# Text Analytics and Natural Language Processing in Finance and Fintech

Matthias Buehlmaier, Ph.D.

December 9, 2025

# Contents

# Chapter 1

# Introduction

## 1.1 Why Is This Course Special?

This course is special because to the best of my knowledge, it was developed as the **first course worldwide** that covers text analytics and natural language processing (NLP) in finance. There are courses in NLP and courses in finance. But none of them cover how both of them work together and interact. This is increasingly becoming important, not only with the rise of fintech, but also in many other fields such as investment management, where often vast amounts of so-called alternative data is used, e.g. text from social media, company filings, or the financial press.

Another reason why this course is special is because it is **project-focused**. Students throughout the course work on projects of their own choosing where they write code and analyze text in a financial setting.

Finally, there are many resources available that explain isolated issues in NLP, but it is very difficult to find a coherent approach. Beginners are mostly overwhelmed by the material they can find in various places, and don't know how to tie everything together in a way that makes sense. This course fills this gap by showing you how to develop a **consistent and well thought-out workflow** for text analytics in finance, see for example Chapter 5 starting on page 37.

## 1.2 Why Are Text Analytics and NLP Useful in Finance and Fintech?

Why are text analytics and natural language processing (NLP) useful in finance? I briefly would like to provide two examples. The first is that you can use NLP in investment management. For example, you could analyze financial media, social media (e.g. Twitter), or company reports to make better investment decisions and ultimately get higher investment returns. Second, you could for example analyze the minutes of the Federal Reserve and calculate the probability of a raise interest rates, which is a trillion-dollar question.

For clarification, in this book we focus on the analysis of existing text, as this is a common use case in financial applications. In particular, we do not spend a lot of time on natural language generation or machine translation.

## 1.3 Important Announcements via HKU Email

Each student in this course is **expected to check his/her HKU email on a daily basis**. The reason is that this is the way the instructor might contact students. Furthermore, important announcements will be made on Moodle, and these announcements will not be repeated in class. These announcements will be forwarded to your HKU email address only. If a student misses important information or announcements sent to the HKU email address, the student will have to bear the consequences, which may include but are not limited to receiving a lower grade in this course, failing this course, or delaying/endangering the graduation.

## 1.4 Course Advice

- As Lev Konstantinovskiy has succinctly pointed out, **"NLP is 80% preprocessing."** So please keep in mind that NLP is mostly grunt work. This is a course about getting your "hands dirty" with NLP, not about a fancy marketing brochure about how fashionable NLP is.
- Text analytics and NLP is a very applied topic that is learned best by actually doing it. In analogy, you cannot learn to play the piano just by reading books. You need to actually do it and play the piano. That is why this course will put emphasis on students working on their own projects and presenting them in class.
- The instructor can point you into the right direction, help you to understand the key concepts, tell you which packages to use (these choices are very important as they can make or break your project), and provide examples. All of these are things you often cannot easily find online. Furthermore, the lecture notes and website contain many code examples.
- But for specific programming problems during your projects you should find out yourself how to write your code. Most (if not all) the packages we use are very well documented (e.g. on ReadTheDocs.org or on their own website) and surrounded by an active community (e.g. on StackOverflow.com, blog posts, etc.), so please make good use of it.
- We are mostly going to focus on analyzing **English** text. However, many of the methods we discuss also work with other languages, e.g. **Chinese**, sometimes with modifications.
- If anything in this document is written in italics (*like this*), it means that either I want to emphasize something, or it is a new concept mentioned/defined for the first time. It will be clear from the context which one it is, emphasize or define. Sometimes I will also write something in bold (**like this**) instead of using italics.

## 1.5 Links

- The electronic PDF version of these lecture notes contains **clickable links** in blue color. For example, www.buehlmaier.net. If you happen to have a printed version, feel free to try the electronic version of the lecture notes as they might be more convenient for clicking on links.

- Furthermore, reference to different **chapters/sections and their respective pages are also clickable** in the electronic version of the lecture notes and will take you directly to the relevant page. For example, Chapter 2 (click on the number after "Chapter").

- Why do many links in these lecture notes point to **GitHub**? Because that is where the original source code lives and the GitHub repos usually also contain brief introductions and pointers to other webpages. Usually you can find a link to the main website at the top of the GitHub page or in the README file that is displayed after the code listing.

## 1.6 Important Deadlines and Dates

- The group leader should hand in the group information to the instructor at the latest on the day after the add/drop period. For details see Section 2.4 starting on page 18.

- The **midterm** is tentatively scheduled in-class for **Wednesday, January 7, 2026**. The format and date of the midterm is subject to change.

- Blog posts: See Sections 2.2 starting on page 13.

- Presentation dates: See Section 2.3.2 starting on page 17.

- Group project: See Section 2.4 starting on page 18.

# Chapter 2

# Group Project

Throughout this course, you will work on projects related to text analytics and NLP in finance. This chapter contains detailed instructions and suggestions about the group projects.

## 2.1 Version Control and Collaborative Coding

- For more information on version control see Appendix D starting on page 229.
- Using version control for your project is **optional and not evaluated**. However, it might make your life easier in the long run if you put your code under version control and host it on GitHub.
- If you would like to use version control and if you do not already have a GitHub account, please sign up now. Please use sensible names for your GitHub user name and your GitHub repositories (or "repos" in short).

## 2.2 Blogging

### 2.2.1 Instructions

- From Wikipedia: "**A blog (a truncation of the expression "weblog") is a discussion or informational website published on the World Wide Web consisting of discrete, often informal diary-style text entries (posts).**"
- Each group should write **at least two blog posts** about the project the group is working on. The first blog post is **due three days after the midterm at 11:59pm**. The last blog post is **due at 11:59pm three days before the due date of the group project**, see Section 2.4 starting on page 18.
- The topic of the blog may be about the work you do for your group project. However, **the blog's content should be significantly different from the content**

**of your presentation slides and group report**, in other words, it should show a different perspective.

- The blog should be **reflective** in nature. For example, you can describe some technical or conceptual problems you encounter, and describe your journey and how you solved (or intend to solve) these problems. You can view the blog as a **reflective journal** that documents your learning progress.
- In any case, please **refrain from posting controversial opinions or topics on the blog.**
- The **evaluation criteria** are that the blog post
  - should be of high quality in terms of content, writing style, and execution,
  - should contain a clear description of what's going on and why it is important,
  - should include well-documented code snippets with explanations that illustrates key things about your project (any code posted should be self-contained and be fully reproducible in the sense that anyone reading the blog post can run the code and verify the results), and
  - should have a word count of at least 700 (source code is not included in the word count).
  - If you prefer to split up the description of your work across several smaller blog posts (instead of one big blog post) it is also fine. The evaluation of the blog is based on the blog's overall content. It is not based on the number of posts.
- One reason why blogging is a good idea is that, similar to posting your code on GitHub, it will increase your visibility and your chances on the job market. Blogging really is community engagement. Its usefulness cannot be overstated in establishing yourself as a thought-leader.
- It is not required, but feel free to mention your names on the blog if you like (e.g. for promoting yourself to potential future employers). But my recommendation is to NOT include your university student numbers on your blog for privacy reasons. Also, make sure to avoid posting login or password information, e.g. when including code snippets.
- The code in your blog post should not be simply a copy of all or most of your code. Instead, it should only be **one or several code snippet(s)**, i.e. a short piece of your code that illustrates what you are trying to convey in this part of the text of your blog post. You should not just "dump" a large chunk of code into the blog post, even less so without much explanation.
- While your blog should look professional, it is important that the **content** is of high quality in every conceivable way. It's not necessary to have a very fancy styling in terms of layout. A minimalistic layout is often a good idea if you want to focus on the written bog content and your code.

- Once you have written a new blog post, please **notify the instructor by email and announce your blog post within three days of its publication on the online forum of the course webpage** so that your classmates can take a look as well.

