next part of the regular expression is matched.

As `re.findall()` returns a list of all matches, `commentssection` is a list of all commentssections. As there is only one comment section (which we can verify by calculating `len(commentssection)`, we can get the whole chunk of HTML-code of the comment section by accessing the first and only element of the list, `commentssection[0]`.

Now, we can do the same for each comment ("article") within the comment section:

```
1  comments=re.findall(r'<article.*?>(.*?)</article>',commentsection[0])
```

Voilà! We got a list of all comments!

But wait, why did we write the `()` around`(.*?)`? This simply means that only the part between `()` is returned. In other words, `comments` contains only what is *between* `<article.*?>` and `</article>`.

If we now but all this together, add some code to actually download the data (and to also parse the metadata, like author and time of writing), we get the following script:

```
1  from urllib import request
2  import re
3  import csv
4  
5  onlycommentslist=[]
6  metalist=[]
```

```
 7
 8   req = request.Request('http://www.geenstijl.nl/mt/archieven/2014/05/
         das_toch_niet_normaal.html', headers={'User-Agent' : "Mozilla/5.0"})
 9   tekst=request.urlopen(req).read()
10   tekst=tekst.decode(encoding="utf-8",errors="ignore").replace("\n"," ").
         replace("\t"," ")
11
12   commentsection=re.findall(r'<div class="commentlist">.*?</div>',tekst)
13   print (commentsection)
14   comments=re.findall(r'<article.*?>(.*?)</article>',commentsection[0])
15   print (comments)
16   print ("There are",len(comments),"comments")
17   for co in comments:
18       metalist.append(re.findall(r'<footer>(.*?)</footer>',co))
19       onlycommentslist.append(re.findall(r'<p>(.*?)</p>',co))
20   writer=csv.writer(open("geenstijlcomments.csv",mode="w",encoding="utf-8"))
21   output=zip(onlycommentslist,metalist)
22   writer.writerows(output)
```

Of course, rather than just saving it to a csv file, we could also add one or two lines to the for-loop and immediately perform a sentiment analysis!

> **!** What happens exactly in line 8 to 10? In line 8, we form a HTTP-request to specify what we want to download. We could skip this and just specify the URL, but we want to perform a little trick to fool the other side. As they do not like people scraping their content, we have to conceal that we actually do not use a web browser like normal people. Therefore, we let our program just pretend to be Firefox ("Mozilla"). In line 9, we download the web page. However, what we get is just a number of bytes, and Python does not know which byte corresponds to which character. This is why we use the `.decode()` method in line 10, which changes the sequence of bytes to a string as we know it. Furthermore, we remove all newlines and tabs from the string, because we want one long, uninterrupted string for further processing (especially, we do not want to care about this when designing our regular expressions).

Now, you should be ready to write your own parser for some content you are interested in!

8.3   XPath-parsing with lxml

One does not always have to reinvent the weel, though. It is useful to understand how parsing would work with regular expressions, and it is very likely to come into a situation where this knowledge can be applied. Especially in the case of HTML pages, though, several parsers exist that can make life easier.

In the example in the last section, we saw that an element in a HTML page can be nested in another element. In other words, a HTML page can be represented as a tree: For example, the page branches into an article, a navigation section, and a comment sections; the latter than branches into several comments. Each element in this tree can be identified by a so-called XPath. You can compare it with the path of a given file (`/home/damian/week6/slides.pdf`), where the leftmost element is the "highest" level, and where after each `/`, it indicates to which lower level one has to turn before finally arriving at the file (`slides.pdf`).

The good news is that you can look up the XPath of an element with your web browser (the bad news, though, is that on some pages it can be a bit tricky to get it *exactly* right). In Firefox, you have to install a plugin for that. It is called XPath Checker and you can get it by googling it or from `https://addons.mozilla.org/nl/firefox/addon/xpath-checker/`. Install it and restart Firefox.

Now, go to a website of your choice, select the part of content that you want to scrape, right-click on it, and select "View XPath". XPath Checker opens (Figure 8.2) and suggests an XPath. It also displays the content that would be selected if one used this XPath.
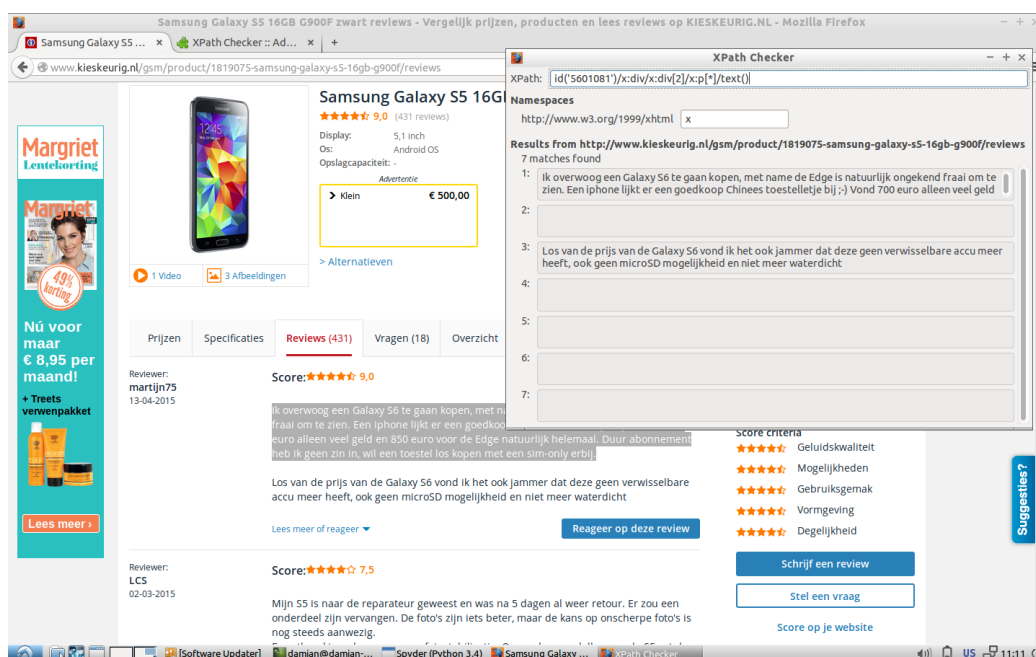

Figure 8.2: Identifying a correct XPath with XPath Checker

Often, you do not only want that *one* comment you selected, but *all*

comments. Nevertheless, select just one, and then try to modify the XPath until it fits your needs. The XPath Checker immediately shows you the results, so you can engage in a lot of trial- and error. Some rules for modifying the XPath:

- `//` (thus, a double slash rather than a single slash) means 'arbitrary depth' (=may be nested in many higher levels, there may be an arbitrary number of higher levels which are not further specified)

- `*` means 'anything' (if `p[2]` is the second paragraph, `p[*]` are all paragraphs)

- Let the XPath end with `/text()` to get all text

> ! When there is a line or paragraph break within the results of an XPATH, the `/text()` function might not function properly, as it sees each part as a separate element. This can be fixed by leaving away the `/text()` in the XPATH itself and instead using the `.text_content()` method later on, as this example illustrates:

```
1  reviews = tree.xpath('//div/div/div[2]/div[*]/div[2]/p[1]')
2  i=0
3  for review in reviews:
4        print("Review",i,":",review.text_content())
5        i+=1
```

In addition, you can also have a look at alternative XPath specifications. Again, select the text you are interested in, right-click, and choose "Inspect element".[1] You can now see how the HTML tags referring to that element are nested, which should correspond to the construction of your XPath (Figure 8.3). In particular,

- If you want to refer to a specific attribute of a HTML tag, you can use `@`. For example, every `*[@id="reviews-container"]` in an XPath would grab what is within a tag like `<div id="reviews-container" class="user-content">`

An example that scrapes all reviews of a specific telephone using one single XPath expression:

---

[1]If you use the Chrome browser, you can directly copy an XPath from the "Inspect element" window, but you have no way of interactively checking the XPath like with the Firefox XPath Checker.

```
1  from lxml import html
2  from urllib import request
3
4  req=request.Request("http://www.kieskeurig.nl/smartphone/product/2334455-
       apple-iphone-6/reviews")
5  tree = html.fromstring(request.urlopen(req).read().decode(encoding="utf
       -8",errors="ignore"))
6
7  reviews = tree.xpath('//*[@class="reviews-single__text"]/text()')
8
9  # remove empty reviews and remove leadint/trailing whitespace
10 reviews = [r.strip() for r in reviews if r.strip()!=""]
11
12 print (len(reviews),"reviews scraped. Showing the first 60 characters of
       each:")
13 i=0
14 for review in reviews:
15     print("Review",i,":",review[:60])
16     i+=1
```

This produces the following output:

```
1  67 reviews scraped. Showing the first 60 characters of each:
2  Review 0 : De Apple Iphone 6 Zilver/16GB het is een heel fijn apparaat
3  Review 1 : Apple maakt mooie toestellen met hard- en software uitsteken
4  Review 2 : Vanaf de iPhone 4 ben ik erg te spreken over de intuitieve i
5  Review 3 : Helaas ontbreekt het Apple toestellen aan een noodzakelijk i
```

Now it's your turn!

8.4 Alternatives

You can get very far with the approaches sketched in this chapter: regular expressions, the lxml-package, and wget for automated downloading tasks. BeautifulSoup is a python package that lies somewhere in between, as it resembles lxml, but internally uses regular expressions rather than an XPath to parse the HTML. Within lxml, you can also use so-called CSS selectors instead of XPATHs.

If you want to write an application that integrates crawling and parsing, you might have a look at the scrapy framework (`http://scrapy.org/`), although it might be a bit of an overkill.

One approach that we haven't covered is how to scrape content from sites that do complicated stuff like interactively loading data while they are visited. For example, think of a news site that uses a JavaScript to display comments: these comments may not be present in the HTML code of the site, but might be dynamically loaded while the user is reading the article. In

such a case, one can make use of a framework like Selenium (google "selenium python" or ask your instructor for some examples). Selenium allows you to start a browser, click somewhere, and so on (in essence, it just automates what a human would do), and then parse what is displayed in the browser.

8.5   Recap

You should be able to write your own web scraper. This entails that you

- have a global understanding of how a HTML page looks like, so that you can use this information to build your scraper

- understand that there are several ways to parse a specific piece of information from a page

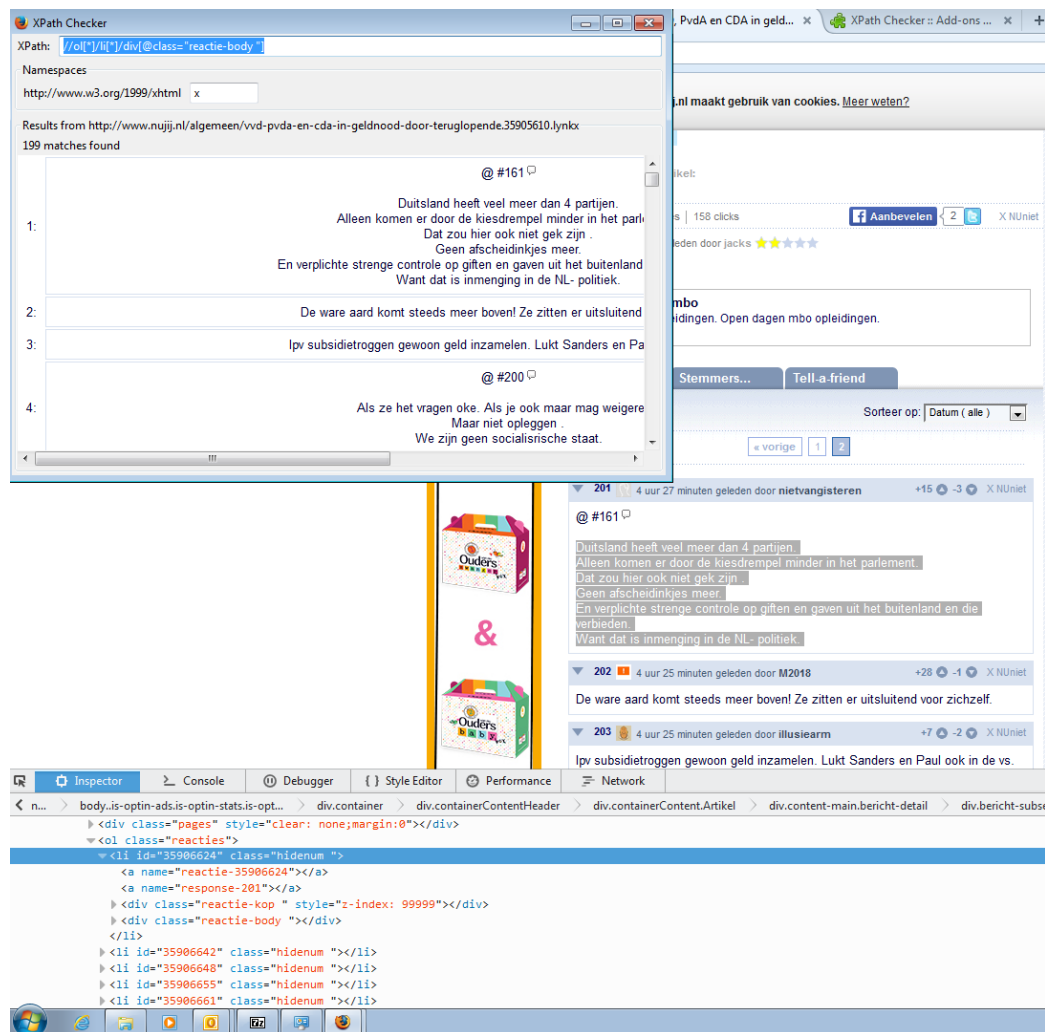- know not only regular expressions, but also how to use packages like lxml.

Figure 8.3: Gathering information for constructing an alternative XPATH specification, using the "inspect element" function.

Chapter 9

Network visualization

We are often used to representation of data in forms of two-dimensional tables (an Excel or SPSS file, a CSV file). We have already seen that this is not always the most appropriate way to conceptualize data: Hierarchical data, for example, can often be much better represented in JSON format. Another way of representing data is in form of a *network* (Figure 9.1).
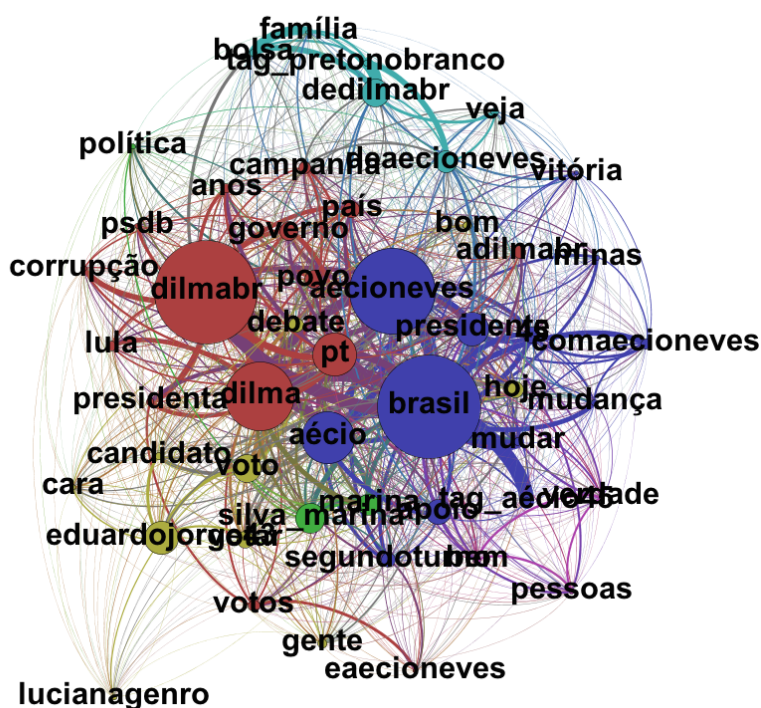

Figure 9.1: A network of co-occuring words

We call the elements in a network *nodes* and the connections between them *edges*. A node can be anything: a word, a person, whatever. Edges van be directed (like an arrow) or undirected. For example, a nework of Twitter users (the edges) could have edges that represent -mentions. These edges would be directed: if A mentions B, that does not imply that B mentions A as well. On a Facebook friendship network, this would be different: Here, we would have undirected edges.