### 2.2.2 Blogging Software

- There is a blog website for this course, where students can publish their blog articles. It is available at buehlmaier.github.io/MFIN7036-student-blog-2025-12
- We use Pelican, which is a static website generator written in Python. You can find usage instructions for Pelican at docs.getpelican.com.
- The workflow for creating a new blog post is as follows (no knowledge of Git is required):
  1. Download the source code of the blog at github.com/buehlmaier/MFIN7036-student-blog-2025-12 (click on "Code" and then click on "Download ZIP")
  2. Write a first draft of your blog post in a Markdown file (e.g. `myblogpost.md`) and add it to the `content` directory; the easiest way to generate this file is to make a copy of the `demo-blog-post.md` file, rename it to `myblogpost.md`, and start modifying its content
  3. Open a terminal in the parent directory of the `content` directory
  4. Type the following command in the terminal:

     ```
     pelican content -t themes/elegant-5.4.0 --autoreload --listen
     ```
  5. Point your web browser to http://127.0.0.1:8000
  6. You can make changes to your markdown blog file (e.g. `myblogpost.md` as mentioned previously) and then reload your browser by pressing F5. Repeat this step as often as you like
  7. Once you are satisfied with your blog post, you can email the file(s) to the course instructor/TA for upload to the blog. The file(s) should include the markdown file of the blog post and potential additional files that are included in your blog post, e.g. picture(s)
- Please use the following naming conventions for any files submitted:
  - Markdown files: `group-name_XX_blog-post-title.md`, where `XX` is the two-digit number of your blog post, e.g. for the second blog post you would replace `XX` by `02`
  - Image files: `group-name_XX_image-description.jpg` (or ending in `.png`)
  - Make sure your file names only contain alphanumeric characters, "`-`" or "`_`" and make sure to avoid spaces in the file names

- Please keep in mind that if a blog post is broken in any way (i.e. does not work with the existing source code of the blog) or is of insufficient quality, the course instructor/TA will refuse to put the blog post on the blog website and this blog post will not be counted towards the evaluation, i.e. the student(s) will receive zero points for this blog post.

## 2.3 Presentations

### 2.3.1 General Points

- When conducting your presentations, please **always leave enough time for discussion, questions, and the instructor's feedback**. Based on previous experiences, around 20% of the presentation time should be set aside for this purpose. For example, if your presentation is scheduled to last for 10 minutes, you should prepare for around 8 minutes presentation and 2 minutes of discussion, questions, and feedback. Students **lose 5% of the overall presentation grade for each minute overtime** (on the group level). No points will be deducted for presentations that finish a bit early.

- The **durations of the group presentations** will be announced on the course website in due time. The reason why we cannot announce them now is that before the add/drop period, we do not know how many students will take the course and therefore how many groups we have in this course.

- All **presentations will be evaluated** based on the following criteria:

  - Confidence of the speaker. Is he comfortable, can easily connect with the audience, make eye contact, and put the audience at ease?
  - Quality of the information presented. Enough details to support the main point, but not too many unnecessary details that confuse or distract.
  - Level of clarity. Should easily be able to convey the main point, vocabulary should be easy to understand and all words should be spoken in a clear and fluent manner.
  - Level of organization. A good presentation should have structure and organization, including a proper introduction and conclusion.

- The group presentations should consist of going through slides and sometimes through short software demonstrations (showing the group's progress or results) on a student's laptop connected to the projector.

- **Presentations will be evaluated on an individual basis** according to the syllabus. Although it is encouraged that students present frequently, **it is not required that each group member presents during each presentation. But each student should present at least one time during the whole course.**

For each student, the overall presentation grade will be the average of all presentations done by this student. The decision about which students present during which presentation should be made within the group.

- During a presentation, please do not switch back and forth between students; if a student is done with his/her part, he/she should not come back later during the same presentation. (Of course, a student can come back to present in a subsequent presentation on another date.)
- The first page of the slides should contain a list of all presenting students including their full name and UID. The list should be **ordered in the sequence in which students present**.

### 2.3.2 Presentation Dates

- Each group presents in-class two times, the first time on **Saturday, January 10, 2026**, and the second time on **Saturday, January 17, 2026**. Depending on the class size, additional presentation dates may potentially be added to accommodate the number of groups. In case additional presentation dates are added, an announcement will be made. In any case (no matter how many presentations dates there are), each group presents two times only.
- In the **first presentation**, you start by laying out **your plan** (i.e. the roadmap and schedule) of the group project for the rest of the course. See also Section 5.1 starting on page 37.
  - What are the goals you would like to accomplish?
  - Why are they important?
  - What are the sub-tasks you need to do to accomplish your goals and what is the timeline for each sub-task? (You can show a Gantt chart for example.)
  - Who is responsible for which task?
  - Optional: What have you done so far to accomplish the first sub-tasks, are there any partial results you can show?
  - You do not need to show any code or results in the first presentation. If you are looking for inspiration about potential topics of your group project, take a look at Chapters 3 and 4 (pages 23 and 31, respectively), as well as some of the blog posts of previous courses taught (see Section 2.2 starting on page 13).
- Keep in mind that the plan is not set in stone. Think of your project as an iterative process. Most likely it will be necessary to go through several rounds of updates and changes. But it is still important to have a plan, even if it needs to be modified during the course of your project.
- The **second presentation** is the final presentation where you tie everything together and summarize what you have accomplished throughout the course in the group project.

- The **sequence of the groups presenting will be determined randomly** on each presentation day. The course instructor will run the following code in class (group names will be updated):

  ```
  from random import sample
  groups = ['Group A', 'Group B', 'Group C']
  sample(groups, k=len(groups))
  ```

## 2.4 Group Project General Points

- The **group project is evaluated** according to the following criteria:

  1. Quality of economic/financial idea
  2. Execution of economic/financial idea
  3. Quality of programming code

  Please keep in mind that the blog post(s) are evaluated separately as described in Section 2.2 starting on page 13. (As mentioned before, the blog post contents are **not** just a repetition of the slide contents and project report.) Moreover, while the presentation slides are part of the group project, the presentation skills of each student are also evaluated separately as described in Section 2.3 starting on page 16.

- You can freely choose a topic as long as it is somewhere at the **intersection of NLP and finance/fintech**. A central element of the project should be the analysis of textual data. Often you may want to include structured data as well, e.g. company fundamentals, macroeconomic data, etc., but the main focus of the project should be on textual data. In general, students are **encouraged to work on a project that has "real-world" NLP applications in the finance/fintech industry**. It is not required to partially replicate an academic paper (although a partial paper replication would also be allowed as a group project).

- Chapters 3 and 4 starting on pages 23 and 31, respectively, can give you a good starting point for generating your own group project ideas. In Chapter 3 there is a list with many potential **data sources**. Furthermore, there are a lot of **application examples of text analytics and NLP in finance** in Chapter 4.

- For the data sources and project ideas, keep in mind that these are only suggestions. If you have a great idea or a new dataset/source you would like to analyze, feel free to pursue it further. However, I strongly recommend consulting with me first to find out whether this project is feasible and suitable, as there are often "hidden" challenges that are difficult to identify in advance if you have not worked on text analytics and NLP before.

- Students should use a database that is freely available or a database subscribed to by the University. **Students should not have to pay for any database or**

**model/API used in the group project.** For a list of potential data sources please see Chapter 3 starting on page 23.

- For the group project, form groups as soon as possible, nominate a group leader, and **email a spreadsheet to the TA/tutor (at the latest on the day after the add/drop period)** containing the following information:

  - Group name (please choose a professional and sensible name)
  - Student ID numbers
  - Last names
  - First names
  - English first names (if any)
  - GitHub user names (if any)

  Next to the group leader's name put a label so that it's easy to identify him/her from the spreadsheet. In case several groups choose the same group name, it is first-come, first-served (in other words, the group who handed in later has to change their group name).

- The **accepted group size is 4-5 students per group**. You may form your groups already at the beginning of the course, but in general you might want to wait a bit with the group formation until students have committed to taking this course (at the latest until the end of the add/drop period). In terms of group size, you should try to strike a balance depending on the strengths and weaknesses of your group members. If you have too few students, each student will have more work to do, and no "bonus" in terms of grading will be given if you choose this arrangement. And if you have too many students, it might be difficult to coordinate the group work. Furthermore, you might not have enough time in your group presentations for everyone to present, which might have a negative impact on the evaluation of your presentation in case you have to rush and/or don't have enough time to get your point across or answer questions.

- To manage the workload efficiently, I suggest to split up your team by **assigning specific responsibilities from the start**. Whether and how you split up the workload is up to you, but you might want to orient yourself at the list from Section 5.1 starting on page 37.

- Please choose a project based on **textual analysis of the English language only**. The reason is simply that most software libraries are originally designed with English in mind. Furthermore, to ensure fairness to all students, it might sometimes be difficult to evaluate projects based on text other than English.

- When choosing a project, please keep in mind that it is often not easy to predict market prices in the future. So while it is tempting to work on a project that potentially could make a lot of money, you should have a **backup plan** if you fail to find predictive power for prices. For example, you could look at predicting volatility. Or you could focus on other variables that are of financial interest, e.g. predicting accounting fraud or others.

- In general, it is a good idea to **start simple**. The "simple" ideas in NLP are often already difficult enough to implement. Once you have finished the "simple" analysis, you still have ample opportunities to make it more complex.
- If some of your attempts do not work out (e.g. you cannot find predictive power for the question you're trying to analyze such as predicting whether the FED is going to raise rates), you should still describe what you did, what problems you encountered, and your attempts to solve these problem, even if those attempts were at the end not successful. **What matters mostly is whether you followed proper process, not so much whether your hypothesis finds support in the data in the end.**
- All files created with an office suite such as LibreOffice (Linux), iWork (Apple), or Office (Windows), with the possible exception of spreadsheets, should be converted and **submitted in PDF format** to ensure consistent behavior across different platforms. Every document that should be submitted in PDF format but is submitted in another format results in a deduction of 1% of the overall course grade for all students in the group.
- Source code files, Jupyter notebooks, and data files should stay in their original format and do not have to be converted to PDF.
- If you use GitHub, the programming code and data should be in repo(s) of the group leader. The group leader should give write access to the repo(s) to all group members who contribute source code or collect data. See also Section 2.1 starting on page 13.
- You should also hand in a project report summarizing what you have done in your group project. The cover page should contain all the names and UIDs of your group members as well as the name of your group.
- The project report should not only describe technical aspects of your work, but also provide information about the economic or financial question you are addressing and explain why your topic is important.
- Furthermore, there should be a separate "who did what" document that describes in half a page the way you split the work. Please **indicate the percentage of work contributed by each team member at the end of the project. If a free-rider is identified according to the percentage listed on the "who did what" document, the free-rider who contribute sufficiently less than their group mates will receive proportionately fewer points in the evaluation of the group project.** The group leader should communicate clearly the expected work allocation for each group member from the outset. Moreover, from the outset, it should be clearly communicated within the group what happens in terms of the

percentages in the "who did what" document in case a group member does not follow the work allocation. These things should be decided and communicated within the group from the outset, in order to minimize problems in the group throughout the course.

For example, if a group member was not a free-rider and did his work according to the work allocation, then his/her percentage would be 100%, meaning that the group member contributed 100% of what was agreed before when the work allocation was determined by the group. This is the baseline case, and hopefully all students in the group would adhere to this scenario.

A free-rider would have a percentage of less than 100%. On the other hand, if a group member has done more work to compensate for the lack of work from a free-rider, then the group member who has done more work would have a percentage of more than 100%.

If no percentage is given, all group members are assumed to have contributed equally (independent of the written description of the work done by each team member).

- In **summary**, the group project handed in should consist of the following files (combined in .zip compression format), **handed in at the latest at 11:59pm on the seventh day after the last class of this course** in one file called `group-name.zip` (replace `group-name` with the actual name of your group) and **submitted to the Moodle course page**:

  1. Project report (PDF format, max. five pages excluding cover page; the page limit includes the appendix, if any; a references section, if any, can be on an additional sixth page)
  2. Slides of all presentations held (PDF format)
  3. "Who did what" document (PDF format, half a page)
  4. Code files (`.py` format)
  5. Optional: Supplementary files, e.g. Jupyter notebooks
  6. Data files necessary to run the code; if the size of the data is larger than one megabyte (MB), you may provide a link to a file sharing service where the data is stored online, e.g. Git Large File Storage, Dropbox, Google Drive, etc.; if a file sharing service is used, it should be used for data files only, and all other files should be submitted (combined in .zip compression format) as as mentioned above

- The group projects are evaluated on the group level (unless some free-riders are identified, in which case adjustments will be made accordingly in the evaluation of the group report as mentioned above). So unless there is a free-rider detected, all students receive the same points for the group project (with the exception of the in-class group project presentations, which are evaluated on the individual level according to the syllabus).

# Chapter 3

# Data Sources

This chapter contains a few data sources for obtaining textual and financial data. The list is not all-inclusive. Its main purpose is to inspire you about what is possible. Keep in mind that I cannot guarantee for each data source listed in this chapter whether it allows bulk downloading and/or web scraping. When considering whether or not to use any data source listed here, **you should always stay within legal limits regarding download policies.**

## 3.1 Hugging Face Datasets

The Hugging Face Datasets library is a versatile tool designed to simplify the process of accessing and managing a wide array of datasets for Natural Language Processing (NLP) tasks. It offers seamless integration with popular machine learning frameworks such as PyTorch and TensorFlow, enabling efficient data handling and preprocessing. In the context of finance and fintech, this library provides access to numerous financial datasets, facilitating tasks like sentiment analysis, risk assessment, and market prediction. By streamlining data acquisition and preparation, the Datasets library empowers researchers and practitioners to focus on developing innovative NLP solutions tailored to the needs of the finance/fintech industry. The data can be found on the following links:

- Datasets library on GitHub
- Hugging Face Datasets Hub

The code below does a quick demonstration of how the Datasets library works. It should mostly be self-explanatory, but if something is not clear you can find further information about Python programmin in Chapter 6 starting on page 41. The code below can be found in the file `code-datasets-ag_news.py` on the course website.

```
# The Hugging Face ‘datasets‘ library provides access to a wide range
# of real-world datasets. Here's a quick example of how to explore and
# use a dataset. The AG News dataset, also known as "AG's News
# Corpus," is a widely used benchmark dataset for text classification
# tasks. It is constructed from news articles and focuses on four main
# categories: World, Sports, Business, and Science/Technology.

# Import the ‘load_dataset‘ function that loads a dataset from the
# Hugging Face Hub, or a local dataset. For details type
# ‘help(load_dataset)‘ after the import.
from datasets import load_dataset

dataset = load_dataset('ag_news')
print(dataset)

# Access the first training example. You can see it has a text and an
# integer label representing the news category.
print(dataset['train'][0])

# Show the category label and text content of the first seven news
# articles.
label_names = dataset['train'].features['label'].names
for example in dataset['train'].select(range(0, 7)):
    label = label_names[example['label']]
    text = example['text']
    print(f'Category:␣{label}\nText:␣{text}\n')
```

## 3.2 Example Texts

Sometimes you may need some example text data to run your code on. This could be the case, for example, if your production text data is still being assembled and not yet fully available, e.g. by web scraping, see Appendix F starting on page 239. To this end, you can use some of the text datasets provided as examples by the NLTK package. Assuming you have installed NLTK before (see Appendix B starting on page 223), you can download the “book” data as follows. This step only has to be done once.

```
import nltk
nltk.download("book")
```

Once the data has been downloaded, you can use it in your programs:

```
from nltk.book import *
texts()            # Show all texts available.
print(text1)       # Moby Dick by Herman Melville.
print(text1[:88])  # Beginning of book.
print(text2)       # Sense and Sensibility by Jane Austen.
```

## 3.3 Text Data

Appendix E starting on page 231 provides an overview of Python packages that can be used to download and web scrape from the various data sources discussed in this section.

## Finance & Corporate / Regulatory & IR

- SEC EDGAR: 10-K, 10-Q, Form ADV, etc. Public filings from the U.S. SEC provide rich, formal financial and risk disclosures, management discussion, and footnotes. Many filings include machine-readable XBRL and exhibit attachments (PDF/HTML).
- Company websites, including (but not limited to) press releases and investor relations. Typical content includes press releases, investor presentations, annual reports, and FAQs; formats are usually HTML or PDF. Useful for event studies and company-specific terminology.
- PR Newswire or other newswires. Syndicated, time-stamped press releases covering product launches, earnings dates, M&A, and executive changes. Offers structured headers (company, ticker) and consistent formatting.
- Factiva by Dow Jones & Company has newspaper articles and newswire articles (e.g. press releases of companies) as well as earnings calls transcripts
- Capital IQ by S&P Global has various business-related transcripts such as earnings calls
- Seeking Alpha hosts a wealth of information, for example earnings calls and articles about the stock market.
- Bloomberg has company filings, analyst reports as well as legal and business documents. Some of them might be downloadable via the Bloomberg Python API.
- FOMC statements/minutes of the Federal Reserve. Monetary policy statements, minutes, and meeting materials in formal policy/economic language. Consistent templates across decades make them suitable for diachronic analysis.