Giving an introduction to network analysis falls outside of the scope of this tutorial, but you will learn how to save your data in a format that can be used to analyze or visualize them.

In particular, we will look how we can construct a network of words that co-occur together in texts, which can be a nice way to conduct some inductive framing analysis (see Boumans & Trilling, 2016; Trilling, 2015).

9.1   Producing a file for analysis with Gephi

Let us considering the following case: We have a list of strings (e.g., a list of tweets) and want to know which words often occur together in the same string (tweet). We could model our data in such a way that each word is a node. We could also attach a weight to each node, so that the node can be displayed bigger the more often the word is mentioned. We could now say that if two words co-occur together, they are connected with a node. Again, we could say that the more often this happens, the thicker the edge should become.

Thus, we basically need

- a list with all words (the nodes) with their frequency (the weight/size of the nodes)

- a list with all connections (the edges, thus, a list of pairs of nodes that are co-occuring in the same string/tweet) with the number of strings/tweets in which they co-occur.

The GDF file format is a format that allows us to store exactly this information. Have a look at the following example. In lines 1, it is defined how the following node definitions are formated: a variable called `name`, which is a string (that's what they mean with VARCHAR), followed by a variable `width` which has the format double (a type of number, we would call it a float). And indeed, you see that the following lines contain strings (the words which are the nodes) followed by the size of the node (the frequency of the words). In line 12, it is defined that the following lines contain the edge definitions: two strings (node1 and node 2, followed by the edge weight.

For example, we see that the words "coffee" occurs three times in total (line 2), and one time together with the word "beer" (line 13).

```
nodedef>name VARCHAR, width DOUBLE
coffee,3
beer,2
i,4
and,1
with,1
friend,1
having,1
like,3
am,1
my,1
edgedef>node1 VARCHAR,node2 VARCHAR, weight DOUBLE
coffee,beer,1
i,beer,2
and,beer,1
with,friend,1
coffee,with,1
i,and,1
having,friend,1
like,beer,2
am,friend,1
i,am,1
i,coffee,3
i,with,1
am,having,1
i,having,1
coffee,and,1
like,coffee,2
am,coffee,1
with,my,1
i,friend,1
like,and,1
am,with,1
having,with,1
i,my,1
having,coffee,1
i,like,3
coffee,friend,1
having,my,1
am,my,1
coffee,my,1
my,friend,1
```

As the format is so simple, we can write a program that produces such a file. In fact, we have already done so for the first part: it is in fact nothing else than a CSV file with frequency counts.

For the second part, we need a module that tells us wich words co-occur together. In other words, we want to know all *combinations* of words in a given string. Not very surprisingly, this method is called `combinations`. See the example below:

```
1  from itertools import combinations
2  words="Let's combine stuff!".split()
3  print ([e for e in combinations(words,2)])
```

produces:

```
1  [("Let's", 'combine'), ("Let's", 'stuff!'), ('combine', 'stuff!')]
```

We already see that it is probably wise to do some preprocessing (to remove the punctuation for example, but you can also think of stopword removal, stemming, or converting to lowercase. We will leave that for know, you already know how to do this and can add that yourself.

There is one thing we still have to consider: We have an undirected network and do not want to distinguish between (`'combine'`, `'stuff!'`) and (`'stuff!'`, `'combine'`). We can construct the following minimal example, where we print a list of edges, based on co-occurrances of words in a set of three tweets. Do you see how we use lines 8 and 9 to turn around (`'stuff!'`, `'combine'`) if we already have (`'combine'`, `'stuff!'`) in our dict of edges that we construct? Also look at line 10, where we specify that a combination is only added to the dict (in line 11) if both nodes are not identical.

```
1   from collections import defaultdict
2   from itertools import combinations
3   tweets=["i am having coffee with my friend","i like coffee","i like coffee
        and beer","beer i like"]
4   cooc=defaultdict(int)
5   for tweet in tweets:
6       words=tweet.split()
7       for a,b in set(combinations(words,2)):
8           if (b,a) in cooc:
9               a,b = b,a
10          if a!=b:
11              cooc[(a,b)]+=1
12  for combi in sorted(cooc,key=cooc.get,reverse=True):
13      print (cooc[combi],"\t",combi)
```

This gives you:

```
1  3        ('i', 'coffee')
2  3        ('i', 'like')
3  2        ('like', 'coffee')
4  2        ('i', 'beer')
5  2        ('like', 'beer')
6  1        ('i', 'am')
7  1        ('am', 'friend')
```

You know have all building blocks you need to write a program that produces a GDF file. You would need to add

- something that reads your input from a file (e.g., a CSV file of tweets or a JSON file of speeches or a set of individual TXT files with newspaper articles)

- some preprocessing

- maybe some filter to filter out nodes and edges that occur very few times

- something to write the output to a file rather than printing to the screen.

The GDF file your program produces can be opened in Gephi (or some other network visualization or analysis tool). I made a screencast where I quickly show how to use Gephi. You can watch it here: `https://streamingmedia` `.uva.nl/asset/detail/t2KWKVZtQWZIe2Cj8qXcW5KF`.

> **!** Rather than producing a GDF file and opening it in Gephi, you can also do network analysis in Python itself, for example with the module `networkx`. This can be very interesting, especially for really large networks (which you maybe cannot open with a graphical tool like Gephi).

9.2   Recap

This chapter taught you how to deal with co-occurrences and similar structures from a network point of view and how to visualize them using Gephi. More in general, you also should have seen how one can write own programs to create specific file formats, even if they are not natively supported.

Chapter 10

Machine learning and topic modelling

The vastness of the topic makes it impossible to cover it extensively in this course. Nevertheless, as last part, we will take a very small and superficial glimpse in the world of supervised and unsupervised machine learning. The lecture slides should give you a bit more context, but here you find two examples to illustrate the general principle — and to give you a starting point for your own investigations. We will use scikit-learn (Pedregosa et al., 2011) and gensim (Řehůřek & Sojka, 2010), two packages that are very widely used and for which you will find a lot of tutorials and usage examples on the web.

10.1 Supervised machine learning

For this demonstration, we use a dataset from the Internet Movie Database (Maas et al., 2011). It contains in total 50000 reviews, separated into a test dataset and a training datasset (25000 each), half of them positive, half of them negative.

Download and unpack the dataset:

```
1 cd /home/damian
2 wget http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
3 tar -xzf aclImdb_v1.tar.gz
4 rm aclImdb_v1.tar.gz
```

You now have a folder `aclImdb` in your home directory. Take a look at the structure of the dataset, maybe read the documentation that is included.

You should probably see rather quickly that there are two folders with an identical structure, `train` and `test`. Within each of them, there are two

folders `pos` and `neg`, with a lot of plain text files within them. Have a look at them.

First of all, we need the training dataset and the test dataset. Lets conceptualize them as a list of tuples where all positive reviews get a 1, and all negative ones a −1:

```
reviews=[("This is a great movie",1),("Bad movie",-1), ... ...]
test=[("Not that good",-1),("Nice film",1), ... ...]
```

Of course, we do not want to insert our 50000 reviews by hand, but read them from the files we downloaded. We did something similar already with for example the LexisNexis articles or the lists of positive or negative words. Let's do it using the package `glob` that makes our live easier here:

```
from glob import glob

reviews=[]

for file in glob ("/home/damian/aclImdb/train/pos/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),1))

for file in glob ("/home/damian/aclImdb/train/neg/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),-1))
```

After that, we do exactly the same for the test dataset. You only have to replace `reviews` with `test` in the code above, and `train` with `test`.

Once we have done this, and thus have two lists, `reviews` and `test`, each of them with a length of 25000, we can train and test the classifier — after importing the necessary modules from the package scikit-learn (Pedregosa et al., 2011), of course:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics
```

We use a Bag-of-Word representation of all reviews rather than the whole reviews. We only want to know the frequency of each word in each review. While we could calculate this ourselves, scikit-learn integrates this nicely in the workflow (they even remove stopwords for us):

```
vectorizer = CountVectorizer(stop_words='english')
train_features = vectorizer.fit_transform([r[0] for r in reviews])
test_features = vectorizer.transform([r[0] for r in test])
```

> **!** Note that `[r[0] for r in reviews]` is a short form of writing something like the following (it is called "list comprehension"):

```
newlist=[]
for r in reviews:
    newlist.append[r[0]]
```

> It gives us a list of the reviews themselves, without the scores – and `r[1]` would give us only the scores.

Training the machine learning algorithm (we chose the Multinomial Naïve Bayes variant here, there are others as well – read the docs and check out which one is best!) takes only two lines:

```
1  # Fit a naive bayes model to the training data.
2  nb = MultinomialNB()
3  nb.fit(train_features, [r[1] for r in reviews])
```

Let's now use the classifier we just trained to predict the classification of our test dataset:

```
1  predictions = nb.predict(test_features)
2  actual=[r[1] for r in test]
```

(of course, we already know the actual, real classes, we just put them in a list in the second line for easier comparison)

We could compare the two lists `predictions` and `actual` by hand to find out in where our classifier is right or wrong. But we can also just calculate the AUC, a measure of accuracy:

```
1  print(metrics.accuracy_score(actual,predictions,normalize=True))
```

We can now play around and see how the classifier classifiers some new data:

```
1  newreviews=["What a crappy movie! It sucks!",
2          "This is awsome. I liked this movie a lot, fantastic actors",
3          "I would not recomment it to anyone.",
4          "Enjoyed it a lot"]
5
6  new_features=vectorizer.transform(newreviews)
7  predictions = nb.predict(new_features)
8  print(predictions)
```

10.1.1 Comparing different classifiers and vectorizers

Usually, when we de supervised machine larning, we want to compare different classifiers. For example, in the last section, we used a Naïve Bayes classifier. But we could use a logistic regression as well, or a so-called support vector machine. It is common to run several of these models and then compare their *precision* and *recall*.

Let us assume that the goal of training above-mentioned classifier is to build an app that shows the user only films we can expect to receive positive ratings. There are two things that we want to achieve: We want to find as many as possible positive films (recall), but we also want that the selection we found *only* contains positive films (precision).

Precision is calculated as $\frac{TP}{TP+FP}$, where TP are true postivies and FP are false positives. For example, if our classifier retrieves 200 articles that it classifies as positive films, but only 150 of them indeed are positive films, then the precision is $\frac{150}{150+50} = \frac{150}{200} = 0.75$.

Recall is calculated as $\frac{TP}{TP+FN}$, where TP are true postivies and FN are false negatives. If we know that the classifier from the previous paragraph missed 20 positive films, then the recall is $\frac{150}{150+20} = \frac{150}{170} = 0.88$.

In other words: Recall measures how many of the cases we wanted to find we actually found. Precision measures how much of what we have found actually is correct.

Often, we have to make a trade-off between precision and recall. For example, just retrieving *every* film would give us a recall of 1.0 (after all, we didn't miss a single positive film). But on the other hand, we retrieved all the negative films as well, so precision will be extremely low. It can depend on the task at hand whether precision or recall is more important.

Another thing we can vary in our classifier is the vectorizer we used. In our example, we used a count vectorizer, which simply means that our *features* (which is just a fancy word for independent variables) are simply the frequency counts of the words. This approach has the disadvantage that some words will be very frequent in almost all articles, while others occur in only a few articles. Arguably, such words are more informative and therefore, their occurrence should weigh more heavily. The tf·idf scheme does exactly that: it weighs the term frequency (TF) by the inverse document frequency (IDF), i.e. the number of texts ("documents") in which the word ("term") occurs at least once.

Scikit-learn allows us to use tf·idf scores instead of simple counts by using another vectorizer[1]:

```
1  from sklearn.feature_extraction.text import CountVectorizer,
       TfidfVectorizer
2  # instead of:
3  myfirstvectorizer=CountVectorizer(stop_words='english')
4  # we could do:
5  mysecondvectorizer=TfidfVectorizer(stop_words='english')
```

---

[1]For details, see http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

Below, you find an example script that combines everything that we discussed in these sections. There are more elegant ways of doing this, for example by writing functions instead of repeating the code that calculates the evaluations, but for illustration purposes, I decided to leave it as it is:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from glob import glob
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics

reviews=[]
test=[]

print("Constructing training dataset")

# glob gives you a list of filenames that match a specific criterion
# in this case, all .txt-files in the postivie training folder

for file in glob ("/home/damian/aclImdb/train/pos/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),1))
nopostr=len(reviews)

print ("Added",nopostr,"positive reviews")

for file in glob ("/home/damian/aclImdb/train/neg/*.txt"):
    with open(file) as fi:
        reviews.append((fi.read(),-1))
nonegtr=len(reviews)-nopostr
print ("Added",nonegtr,"negative reviews")

print("Constructing test dataset")

for file in glob ("/home/damian/aclImdb/test/pos/*.txt"):
    with open(file) as fi:
        test.append((fi.read(),1))
noposte=len(test)
print ("Added",noposte,"positive reviews")

for file in glob ("/home/damian/aclImdb/test/neg/*.txt"):
    with open(file) as fi:
        test.append((fi.read(),-1))
nonegte=len(test)-noposte
print ("Added",nonegte,"negative reviews")

#%% Checking the data
# Thus, we got two lists, reviews and test.
```

```
45  # Both contain tuples (pairs of two values: The first is the review,
46  # the second the classification: 1 or -1)
47
48  # We can easiliy verify this by looking at a random entry:
49  print(reviews[244])
50  # or
51  print('The following review\n\n\n',reviews[244][0],"\n\n\nis evaluated as
         ",reviews[244][1])
52
53  #%% Training the classifier
54
55  print("Training classifier...")
56
57  # Generate BOW representation of word counts
58  vectorizer = CountVectorizer(stop_words='english')
59  #alternatively, you could provide a list of stop words yourself
60  train_features = vectorizer.fit_transform([r[0] for r in reviews])
61  test_features = vectorizer.transform([r[0] for r in test])
62
63  # Fit a naive bayes model to the training data.
64  nb = MultinomialNB()
65  nb.fit(train_features, [r[1] for r in reviews])
66
67  #%% testing the classifier
68  # Now we can use the model to predict classifications for our test
         features.
69  predictions = nb.predict(test_features)
70  # and also put the 'true' results from the test dataset in a list
71  actual=[r[1] for r in test]
72
73  # now we can compare whether the predicted values and the actual values
         match.
74  # We could write the output to a tab seperated file to see what matches:
75  with open("/home/damian/agreement.tab", mode='w') as fo:
76      fo.write("actual\tpredicted\tfirst words\n")
77      for i in range(len(predictions)):
78          fo.write(str(actual[i])+"\t"+str(predictions[i])+"\t"+test[i
                ][0][:50]+"\n")
79
80  # That can be helpful for inspecting where sth goes wrong, but we can
81  # also calculate some measures of performance immediately:
82
83  print('Accuracy:')
84  print(metrics.accuracy_score(actual,predictions,normalize=True))
85
86  print('Precision:')
87  print(metrics.precision_score(actual,predictions,pos_label='1', labels =
         ['-1','1']))
88  print('Recall:')
```

```
89  print(metrics.recall_score(actual,predictions,pos_label='1', labels =
        ['-1','1']))

90
91  # Note that Precision is not a symmetric measure.
92  # If we want the precision for retrieving a NEGATIVE review # instead of a
        positive we get a different value:

93
94  print('\t side note: Precision if we are interested in retrieving negative
        reviews:')
95  print('\t',metrics.precision_score(actual,predictions,pos_label='-1',
        labels = ['-1','1']))
96  print('\t side note: Recall if we are interested in retrieving negative
        reviews:')
97  print('\t',metrics.recall_score(actual,predictions,pos_label='-1', labels
        = ['-1','1']))

98
99  # back to normal

100
101  print('F1-score:')
102  print(metrics.f1_score(actual,predictions,pos_label='1', labels =
        ['-1','1']))
103  print('Confusion matrix:')
104  print(metrics.confusion_matrix(actual,predictions))

105

106
107  #%% Now, let's play around and see how well it would work on
108  # new unseen data:
109  newreviews=["What a crappy movie! It sucks!",
110             "This is awsome. I liked this movie a lot, fantastic actors",
111             "I would not recomment it to anyone.",
112             "Enjoyed it a lot"]
113  newdata=vectorizer.transform(newreviews)
114  predictions = nb.predict(newdata)
115  print(predictions)
116  for i in range(len(predictions)):
117      if predictions[i]==1:
118          print(newreviews[i],'\nis probably about a GOOD movie\n')
119      elif predictions[i]==-1:
120          print(newreviews[i],'\nis probably about a BAD movie\n')
```