## News & News APIs

- News, e.g. mine websites of major news providers or take a look at Google/Yahoo News, Google/Yahoo Finance, etc. or directly at news sources like NYT, AP, CNN, as well as international sources. See also the Python packages we have mentioned under web scraping in Appendix E starting on page 231.
- You can obtain news from NewsData.io, NewsAPI.ai or NewsCatcher, which are some of the simplest ways to obtain news data if you don't mind the download restrictions on their free tiers. These typically return JSON with headlines, bodies, sources, and timestamps; rate limits apply.
- There is also a list of other news media APIs on Wikipedia. This catalog helps compare coverage, access models, and historical availability across providers.

## General Reference, Archives & Corpora

- Wikipedia: Download the dump of all Wikipedia articles from here (you want the file `enwiki-latest-pages-articles.xml.bz2`, or `enwiki-YYYYMMDD-pages-articles.xml.bz2` for date-specific dumps). This file is about 10 GB in size and contains (a compressed version of) all articles from the English Wikipedia. Using gensim on the command line, you can convert the articles to plain text and store the results as sparse tf-idf vectors:

  ```
  $ python -m gensim.scripts.make_wiki
  ```

  It might take up to ten hours. You might want to compress the output with bzip2 as Gensim can work with compressed files directly to save disk space.

- Project Gutenberg. Public-domain books (novels, essays, poetry) in plain text and EPUB; excellent for long-form, clean literary English. Good for language modeling, style transfer, and readability tasks.

- JSTOR. Scholarly articles and book chapters with stable metadata and citations. Text is formal/academic; access is typically paywalled, though some content and abstracts are open.

- Internet archive. A digital library hosting books, magazines, and datasets in varied formats (PDF, text, EPUB). Useful for historical corpora and out-of-print materials.

- You can use the WayBack Machine to download historical news, e.g. from the archives of Yahoo News, Google News, or obtain data from Reddit, which is a social news aggregation, web content rating, and discussion website. However, keep in mind that the WayBack Machine does not always take snapshots on a regular basis, so the coverage might have gaps.

## Social, Forums, Blogs & Multimedia

- Twitter and/or StockTwits, see also Section F.4 starting on page 248. There are also some websites that provide data for a subset of Twitter, e.g. the Trump Twitter Archive.

- Sentiment140 is a dataset with 1.6 million tweets. Labels are derived from emoticons, enabling large-scale positive/negative sentiment training on short, informal text.

- Reddit. Thematic subreddits with threaded discussions and up/down-votes; language ranges from informal to technical. Public API (with auth) returns JSON suitable for comment/post-level analysis.

- Online forums and discussion groups. Niche communities (e.g., tech, health, hobbies) often running phpBB/vBulletin/Discourse; long conversational threads. Scraping typically requires site-specific parsers and robots.txt awareness.

- Blogs and blog aggregators/platforms like Blogspot and Blogger. Personal essays, tutorials, and commentary with RSS feeds for crawling. Writing quality and editing standards vary widely, which is useful for robustness testing.
- Medium.com. Curated long-form posts with tags, publications, and engagement signals (claps, comments). Some content is paywalled; editorial tone skews explanatory/opinion.
- Tumblr. Microblogging with short posts, reblogs, and rich tagging; content mixes text with media. Useful for studying slang, fandom, and meme diffusion.
- Facebook and Google+. Facebook Pages/Groups provide posts and comments (access depends on permissions and API policies). Google+ is discontinued for consumers; only historical archives/dumps are available, making it a legacy source.
- Instagram. Short captions and comments accompanying images/videos; hashtags and mentions enable topic/time-series analysis. API access is restricted; public web data is primarily from public accounts.
- YouTube, e.g. you can analyze the comments. In addition, video transcripts (auto-generated or uploaded) and metadata (titles, descriptions) provide long-form, topic-rich text for NLP tasks.
- There are also services that provide firehose access to social media, e.g. Six Apart, Spinn3r, Datasift, and GNIP (now part of Twitter). These offer enriched or historical streams with filters/metadata; access is commercial and governed by strict licensing/ToS.
- In principle you could also look at VK or Sina Weibo, but we want to focus on the English language in this course to keep things simple. These are major Russian/Chinese platforms with their own APIs and linguistic ecosystems; English content is limited.

## Employment, Reviews & Ratings

- LinkedIn. Public profiles, job descriptions, and long-form posts provide professional/industry vocabulary; access is limited by API and ToS. Company pages supply mission statements and hiring signals.
- Glassdoor company reviews. Employee-written reviews, ratings, and salary insights with free-text pros/cons. Valuable for sentiment and aspect extraction about workplace culture.
- Product reviews (the exact location depends on what kind of product you're looking at, e.g. Amazon, Walmart, Ticketmaster, and TripAdvisor). Review text paired with structured metadata (stars, categories, timestamps) is ideal for supervised sentiment and recommendation tasks.

- Rotten Tomatoes. Critic snippets and audience reviews with aggregated scores (Tomatometer/Audience). Good for comparing expert vs. crowd sentiment and for domain adaptation across review styles.

### Datasets & Competitions

- Kaggle contains text datasets used in some competitions. Many datasets include prepared train/test splits, detailed metadata, and community notebooks, which simplify benchmarking and reproducibility.

## 3.4 Fetching News With the NewsData.io API

This short script demonstrates how to connect to the NewsData.io REST API to pull top breaking news from the past 48 hours and then query business/finance articles for a specific ticker (e.g. AAPL). NewsData.io aggregates articles from many outlets and returns normalized metadata, making it convenient for rapid prototyping of retrieval, sentiment, and event-study pipelines without building custom scrapers.

After reading your API key from a local file, it calls the `/latest` endpoint, pretty-prints the complete JSON payload (including keys such as `results` and `nextPage`), and then issues a filtered request with query, category, and language parameters. Finally, it iterates over `results` to print selected fields (title, publication time, source, creator, description, link); note that on the free tier some fields may be empty or placeholders (e.g., `content` often requires a paid plan).

The code below can be found in the file `code-API-NewsDataIO.py` on the course website.

```
# This script connects to NewsData.io through their API and fetches
# top breaking news from the past 48 hours. You can obtain a free API
# key by creating an account on https://newsdata.io, then enter the
# API key into the file named `code-API-key-newsdataio.txt`.

import requests
import json                      # For pretty printing.

# Read API key from file or paste as text string, for example:
# API_KEY = 'paste_here'
with open('../code-API-key-newsdataio.txt', 'r') as f:
    API_KEY = f.read().strip()

# Retrieve the latest news.
response = \
    requests.get(
        f'https://newsdata.io/api/1/latest?apikey={API_KEY}')
# sort_keys=True` `sorts the JSON keys alphabetically (e.g.,
# "article_id" before "category", and top-level like "nextPage" before
# "results").
```

```
print(
    json.dumps(
        response.json(),
        indent=4,
        sort_keys=True))

# Retrieve the latest news about Apple (stock ticker: AAPL).
stock = 'AAPL'
response = \
    requests.get(
        ('https://newsdata.io/api/1/latest?'
         f'apikey={API_KEY}'
         f'&q={stock}'
         '&category=business'
         '&language=en'))
# Print everything. But on the free tier, many fields are empty or
# filled with placeholders.
print(
    json.dumps(
        response.json(),
        indent=4,
        sort_keys=True))
# Print only a few selected fields.
for article in response.json()['results']:
    print(f"Title:␣{article['title']}")
    print(f"Published:␣{article['pubDate']}")
    print(f"Source:␣{article['source_name']}")
    print(f"Creator:␣{article['creator']}")
    print(f"Description:␣{article['description']}")
    print(f"Content:␣{article['content']}") # Only in paid plan.
    print(f"Link:␣{article['link']}")
    print('-' * 80)  # Separator for readability.
```

## 3.5 Financial Data

This section provides a brief overview of some of the most popular financial databases. These data providers can be useful if you want to figure out how your textual data relates to various financial statistics and events, e.g. asset prices, corporate events, or fundamental data.

- **Bloomberg** has almost all financial data you could imagine, but it is also one of the most expensive data providers out there.

- If Bloomberg is too expensive, you can consider Thomson Reuters **Eikon** as an alternative.