We used a Naïve Bayes classifier, but we could as well use a different one, for example a logistic regression[2]

You can run a logistic regression classifier just as you would run a NB

---

[2]You can find one explanation of the difference here: `https://www.quora.com/What-is-the-difference-between-logistic-regression-and-Naive-Bayes?share=1`. Basically, in contrast to a logistic regression, in a Naïve Bayes classifier, all features are assumed to be uncorrelated.

classifier[3]:

```
1  from sklearn.linear_model import LogisticRegression
2  logreg = LogisticRegression()
3  logreg.fit(train_features, [r[1] for r in reviews])
4  predictions = logreg.predict(test_features)
```

A last one would be a support vector macine (SVM) (for more info, see `http://scikit-learn.org/stable/modules/svm.html`)

```
1  from sklearn import svm
2  mysvm = svm.SVC()
3  mysvm.fit(train_features, [r[1] for r in reviews])
4  predictions = mysvm.predict(test_features)
```

Note that the names I assigned to each classifier (`nb`, `logreg`, `mysvm`) are completely arbitrary.

10.1.2   Saving the trained model

In theory, you could just train the model everytime you want to use it. However, that does not only take unnecessary time and ressources, but it also requires you to have the training data at hand. Therefore, it can be useful to save your vectorizers and classifiers. It is important to realize that you need to save both: After all, the vectorizer determines which word is mapped to which internal numeric representation, that is used by the classifier.

Once you have fitted both vectorizer and classifier, you can save them as follows (assuming you have called your instance of the vectorizer `vectorizer` and your instance of the classifier `nb`):

```
1  import pickle
2  from sklearn.externals import joblib
3
4  pickle.dump(vectorizer,open("myvectorizer.pkl",mode='wb'))
5  joblib.dump(logreg, 'myclassifier.pkl')
```

Then, later on, instead of fitting a new vectorizer, you can simply load the old one and use it

```
1  import pickle
2  vectorizer = pickle.load(open("myvectorizer.pkl",mode='rb'))
3  new_features = vectorizer.transform([listwithnewdata])
```

---

[3]We could get more info, see `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`. For example, if you are interested in coefficients:
```
predictionsproba = logreg.predict_proba(test_features)
print([j for i in logreg.coef_ for j in i])
```

As you see, you do not do any `.fit_transform` any more, because the vectorizer is already fitted (that was why we saved it, after all).

Also the classifier can be loaded again very easily and be used immediately for prediction

```
1  from sklearn.externals import joblib
2  nb = joblib.load('myclassifier.pkl')
3  predictions = nb.predict(new_features)
```

10.2 Latent Dirichlet Allocation (LDA)

In the previous chapter, we were had *labeled* data, i.e. we had an outcome which we could predict. In other words, we applied supervised machine learning. But sometimes, we do not have an outcome to predict.For instance, we want to group the films into topics, but we have no idea which topics exist to begin with.

Topic Modelling therefore is a form of *unsupervised* machine learning. To get to know it, we will use the gensim package (Řehůřek & Sojka, 2010).

Furthermore, let us assume you have a list of lists of words (!) called `texts`:

```
1  articles=['The tax deficit is higher than expected. This said xxx ...', '
      Germany won the World Cup. After a']
2  texts=[art.split() for art in articles]
```

which looks like this:

```
1  [['The', 'tax', 'deficit', 'is', 'higher', 'than', 'expected.', 'This', '
      said', 'xxx', '...'], ['Germany', 'won', 'the', 'World', 'Cup.', '
      After', 'a']]
```

I'll take the movie reviews from the last section as a test case (without the ratings, which are irrelevant now), but of course you can use any collection of text files.

```
1  from glob import glob
2  texts=[]
3  for file in glob ("/home/damian/aclImdb/train/pos/*.txt"):
4      with open(file) as fi:
5          texts.append(fi.read().split())
```

Pay attention to the `.split()` method, which makes that we now have a list of lists of words, just as we wanted.

We now let gensim create a BOW representation of the texts — just as in earlier examples, but each packages has a slightly different syntax for that. Luckily, you don't have to remember that, that's where the documentation and example scripts that come with each package are for.

```
1  from gensim import corpora, models
2  # Create a BOW represenation of the texts
3  id2word = corpora.Dictionary(texts)
4  mm =[id2word.doc2bow(text) for text in texts]
```

We can now train the LDA models:

```
1  lda = models.ldamodel.LdaModel(corpus=mm, id2word=id2word, num_topics=100,
       alpha="auto")
```

Of course, you can specify any other number of topics. Most people use something like 50 or 100 topics, though. Let's print the most characteristic words for each topic:

```
1  for top in lda.print_topics(num_topics=NTOPICS, num_words=5):
2      print ("\n",top)
```

Only one thing left to do: Calculate the scores for each topic per document and save them to a file. I chose a tab-separated one:

```
1  scoresperdoc=lda.inference(mm)
2  with open("topicscores.tsv","w",encoding="utf-8") as fo:
3      for row in scoresperdoc[0]:
4          fo.write("\t".join(["{:0.3f}".format(score) for score in row]))
5          fo.write("\n")
```

This is the general principle, but of course, there are a lot of ways of tuning it. To start with, some stopword removal would be advisable - and we did not even remove punctuation or all the weird HTML tags, or bother about converting it to lower case. However, you already learned all this and should be able to write a good script by integrating the structure from above with other techniques.

For some suggestions, see the documentation of gensim.

One particular thing you might want to try, though: Just as in the case of supervised machine learning, also for unsupervised machine learning, you might want to try to use a tf·idf vectorizer instead of the default count vectorize. In gensim, you can do so by slightly modifying the script presented above:

```
1  from gensim import corpora, models
2  # Create a BOW represenation of the texts
3  id2word = corpora.Dictionary(texts)
4  mm =[id2word.doc2bow(text) for text in texts]
5
6  # create a tf-idf representation
7  tfidf = models.TfidfModel(mm)
8
9  # use that tfidf-representation instead of the pure counts
10 lda = models.ldamodel.LdaModel(corpus=tfidf[mm], id2word=id2word,
       num_topics=100, alpha="auto")
```

10.3 Recap

You should be able to explain what supervised and unsupervised machine learning are and give some examples. More practically, you should be able to use the following widely used packages to conduct such analysis:

- scikit-learn

- gensim

In particular, you should be able to not only train models and apply them to new, unseen data, but you should also know how to do a basic evaluation of the performance of the model.

Chapter 11

Statistics with Python

One of the things you have learned so far is how to output data in a format that suits your needs. For example, no matter what your input data are, you can always save the results of your program in a CSV file that you can open with R, Stata or SPSS for further analysis. And indeed, it can make sense to use different tools in different stages of a project.

However, if you only want to calculate a mean and a standard deviation or even conduct a regression, this is actually not neccessary – and, let's face it, is pretty cool if your program just does *everything* in a single run. In addition, Python is actually pretty good in doing statistics and used by many data scientists for this purpose. In this chapter, I'll briefly mention some modules that are useful for this, so that you know where to look further.

11.1   numpy & scipy

Numpy (Van der Walt, Colbert, & Varoquaux, 2011) is a package that provides a lot of mathematical functions. It can, for example, be used to calculate means (ok, that's boring, you could do that yourself), standard deviations, correlations, and much, much more. It also offers some specific data types that are more efficient for some calculations than the build-in lists. It goes a long with the similar package SciPy (Jones, Oliphant, Peterson, et al., 2001).

```
1  import numpy as np
2  >>> x = [1,2,3,4,3,2]
3  >>> y = [2,2,4,3,4,2]
4  >>> np.mean(x)
5  2.5
6  >>> np.std(x)
7  0.9574271077563381
```

```
8  >>> np.corrcoef(x,y)
9  array([[ 1.       ,  0.67883359],
10        [ 0.67883359, 1.       ]])
11
12 >>> from scipy import stats
13 >>> stats.skew(x)
14 0.0
15 >>> stats.kurtosis(x)
16 -0.942148760330578
```
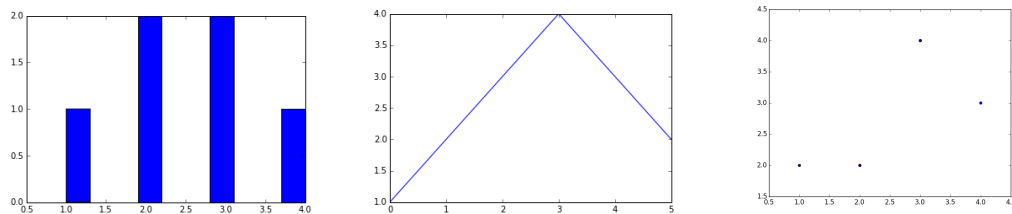
11.2   matplotlib

Figure 11.1: Examples of plots generated with `matplotlib`

While there are tools that make fancier graphics (take a look at the Python module `seaborn`; or use `ggplot2` in R), `matplotlib` (Hunter, 2007) is really useful if you just want to make a quick histogram, line graph, or scatter plot (Fig. 11.1).

```
1  import matplotlib.pyplot as plt
2  x = [1,2,3,4,3,2]
3  y = [2,2,4,3,4,2]
4  plt.hist(x)
5  plt.plot(x,y)
6  plt.scatter(x,y)
```

11.3   pandas & statsmodels

Pandas (McKinney et al., 2010) is a framework that allows statistical modelling in Python. Those of you who know R will see a lot of similarities. Within pandas, data are stored in a table, very much like a Stata or SPSS dataset – and yes, it is indeed called exactly as it is called in R: a *dataframe*. As you see below, a dataframe basically consists of different columns with a

name, which consist of a list of data. Pandas can read directly from a CSV file, but as you can see below, you can also construct a dataframe yourself.

While pandas has a lot of nice functions to explore and manage datasets, it is – for our purposes – especially powerful when combined with statsmodels (Seabold & Perktold, 2010). Statsmodels allows provides a wide range of models, just like what you would expect from SPSS, Stata, or R.

To run an OLS regression, for example, you only have to specify which columns are the dependent and the independent variables.

```python
import pandas as pd
from statsmodels.formula.api import ols
df = pd.DataFrame({"income": [10,20,30,40,50], "age": [20, 30, 10, 40,
    50], "facebooklikes": [32, 234, 23, 23, 42523]}
# alternative: read from CSV file:
# df = pd.read_csv('mydata.csv')

myfittedregression = ols(formula='income ~ age + facebooklikes', data=df).
    fit()
print(myfittedregression.summary())
```

prints a regression table like you would expect from any statistics program:

```
OLS Regression Results
==============================================================================
Dep. Variable:                 income   R-squared:                       0.579
Model:                            OLS   Adj. R-squared:                  0.158
Method:                 Least Squares   F-statistic:                     1.375
Date:                Mon, 31 Oct 2016   Prob (F-statistic):              0.421
Time:                        18:11:40   Log-Likelihood:                 -18.178
No. Observations:                   5   AIC:                             42.36
Df Residuals:                       2   BIC:                             41.19
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
Intercept      14.9525     17.764      0.842      0.489     -61.481     91.386
age             0.4012      0.650      0.617      0.600      -2.394      3.197
facebooklikes   0.0004      0.001      0.650      0.583      -0.002      0.003
==============================================================================
Omnibus:                          nan   Durbin-Watson:                   1.061
Prob(Omnibus):                    nan   Jarque-Bera (JB):                0.498
Skew:                          -0.123   Prob(JB):                        0.780
Kurtosis:                       1.474   Cond. No.                     5.21e+04
==============================================================================
```

Pandas is a *huge* framework, especially in combination with Jupyter Notebook (Section 3.5). It is extremely popular in the world of data science, but also in areas like finance.

However, it would be kind of useless to cover it extensively in this intro, as others have already done so. You can have a look at the books by McKinney (2012) and Russel (2013).

You can also find some iPython notebooks with typical analyses with pandas, ranging from correlations and t-tests to regression models and time-series analysis, here: `https://github.com/damian0604/bdaca`.

11.4 Recap

Things you should remember:

- There are multiple packages for doing statistical analysis like you know them from programs like SPSS, Stata, or R.

- pandas offers you R-like data structures.

- There are online ressources with a lot of examples.

In particular, once you arrived at this part of this book, you can run *your whole workflow* in Python — from collecting the data until the final analysis. This obviously has major advantages compared to switching between environments.

Chapter 12

Further reading

The following books provide the interested student with more and deeper information. They are intended for the advanced reader and might be useful for your individual projects (or, maybe, a thesis):

- Russel, 2013. Gives a lot of examples about how to analyze a variety of online data, including Facebook and Twitter, but going much beyond that. Because social media APIs have undergone multiple changes in the last years, the examples might be slightly outdated. A PDF of the book can be downloaded for free on `http://www.webpages.uidaho.edu/%7Estevel/504/Mining-the-Social-Web-2nd-Edition.pdf`

- Bird et al., 2009. This is the official documentation of the NLTK package that we are using. A newer version of the book can be read for free at `http://nltk.org`

- McKinney, 2012: Another book with a lot of examples. A PDF of the book can be downloaded for free on `http://it-ebooks.info/book/1041/`.

In the last years, some other tutorials, partly similar to this one, have been published. See for exampe:

- Jürgens & Jungherr, 2016. Provides examples and code for non-pogrammers to analyze Twitter data using Python and R.

Appendix A

Exercise 1: Describing an existing structured dataset

A.1 Downloading the data

In this exercise, you will do some basic descriptive analyses of an existing dataset. Mazières, Trachman, Cointet, Coulmont, and Prieur (2014) scraped data from online porn sites, resulting in two datasets with metadata about almost two million amateur porn videos. We will work with one of these datasets, consisting of all metadata on all videos posted on `http://xhamster.com` from its creation in 2007 until February 2013. The authors made the dataset available on on `http://sexualitics.github.io`, and you can find a description of it in Figure A.1.