- **Capital IQ** and **FactSet** are popular for fundamental data about companies (among other things). Capital IQ and FactSet are popular in investment banking, e.g. Mergers & Acquisitions, so if you need to analyze a company in detail, it is very useful to look at one of these databases.
- **Morningstar**, **Compustat** (via WRDS), **Reuters Fundamentals**, and **Worldscope** also have fundamental data.
- While both Capital IQ and Compustat have fundamental data and both are provided by S&P, they differ in the way they standardize the accounting variables and also in the time period covered (Capital IQ goes back to 1989 while Compustat goes back to 1950).
- For industry-specific metrics (e.g. ROAA or Tier-1 capital ratios for financial institutions) or news, you can use S&P's **Market Intelligence Platform**, formerly known as SNL Financial.
- **I/B/E/S** has earnings estimates.
- The **Wind** database is a leading provider for fixed income data for China.
- **AlphaVantage** is a free data provider for stock prices and digital/crypto-currencies (see also Section G.8 starting on page 271).
- **Coinbase** can be used for digital/crypto-currencies.
- **Yahoo Finance** and **Google Finance** provide stock prices, among other data.

# Chapter 4

# Applications of Text Analytics and NLP in Finance

This chapter contains examples of text analytics and NLP applications in finance and beyond. Its purpose is not to go into great detail for every application, but instead to provide an overview of what is possible. Furthermore, after going through this chapter, one should have a more intuitive understanding of the potential capabilities of text text analytics and NLP.

## 4.1 General NLP Applications

Many people are unaware that they have already used text analytics and NLP in one form or the other before. The following list provides an overview of general applications, while we discuss finance-specific examples further below.

- Spell checking in Microsoft Word
- Email spam filters
- Call centers transcribe conversations into text and analyze it to find out more about common complaints and problems
- Email messages with complaints to a municipal authority are automatically routed to the appropriate department (or returned if they contain inappropriate or obscene messages)
- Search engines such as Google and Bing
- Google Translate
- Amazon's Alexa, Google's Now, Apple's Siri, or Microsoft's Cortana respond to vocal prompts (which are transcribed to text internally) and do everything from finding a coffee shop to getting directions or turning on the lights at home
- Question answering, e.g. IBM Watson for healthcare, weather, and insurance.

## 4.2 Applications in Finance

The list below includes many applications to make it clear there are relevant topics in the real-world that can help you make money and/or make the world a better place in general:

- Fed watching, i.e. analyzing reports and minutes coming out of the Federal Reserve. For example, you can calculate the text-based probability that they are going to raise or lower rates, which is a trillion dollar question.
- Understanding and responding to consumer and investor sentiment in financial newspapers and social media. This is not only of interest to corporations for the public/investor relations departments, but can also provide important feedback loops for top managers, e.g. the C-suite (CEO, CFO, COO, and CIO).
- Improve risk-adjusted performance in asset/investment management by analyzing financial newspapers, social media, company filings, etc. The idea is to tap the "wisdom of the crowd" to extract additional information that is otherwise not readily available. This is important because it benefits investors (they get richer) and ultimately can give you as the portfolio manager a higher bonus and the hedge/mutual fund can generate higher income from fees.
- Auditing can be automated to some degree, as is currently being done by KPMG using data analytics. In particular, textual auditing content can also be analyzed using NLP and text analytics.
- Using the same data sources, one can estimate the profitability or the default risk of companies.
- Using online product reviews to gauge customer satisfaction as well as product quality and whether people are receiving a good service. Happy customers might translate into higher sales and generally a better company, which gets reflected in higher stock prices.
- Banks can use chatbots to automate customer service functionality or provide robo-advisory services (e.g. you're investing according to a quantitative model provided by your robo-advisor, and if you have questions you can simply go have an online chat with your virtual advisor).
- Monitoring of company events such as earnings announcements, share buybacks, or mergers and acquisitions. Potential data sources are company reports, analyst reports, newspaper and newswire articles, and social media. For example, if you are running a hedge fund for merger arbitrage, you can try to find out the ex-ante probability of a merger being announced or post-announcement you can find out the probability that the merger completes. Having sharper insights into these events can directly translate into higher profits.

- Due diligence of business/company and legal documents. If textual analysis produces a red flag, a lot of money can be saved by canceling a merger deal or an IPO, for example.
- Detecting insider trading. Again this may be done by analyzing company filings, business/legal documents, newspapers, social media, etc. Of course this kind of information can only be based on circumstantial evidence and is no substitute for a deep investigation using e.g. forensic accounting, but it is much easier and faster to do and allows you to monitor a much larger set of companies, and importantly, you can do this using public information only (no private information required).
- Monitor the news for key hire alerts, e.g. when a company (maybe your competitor) hires talented finance, development, marketing, or sales executives, or even people from the C-suite.
- Fraud detection, e.g. transcribing earnings calls and analyzing the choice of words used by C-level executives (CEO, CFO, COO). In fact, one could even go further and analyze their voices directly, but this would go beyond text analytics. Similarly, one could analyze regulatory filings for language that implies fraud, e.g. increased usage of negative-emotion words or reduced usage of first-person pronouns.
- Analyzing warranty or insurance claims.
- Analyze competitors by crawling their website and learn about what topics, terms, and features are most important.

## 4.3 Ideas for Further Work

The examples below go into greater detail and discuss potential applications in finance including the relevant databases. Please respect all legal limits regarding downloading and usage of the databases mentioned below.

- Realvision trade ideas. 1) Low likes, high like/dislike ratio, overwhelmingly negative comments: Go with it in size. 2) High like/dislike ratio, massive enthusiasm in comments: Sell as much as you can.
- How do discussions on social media differ during expansions and recessions? Can we use these differences to predict recessions?
- Do firms that have mostly good Glassdoor reviews perform better than firms with bad reviews?
- Can we learn something from the tweets of “famous” finance Twitter (or StockTwits) users about the future behavior of stock prices or stock market indices? For example, can we say something about future volatility or future returns?
- Does news coverage of bitcoin lead or lag the bitcoin price?

- Which articles on Medium are about finance-related topics? If they are, in what groups can they be clustered?
- Are stocks with more news coverage less risky?
- Which SEC EDGAR filings are most suitable for predicting fraud or scandals, e.g. in accounting (you could look at several famous scandals such as Enron to begin with and use them as a training sample)?
- Scan PR Newswire for merger announcements and extract the names of the target and the acquirer. How fast does the stock market react to this news?
- Do firms with more disclosure have less surprises in their earnings announcements?
- Calculate news sentiment based on streaming analysis of online news. Are news particularly euphoric/upbeat before stock market drawdowns?
- When analyzing the textual content of company filings from SEC EDGAR (e.g. of financial institutions), can we detect signs of financial market instability?
- Do companies that are frequently mentioned together with fintech terms such as "blockchain" do better on the stock market?
- Can we learn anything from FOMC statements/minutes about the probability of a future change in interest rates?
- Can textual content from the insider trading Forms 3, 4, 5 on SEC EDGAR predict strong selloffs in stock prices? See also here.
- Do firms that mention terms related to hedging and/or derivatives usage have riskier cash flows and/or riskier stock prices?
- Scan newspaper articles (e.g. from Google/Yahoo News or other sources) and/or newswire articles (e.g. from PR Newswire or other sources) and find out which companies have the most mentions of "hot" topics such as "blockchain," "fintech," or others. Do companies that are mentioned more frequently with these hot topics have higher stock returns than comparable companies? You should adjust for company size and/or the number of articles released for each company to ensure that your results are not driven by size or by media coverage.
- Extract product descriptions from 10-Ks and find out how similar companies are to each other in this space. It is possible to derive industry groups based on this similarity, see e.g. Hoberg and Phillips (2018 JPE). There are many questions that can be addressed with this data, e.g. do stocks of companies in similar text-based industries co-move together?
- Construct sentiment from conference call transcripts. Does positive sentiment predict higher stock prices?

- If there are more words related to "risk" in analyst reports (or other financial text documents), are stock prices or operating performance riskier/more volatile?

- Look at news articles and check what clusters they are falling into, e.g. using $k$-means clustering. Then find out what the market does on the following day depending on which cluster an article falls into. If you find a pattern, you can build a trading strategy based on that. For example, you find there are three clusters, and whenever an article falls into cluster two, the market goes up the next day on average. From now on, every time an article falls into that cluster, you go long the market.

- Does more disagreement on stock message boards (or Twitter or FinTwit) imply the stock is more volatile? Is it possible to trade profitably on this higher volatility, e.g. using option trading strategies such as straddles or strangles?

# Chapter 5

# Data Workflow

The following sections illustrate typical data workflows in finance. Of course, depending on the actual application there might be some differences. Furthermore, it is important to keep in mind that these steps are pretty general and might or might not include a textual analytics component.

## 5.1 General Data Workflow

An important point is that this workflow should not be thought of as going linearly through it. Instead, it requires a lot of iteration. For example, when you validate your data, you might discover some flaws and need to go back to data cleaning. Or when you analyze your data, you find out that you need a different variable, in which case you go back to data transformation. In any case you can tie much of the workflow together programmatically with data workflow management tools such as Apache Airflow.

1. Data collection: This is obviously the first step, but unless you just download raw files, you should already at this stage start thinking about the next step. In any case, it is usually a good idea to separate the raw data from any data that is derived from the raw data. **Put the raw data into a separate directory and write-protect it** to avoid accidental alterations.

2. Data structure: This step is not so much about programming, but more about thinking and planning ahead on a conceptual level.

    - It is really important to have thought about how you want to structure your data. If you are in doubt, use **3NF / tidy data** (see also Hadley Wickham's tidy data article in the JSS).
    - Another big question is on database design, which does not always have to be fully laid out in all formality (depending on the size of your project), but you should have thought about this topic in any case.
    - Finally, you should also think about naming conventions for your data files and variable names. Often you will have a lot of files and variables at the end of

the project, and it makes a lot of sense to have consistent naming conventions for file names (actually for both data and code files) and variable names.

3. Data storage: Often it is simpler to just store your data in files without using a full-blown database. Examples of these files are binary files (e.g. `.rds` files in R or `.pkl` files in Python, see also Section G.3) as well as text files (plain text, XML, JSON, etc.). In some cases a database might be helpful or necessary, especially for large projects. Depending on your needs, I recommend looking at SQLite, PostgreSQL, MongoDB, or Elasticsearch, as these are some of the most popular databases in the main database categories out there.

4. Data cleaning, reshaping, preprocessing: This step is basically the programmatic implementation of the previous step (which was more conceptual). This step cannot be overestimated in terms of its importance because you know what happens to the output of your analysis when there is "garbage in."

5. Data validation: Does the data indeed have the properties you think it has? **You will be surprised how often in the real world the data does NOT look the way you think it looks!** Data validation often is not a single step where you write a separate program that performs a set of tests on your data. Instead, data validation should take place throughout your code. **Whenever you make an assumption about your data, you should state this assumption explicitly at the appropriate point in the code. If the data does not satisfy your assumption, your program should throw an error.** In this respect the `assert` statement in Python or the `stopifnot` function in R are valuable and practical tools. For example, if you assume that for each stock ticker symbol there is only one observation per date, you should explicitly state this assumption using `assert` or `stopifnot`.

6. Merging: This step consists of merging (or "joining") different datasets into a new dataset. For example, if you have data on stock prices and another dataset on accounting variables, you might want to bring both of them together and merge them into a single new dataset. Another example is if you have calculated some scores based on textual analysis and want to merge these scores to your stock market data. We discuss merging in detail in Appendix G starting on page 255.

7. Data transformation: This step takes the cleaned data and transforms it such that you can use it straight away in the following step about data analysis. For example, you might want to center some variables or apply other transformations. Depending on the task at hand, data transformation might take place before and/or after merging.

8. Data analysis

    - Summary statistics, statistical methods, machine learning, etc.

- Data visualization (more details in Appendix H starting on page 277): This encompasses plotting and online apps. This step is actually a very important part and you should think early on how you want to present and visualize your results, be it a Jupyter Notebook or a dashboard made with Dash or Pyxley in Python, or ggplot2, Shiny and related packages in R.
- Summarize the information we can learn and the actions we should undertake based on our data analysis.

9. Community engagement and public relations:
   - Documentation: Your work should be extremely well-documented, both in terms of commenting your source code (explain on a high level what it does and, more importantly, why it does so) as well as writing external documentation. This is not only important for other people who want to use your software or contribute to it, but also for future self when you get back to working on your project after taking a break.
   - Set up a blog and write about your work. See also Section 2.2 starting on page 13 for more details.
   - Give presentations about your work at user groups, fora, and conferences.

10. Quality control:
    - Code review should take place on regular intervals where people review other people's commits or the source code.
    - Pair programming means that two people sit in front of the computer, one is the "driver" who writes code, while the other is the "observer" who reviews the code as it is being typed in by the driver. The two people switch their driver/observer roles frequently. Pair programming can be extremely useful because it solves a couple of problems. First, it avoids that people get too tired. After all, our attention span is typically pretty short. Second, there is a lot of learning involved as it is a collaborative exercise. Third, it yields higher code quality.
    - Code refactoring: This refers to rewriting the code in a better way without changing its functionality. This should be taking place basically all the time to constantly improve the code quality.
    - TDD: Test-driven development (TDD) means that you add tests to your code and make sure they pass. Although you can never be 100% sure there are no bugs, it really increases your confidence if you have a lot of tests that pass. TDD makes mainly sense for larger projects, but even for smaller projects you can often add one-liners where some basic assumptions about the behavior of your code or your data are checked.

## 5.2 Investment Management Data Workflow

This workflow is more specifically tailored towards investment management. You can view it as a different flavor of the workflow from Section 5.1, looking at things from a slightly different and more specialized angle.

1. Identify and acquire the data
    - Formats: CSV, JSON, API, streaming, Text, HTML, etc.
    - People: Data managing team, legal/compliance
2. Store, structure, and preprocess the data
    - Database/computing infrastructure: SQL, NoSQL (MongoDB, Redis, Cassandra, etc.), HDFS, the "cloud," Spark, etc.
    - People: Software, system, data engineers
3. Analyze data via machine learning, design signals, backtest strategies
    - Analytics software: Excel, R, Python, Dask, Spark, Tableau, etc.
    - People: Data scientist, quantitative researchers, portfolio managers
4. Trade ideas, trading signals, risk analyses
    - Output: Report, alert, signal, dashboard, etc.
    - People: Traders, portfolio managers, execution system

# Chapter 6

# Introduction to Python Programming

Where does the name “Python” come from? Originally it nothing to do with the animal of the same name. Instead, it is named after the BBC show “Monty Python’s Flying Circus,” which is a British sketch comedy series from the late 1960s and early 1970s. The Python language was created by Guido van Rossum, a Dutch programmer who now lives and works in the U.S. Until 2018 he was the “benevolent dictator for life” (BDFL) of the Python programming language, a title given to him by the Python community.

Python is one of the most widely-used programming languages for text analytics and NLP. Of course you can do NLP in any programming language, but not all of them offer a large ecosystem of NLP-related libraries. Furthermore, for languages other than Python, the community might not always be very strong for NLP-related tasks, e.g. you might not be able to find a lot of answers on StackOverflow.

In this chapter will gently go through the basics of Python programming. If you would learn more about **text processing** in Python, you can visit Chapter 12 starting on page 153.

## 6.1 Python Philosophy

At the time when Python came out in 1991, another language called Perl was relatively popular. One of Perls programming mottoes was “**there is more than one way to do it.**” This is fine, and Perl is still a great language, but one problem with this motto was that it sometimes confused newbies and made the language maybe less accessible to some people. Python on the other hand had a very different philosophy. One important Python motto is, “**there should be one– and preferably only one –obvious way to do it**.” This really rang a bell with many people and Python has been and still is growing at an amazing speed, having overtaken Perl in popularity long ago.

You can read the Zen of Python directly here (you can also read it in Python by typing `import this`). There is a lot of truth and wisdom hidden in there that benefits most software developers, no matter which language they program in.

```
Beautiful is better than ugly.
Explicit is better than implicit.
```

```
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you’re Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it’s a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let’s do more of those!
```

## 6.2 Why Are We Using Python?

Because it’s a language that has a few key features that makes it very suitable to what we are doing in this course.

1. Easy to learn: Python is a high-level programming language.

   - It is dynamically typed. This means that you do not have to do a lot of bookkeeping such as in lower-level languages such as C, C++, or Java. For example, you can define new variables on the fly and you don’t have to specify their type.
   - You also don’t have to worry about memory management because a garbage collector is included.
   - It’s interpreted. It means you don’t have to compile a bunch of files like in C or C++, so you don’t have to worry about linking files and loading them. Furthermore, you have a terminal where you try things out interactively. You type something and immediately you can see the result.