We do the exercise together in class, but make sure you have downloaded the dataset before class. You can do so as follows (but, of course, replace "damian" with your own user name):

```
1  cd /home/damian
2  mkdir pornexercise
3  cd pornexercise
4  wget pornstudies.sexualitics.org/data/xhamster.json.tar.gz
5  tar -xzf xhamster.json.tar.gz
```

The `wget` command downloads the dataset. It is compressed, so we have to uncompress it, which is done by the `tar` command (most of you probably are used to having `.zip` files for compressed or archived data, which is essentially the same; `.tar.gz` is more common among the nerdier part of the population). Lets check if everything went right:

```
1  ls -lh
```

should give you an output like this:

```
1  damian@damian-VirtualBox:~/pornexercise$ ls -lh
2  total 284M
3  -rw-r--r-- 1 damian damian 229M feb 8 2014 xhamster.json
4  -rw-rw-r-- 1 damian damian 55M feb 8 2014 xhamster.json.tar.gz
```
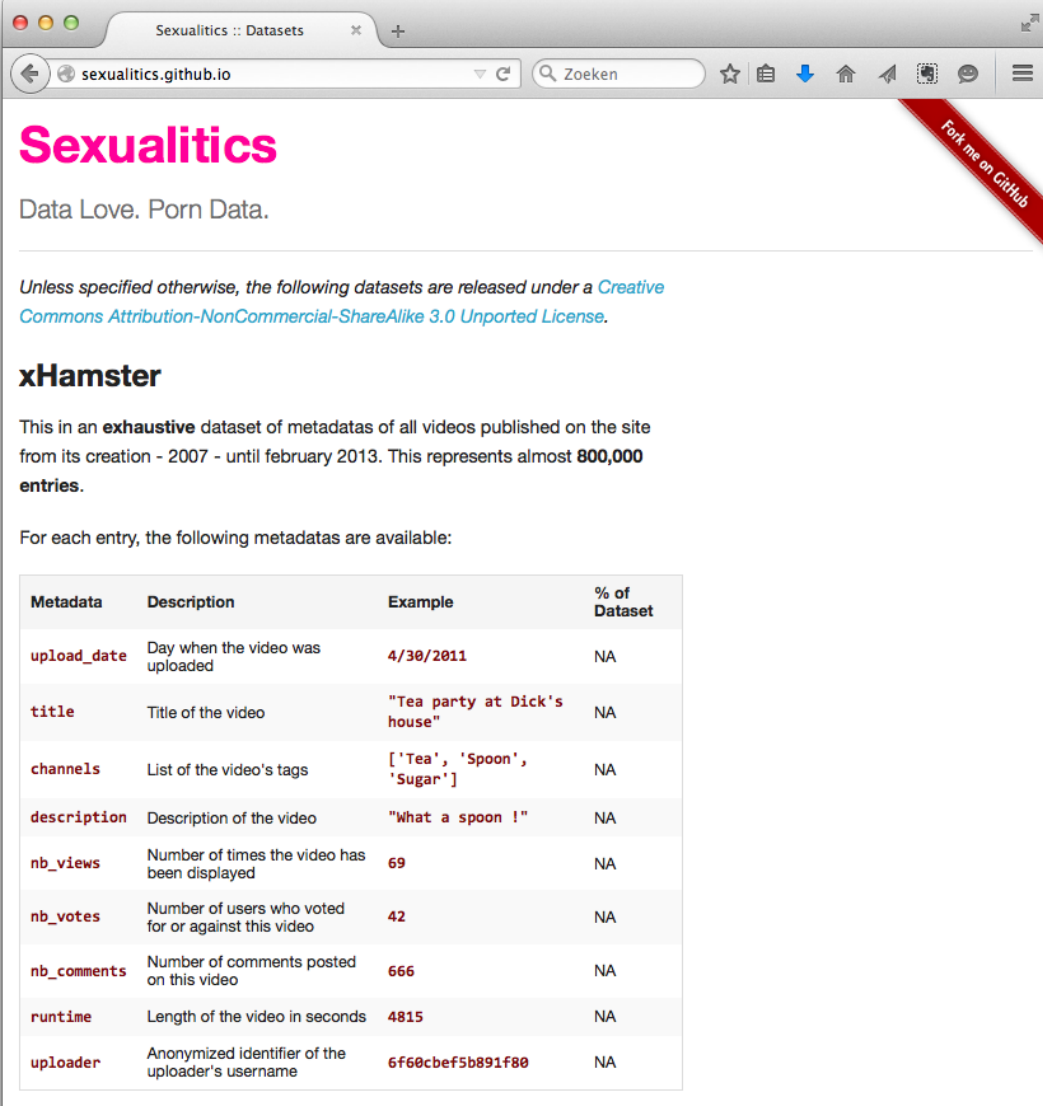
You see that the compressed file is 55MB large, but the uncompressed one is more than four times as large. Let's delete the compressed one, we don't need it any more:

```
1  rm xhamster.json.tar.gz
```

A.2   The tasks

Start with having a look at Figure A.1. It is important to understand the structure of the data: Which fields are there, how are they named, and what do they contain? For example, we see that the field "channels" contains a *list* of different tags, while "nb_votes" seems to contain an *integer*. Ready to go? Let's do some work:

1. Print the title of each video.

2. (a) What are the 100 most frequently used tags?

   (b) What is the average number of tags per video?

3. What tags generate the most comments/votes/views?

4. What is the average length of a video description?

5. What are the most frequently used words in the descriptions?

Figure A.1: The discription of the dataset.

Appendix B

Exchanging files by mounting your host system

One of the nice things of using a Virtual Machine is that you cannot break anything on your own computer. The obvious downside is that you also cannot access files on it. Of course, you can always mail stuff to yourself or use some other workaround.

But if you *really* want to break the isolation and have access to folders on your computer, here is how it works.

The process of connecting a device (a USB stick, a harddisk, ...) and assigning it a name is called "mounting". For example, when you insert a USB stick in a Mac or Linux computer, it might be (nowadays, usually automatically) mounted on `/media/mystick` or something similar.

What you want to do, is mounting a directory from your "real" computer, the so-called host, in the file system of the VM. Select the folder you want to share in Virtual Box (see Figure B.1). Select permanent, but NOT auto-mounting (screenshot). Remember the name of the share (in my example, Desktop)

Within the VM, now create a folder where you want the content of your host folder to appear. For example, this could be:

mkdir /home/damian/myrealcomputer

Now, use the following line to mount the share (in my example, it is called Desktop) to that folder:

```
sudo mount -t vboxsf -o uid=$UID,gid=$(id -g) Desktop /home/damian/
    myrealcomputer/
```

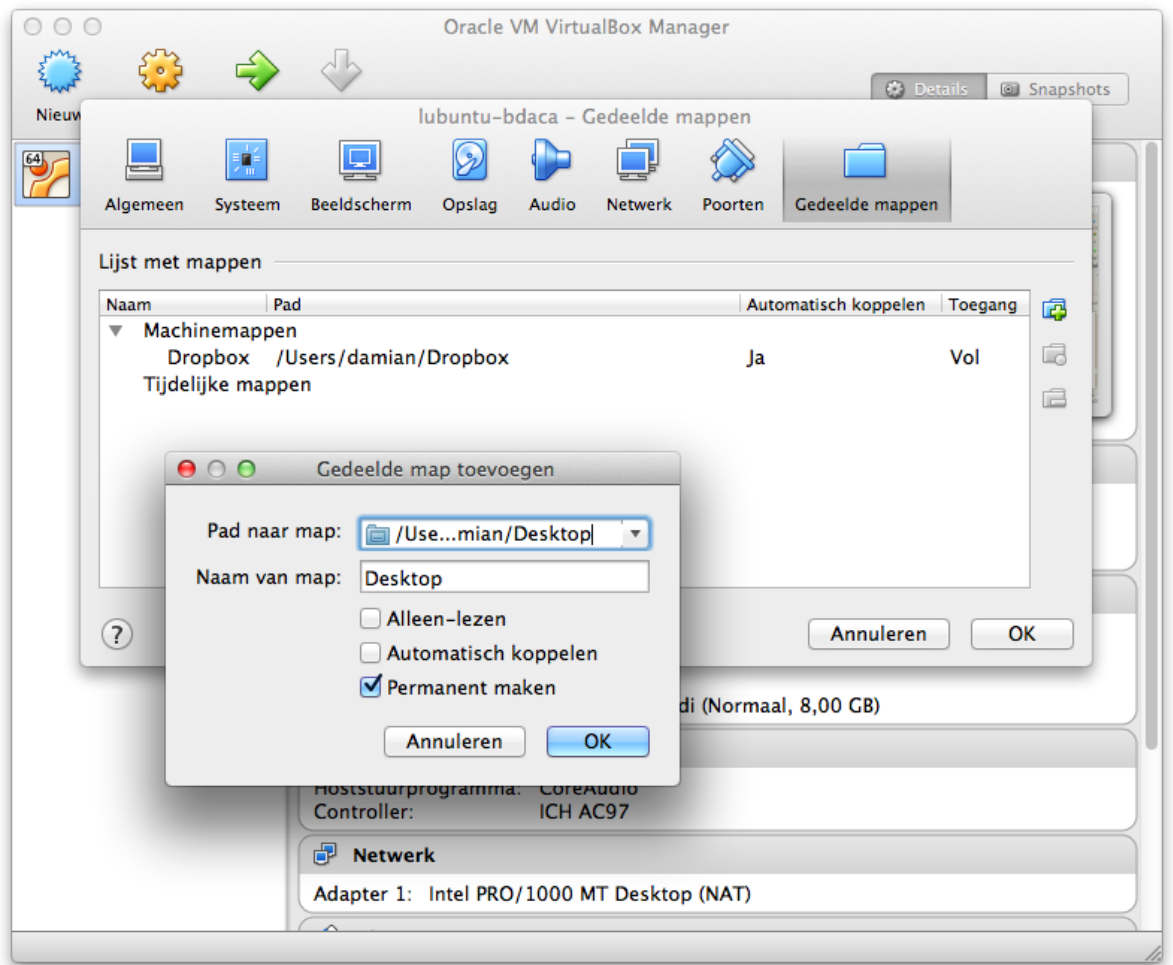Voilà!