   So Python tries to do a lot for your behind the scenes. Of course, this comes at a price. Python is slower than low-level languages. So the tradeoff is that you can write programs faster than in, say, C, but your programs tend to run slower. Still, for many tasks the speed of Python is just fine, so it’s a great programming language to learn.

   Furthermore, Python is a simple and minimalistic language. Reading a Python program almost feels like reading English. It allows you to concentrate on the solution rather than the language.

2. Powerful: Python is a full-fledged programming language originally designed by Guido van Rossum. Due to it being a high-level programming language, you can get things coded up very quickly.

3. Community: Even a great programming language cannot really thrive unless it's backed by a great community. (Of course, great programming languages often foster a great community, but it is not necessarily always the case.) A great community means that people share their ideas and experiences in blog posts, contribute to discussions in online forums, and share and answer questions on sites such as StackOverflow.com. This means that if you're looking for something, instead of studying a potentially obscure manual, you simply google whatever question is on your mind and you will most likely find an answer very quickly. This can be a tremendous productivity boost for your programming.

4. Documentation: This is the minimum requirement for any great programming language, but I view documentation in a larger context. It should not only encompass the language itself, but also the whole ecosystem of software packages around that language. Specifically, it should be relatively easy to create professional-looking documentation if you're writing your own software so as to encourage best practices and sharing of information. Even a great software package is not very useful if it is not well documented. And we all know that documentation is not necessarily the favorite hobby of most programmers. So it is good if there are state-of-the-art tools available such as Sphinx that facilitate documentation and make it easier.

5. Libraries and ecosystem: Even if you have a well-designed programming language, if it doesn't come with "batteries included" (i.e. a set of useful add-on libraries), it might be difficult to get things done efficiently and quickly. In the worst case, you might have to re-implement what is already available in other programming languages. Consider for example Haskell. It is a beautiful language, but at the time of this writing it is behind Python or R in terms of its add-on libraries for data science. Even if you had the time to re-implement some of the Python or R packages in Haskell, it would likely be a painful, slow, and error-prone process. Take for example BLAS, which originated as a Fortran library in 1979. All it does is do common linear algebra operations such as vector addition or matrix multiplication. Sounds easy enough. But if you would implement it yourself, you would quickly run into many subtle problems. Among others, there are subtle numerical issues that a naive implementation almost certainly overlooks such as numerical stability when inverting a matrix or round-off errors when solving systems of linear equations. In many cases these issues might not show up, but when they do, they can lead to false conclusions and ultimately to wrong decisions, which, as we all know, can be very costly in finance.

   Especially in the areas of text analytics, NLP, data science, and machine is where Python shines. It is a wealth of add-on libraries that are state-of-the-art and have an active developer and user community. Besides Python being easy to use, this is the main reason why we are using Python in this course.

## 6.3 Software Development

When you develop your software, it is typically a good idea to follow these steps:

1. Analyze the problem you're dealing with.
2. Design a potential solution or approach to deal with your problem.
3. Implement it using software, i.e. start writing code. Begin with a simple version that gets the core job done.
4. Test and debug the previous step.
5. Deploy your software to make sure it works as expected.
6. Maintain and refine your software.
7. Add new features (start again with the implementation part and repeat the steps thereafter as often as necessary)

## 6.4 A First Tour of Python

You can start the Python shell by typing `python` or `ipython` in your operating system's command line, and you can quit by pressing `[ctrl+d]`. After starting Python, you can enter commands, for example you can print a string:

```
print('I love NLP.')
```

You can also save this code into a file, e.g. `testing.py` (use any IDE or text editor you like, see Sections C.1 and C.2). Then run it from the command line by typing:

```
python testing.py
```

When doing so, make sure that you're in the same directory as the `testing.py` file, otherwise change into it by using `cd mydirectory`.

Now let's go back into the Python shell. If you have text that should not be run ("interpreted") by Python, you can add it as a **comment** by using the # symbol.

```
print('I love NLP.')   # This is a comment.
# This is another comment (below we print another string):
print('I love text analytics.')
```

Why should you be using comments? Because they let you explain the **why**. Comments can help you explain what you're actually trying to accomplish in this part of the code, the assumptions you're making, and further details that might not be obvious from just reading your code. They are not only useful for your teammates to understand you, but also for you to understand yourself two weeks from now.

If you would like to learn more about a given function or statement (e.g. the `print()` function), you can simply type the following command to **display the help page** and you can type `q` to exit the help.

```
help(print)           # Exit by typing ‘q‘.
```

A **string** is a sequence of characters. As you can imagine, in this book, strings are particularly important, since they are the fundamental building blocks for text analytics and NLP. In Python, it mostly does not matter whether you use single or double quotes:

```
'Hello world.'
"Hello world."    # Both give the same result.
```

Sometimes you want triple quotes in case you have long strings spanning several lines, so-called **multi-line strings**. You can use ' or " to create triple quotes. In the following example we use ' for the triple quotes.

```
'''I love NLP.
NLP is great.
'''
```

Multi-line strings can be useful, e.g. if you want to document a **function** you are defining. The multi-line string in this case is called a **docstring** because it contains the function's documentation.

```
def f(x):       # Here we are defining the function ‘f‘.
    '''Simple function that just squares
    its input and returns it.
    '''
    return x ** 2
```

After defining the function, you can display the docstring by typing

```
help(f)           # Exit by typing ‘q‘.
```

And you can call the function in the usual way in the Python shell:

```
>>> f(3)
9
```

## 6.5 Elementary Python

### 6.5.1 Basics

- Difference between a Python module and a Python package: A **module** is a Python source file, while a **package** is a directory of Python modules. For further details see Section 6.11 starting on page 80.

- Manage the working directory. The working directory is the location on your hard drive where Python reads or writes files by default.

```
import os           # Load the ‘os‘ module.
os.getcwd()         # Show current working directory.
os.chdir("mydir")   # Change to another working directory,
                    # e.g. the ‘mydir‘ subdirectory.
os.getcwd()         # Optional: Check whether the change was successful.
```

- If you have a command that goes beyond one line, you can break it up using a backslash (“\”). Make sure to add **no space after the backslash.** The following is a stylized example where we break up the assignment into two lines of code.

```
very_long_variable_name = \
    8
```

- Multiple assignment

```
x = y = z = 8                # x, y, z will all be 8.
x, y, z = 'hello', 8, 3.14   # Same as assigning separately.
```

- pprint is useful for **pretty-printing of data structures**. The following example is based on the official Python documentation:

```
>>> import pprint
>>> pp = pprint.PrettyPrinter(indent=4)
>>> x = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> x.insert(0, x[:])    # Add shallow copy of ‘x‘ at beginning.
>>> print(x)             # "Normal" printout of ‘x‘ for comparison.
[['spam', 'eggs', 'lumberjack', 'knights', 'ni'], 'spam', 'eggs',
'lumberjack', 'knights', 'ni']
>>> pp.pprint(x)         # Nice printout of ‘x‘ using ‘pprint‘.
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
```

### 6.5.2 Strings

As we have seen before in Section 6.4 starting on page 44, you can define strings using single quotes or double quotes. In most cases it does not make a difference which way you use.

```
>>> s = 'hello world'     # Single quotes.
>>> print(s)
hello world
```

```
>>> s = "hello world"     # Double quotes.
>>> print(s)
hello world
```

For multi-line strings we use triple quotes, either three times a single quote ' or three times a double quote '' as opening and closing triple quotes. In the following example we use three single quotes as triple quotes:

```
s = '''This is a multi-line string.
It can span multiple lines with ease.
Just use triple double quotes at the beginning and end.
'''
```

A multi-line string can contain newlines. A **newline** is a special character that represents the end of a line of text and the beginning of a new line of text. You can observe the newlines contained in the mutli-line string by printing the string:

```
>>> print(s)
This is a multi-line string.
It can span multiple lines with ease.
Just use triple double quotes at the beginning and end.
```

If you don't want a multi-line string but just want a long single-line string without newlines, you can enclose the strings in brackets *without* commas between the strings (otherwise you would be creating a tuple, see Section 6.7.4):

```
('Natural language processing (NLP) is a field of computer '
 'science concerned with the '
 'interactions between computers and human (natural) '
 'languages, and, in particular, concerned with programming '
 'computers to fruitfully process large natural language data.')
```