Figure B.1: Configuring VirtualBox to make it possible to mount folders from the host system

Appendix C

Installing Python on other systems

Before you install Python outside of the virtual machine, read this so that you know what you are doing. While you could just go to `http://python.org` and download Python for your operating system, this may or may not be a good choice. First, Python might already be installed (which you could find out by typing `python` or `python3` in your Terminal). Second, even if it is not, if you prefer one-stop-shopping above dealing with packages, you might like Anaconda (see below).

Keep in mind that one can have multiple versions of Python on one computer, and you don't want to litter your system with a lot of different versions that you then mix up accidentally.

In this book, we use a virtual machine with Linux as operating system. However, if you want to install Python somewhere else, chances are high that it isn't Linux. Within Linux, we used the system's package manager `apt-get` to install Python itself (and maybe some other programs). We then used `pip` to install specific Python packages. This requires you to know which packages you want (that's why we installed a bunch of them on page 4), but on the other hand, if you forgot one, it doesn't really matter because you can easily just install it later on.

All very easy, it seems, but to install some (few) packages via pip one needs specific programs like, for instance, a C compiler. You probably didn't even realize this, because on Linux, this stuff is usually present, and if not, it can be easiliy installed via `apt-get`. MacOS *sometimes* has the necessary stuff installed (usually if you have installed XCode via the App Store). But on a common Windows installation, things can quickly become complicated (unless you know exactly what you are doing).

Because people do not really like dealing with all this stuff, there is an

easier solution: Anaconda `https://www.continuum.io/downloads`.

Anaconda is a platform that includes Python together with a lot of commonly used scientific Python packages preinstalled. In addition, it has its own package manager, `conda`, which can solve some dependencies `pip` cannot resolve.

So, if you want to use Python outside of the virtual machine that we used in this book, you might want to give Anaconda a try.

One important characteristic of anaconda is that it installs an own Python installation in your home directory. To quote from their documentation:

```
1  On Windows this might be a path such as C:\Users\Jane Smith\anaconda\bin\
       python.
2  On macOS this might be a path such as /Users/jsmith/anaconda/bin/python.
3  On Linux this might be a path such as /home/jsmith/anaconda/bin/python.
4  As well as anaconda, the folder in your home directory might be named
       anaconda2 or anaconda3.
```

Because you might *also* have a version of Python already installed on your system (at least on Linux and MacOS, this is generally the case), you have to make sure that you actually run the correct version: If you just type

```
1  python
```

in your Terminal, then you probably will *not* start the anaconda version – and thus be unable to use the packages installed with anaconda. You would have to explicitly refer to the location where anaconda is installed.

References

Bird, S., Loper, E., & Klein, E. (2009). *Natural language processing with Python.* Sebastopol, CA: O'Reilly.

Boumans, J. W., & Trilling, D. (2016). Taking stock of the toolkit: An overview of relevant autmated content analysis approaches and techniques for digital journalism scholars. *Digital Journalism*, *4*(1), 8–23. doi: 10.1080/21670811.2015.1096598

De Smedt, T., & Daelemans, W. (2012). Pattern for Python. *The Journal of Machine Learning Research*, *13*, 2063–2067.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi: 10.1109/mcse.2007.55

Hutto, C. J., & Gilbert, E. (2014). Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth international aaai conference on weblogs and social media.*

Jones, E., Oliphant, T., Peterson, P., et al. (2001). *SciPy: Open source scientific tools for Python.* Retrieved from `http://www.scipy.org/`

Jürgens, P., & Jungherr, A. (2016). A tutorial for using Twitter data in the social sciences: Data collection, preparation, and analysis. *SSRN*. doi: 10.2139/ssrn.2710146

Kitchin, R. (2014). Big Data, new epistemologies and paradigm shifts. *Big Data & Society*, *1*(1), 1–12. doi: 10.1177/2053951714528481

Lazer, D., Pentland, A., Adamic, L., Aral, S., Barabási, A.-L., Brewer, D., ... van Alstyne, M. (2009). Computational social science. *Science*, *323*(February), 721–723. doi: 10.1126/science.1167742

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies* (pp. 142–150). Portland, Oregon, USA: Association for Computational Linguistics. Retrieved from `http://www.aclweb.org/anthology/P11-1015`

Mazières, A., Trachman, M., Cointet, J.-P., Coulmont, B., & Prieur, C. (2014). Deep tags: toward a quantitative analysis of online pornogra-

phy. *Porn Studies*, *1*(1-2), 80-95. doi: 10.1080/23268743.2014.888214

McKinney, W. (2012). *Python for data analysis.* Sebastopol, CA: O'Reilly.

McKinney, W., et al. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th python in science conference* (Vol. 445, pp. 51–56).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., . . . Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Řehůřek, R., & Sojka, P. (2010). Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (pp. 45–50). Valletta, Malta: ELRA. (`http://is.muni.cz/publication/884893/en`)

Russel, M. (2013). *Mining the social web. Data mining Facebook, Twitter, LinkedIn, Google+, GitHub, and more* (2nd ed.). Sebastopol, CA: O'Reilly. Retrieved from `http://www.webpages.uidaho.edu/%7Estevel/504/Mining-the-Social-Web-2nd-Edition.pdf`

Seabold, S., & Perktold, J. (2010). Statsmodels: Econometric and statistical modeling with Python. In *9th Python in science conference.*

Thelwall, M., Buckley, K., & Paltoglou, G. (2012). Sentiment strength detection for the social web. *Journal of the American Society for Information Science and Technology*, *63*(1), 163–173. doi: 10.1002/asi.21662

Trilling, D. (2015). Two different debates? Investigating the relationship between a political debate on TV and simultaneous comments on Twitter. *Social Science Computer Review*, *33*(3), 259–276. doi: 10.1177/0894439314537886

Van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), 22–30. doi: 10.1109/mcse.2011.37

Van Atteveldt, W. (2008). *Semantic network analysis: Techniques for extracting, representing, and querying media content.* Charleston, SC: BookSurge.

Wilson, T., Wiebe, J., & Hoffmann, P. (2005). Recognizing contextual polarity in phrase-level sentiment analysis. In *Proceedings of the conference on human language technology and empirical methods in natural language processing* (pp. 347–354).