Alternatively you could also concatenate the strings together. You can use + and then tell the interpreter that there is more coming in the following line by adding a backslash ("\") at the end of the line and making sure there is no space after the backslash.

```
'Natural language processing (NLP) is a field of computer ' + \
'science concerned with the ' + \
'interactions between computers and human (natural) ' + \
'languages, and, in particular, concerned with programming ' + \
'computers to fruitfully process large natural language data.'
```

In general, **string concatenation** can be done in several ways. Usually **f-strings are preferred** due to their simplicity, but it's good to know the other ways as well.

```
w1 = 'hello'
w2 = 'world'
w1 + w2                  # helloworld
```

```
w1 + ' ' + w2           # hello world
f'{w1} {w2}'            # f-string.
'%s %s' % (w1, w2)      # %-formatting, similar to `str.format`.
'{} {}'.format(w1, w2)  # Format string syntax (`str.format`).
'{0} {1}'.format(w1, w2) # Same as before.
'{1} {0}'.format(w1, w2) # `w2` is inserted first, then `w1`.
```

Sometimes you need to be careful if you have different data types.

```
w1 = 'hello'
w2 = 8                  # int (not str).
w1 + w2                 # Does NOT work.
w1 + str(w2)            # Works.
f'{w1} {w2}'            # Works.
'%s %s' % (w1, w2)      # Works.
'{} {}'.format(w1, w2)  # Works.
```

Most of the time when you define a string, you can simply enter the text directly into the string, for example:

```
s = 'hello world'
```

However, this is not always the case. For example if you would like to have a newline in the string, you cannot do so directly in a regular string, and even in a multi-line string you can only do so indirectly.

```
# Cannot enter newline into a regular string (syntax error).
s = 'hello
world'
```

Let's investigate more closely how Python handles the newline in a multi-line string:

```
>>> s = '''hello
...  world
... '''
>>> s
'hello \n world\n'
```

We can see that the newline has something to do with `\n`. In fact, `\n` is an example of a so-called **escape sequence** which in this case represents the newline character. There are other escape sequences such as `\t` (tab) or `\b` (backspace). The main idea of an escape sequence is to represent a special character or a control character that is otherwise difficult or impossible to enter into a string.

```
>>> s = 'hello \n world'
>>> print(s)
hello
 world
```

Every escape sequence starts with the backslash \, which in this context is called the **escape character**. The escape character indicates that the following character(s) have special meaning and/or should be treated differently.

While the escape character can be used to indicate the beginning of an escape sequence, it can also be used to escape the subsequent character, e.g. `\'` (escape the single quote) and `\"` (escape the double quote). For example in the following example, the escape character is used to escape the subsequent single quote `'`, indicating that it should be treated as a literal character and not as the closing quote of the string.

```
>>> s = 'hello \' world'
>>> print(s)   # The string contains a single quote.
hello ' world
```

As another example, if we would like to get a backslash as a literal character, we need to prepend it with the escape character (which is another backslash). So in the following example, the first backslash is the escape character, while the second backslash is the actual backslash (the literal character) we would like to have in the string.

```
>>> s = 'hello \\ world'
>>> print(s)  # The string contains one backslash.
hello \ world
```

So how can we add the literal character \ followed by the literal character `n` when defining a string? If we enter `\n` into the string, it will be interpreted as a newline escape sequence, which is not what we want in this case:

```
>>> s = 'hello \n world'
>>> print(s)
hello
 world
```

The issue here is that the first backslash is not interpreted as a backslash (the literal character), but instead as the escape character. On the other hand, if we want a backslash as a literal character, we need to prepend it by the escape character, i.e. we need to enter `\\` into the string as we have just seen in the previous example before. After that we still need to add the literal character `n`. We can enter `n` directly into the string because there is no connection to the preceding backslash any more (the backslash preceding `n` is no longer interpreted as the escape character).

```
>>> s = 'hello \\n world'
>>> print(s) # String `s` contains one backslash followed by 'n'.
hello \n world
```

As you can see, in some cases it may be a bit cumbersome if you would like to define a string that contains the literal character \. This is a reason why Python has raw strings. A **raw string** is when you add `r` in front of the string, which **disables the special interpretation of the escape character** \. When defining a raw string, the backslash does not have any special meaning, so it would be treated as a literal character, not as an escape character. Using raw strings, we can now make the previous examples more concise and easier to understand:

```
>>> s = r'hello \ world'
>>> print(s)
hello \ world
>>> s = r'hello \n world'
>>> print(s)
hello \n world
```

We will see raw strings again in Section 12.3 starting on page 156 when dealing with regular expression (**regex**) patterns. It is usually a good idea to use raw strings for regex patterns. The reason is that regex patterns often contain backslashes and we would like to treat them as a literal characters when defining a string containing a regex pattern.

### 6.5.3 Identifiers and Variables

**Identifiers** can be variable names, function names, class names, and so on. You can basically use any name you like as long as it doesn't start with a number or special symbol such as > and doesn't have spaces or dashes in it.

```
# VALID:
hello_world
helloWorld
hello_world8
# INVALID:
8hello
>hello
with spaces in between
hello-world
```

Assigning things to variables is very straightforward in Python.

```
>>> x = 8
>>> x = x + 3   # Add three.
>>> print(x)
11
>>> x += 1      # Shorter: add one.
>>> print(x)
12
```

You can delete identifiers using `del`:

```
>>> x = 10
>>> x
10
```

```
>>> del x       # Delete ‘x‘.
>>> x           # ‘x‘ can no longer be found.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name ’x’ is not defined
```

Sometimes you have a **very long line of code and you want to break it into several lines for better readability**. Backslash to the rescue! Whenever you're not finished typing your command but want a newline to make your code more readable, just add a backslash at the end. But be careful, **after the backslash should not be any whitespace, otherwise it won't work**.

```
x = \
    ’This is a really long’ + \
    ’string and I want to break’ + \
    ’out the code across’ + \
    ’several lines.’
```

### 6.5.4 Whitespace (Indentation) Has Special Meaning in Python

Speaking of whitespace, in Python code you might notice that there are very few curly brackets like in other languages. Here are two examples of a function definition, one in R, the other in Python.

```
# Function definition in R.
f = function(x) {
    y = x^2
    return(y)
}
# Function definition in Python.
def f(x):
    y = x ** 2
    return y
```

As you can see, what really matters in Python is whitespace, or in this specific case, indentation. **In Python, indentation has a special meaning** and the best practice is to use four spaces for indentation. For example, in R you have curly braces around the function definition so whitespace is irrelevant and the following code would work in R. In Python however the following code would fail to work because whitespace is part of the syntax.

```
# Would work in R (although it’s ugly).
f = function(x) {
y = x^2
return(y)
}
# Would NOT work in Python as whitespace has meaning.
def f(x):
y = x ** 2
return y
```

### 6.5.5 Arithmetic, Comparison, and Logic with Operators

Operators generally behave like functions, but they usually have a special syntax. For example, you could write a function called `plus` and calculate `plus(5, 2)` which would yield 7, but I guess we can all agree that writing `5 + 2` is much easier and more direct.

The most important **operators** in Python are as follows:

```
+, -, *, /             : basic arithmetics
**, //, %              : power, divide and floor, modulo
<, <=, >, >=, ==, != : less/greater, equal to, not equal to
not, and, or           : boolean operators
```

For example:

```
>>> 8 ** 2     # 8 to the power of 2.
64
>>> 7 // 2     # Divide and floor.
3
>>> 7 % 2      # Modulo.
1
>>> 8 < 10
True
>>> 8 == 8
True
>>> 8 != 10
True
>>> True and False
False
>>> True or False
True
>>> not True
False
```

Sometimes you want to assign a calculation back to the variable. In this case there are some **shortcuts for using operators**. The basic idea is simply that you add the operator you want to apply in front of the equality sign.

```
>>> x = 8
>>> x += 2
>>> print(x)
10
>>> x *= 3
>>> print(x)
30
```

### 6.5.6 if, for, while

`if`, `for`, and `while` are three key control flow statements in Python. They allow you to tell Python to do different things depending on different situations.

```
my_preference = 'yes to NLP'
if my_preference == 'yes to NLP':
    print('I love NLP')
else:
    print('I love text analytics')
```

This the basic `if` statement, but you can also make it more versatile by adding a few other conditions with one or several `elif`. But keep in mind that `else` and `elif` are optional.

```
g = input('What do you like: ')   # Use `int()` if need to convert to int.
if g == 'NLP':
    print('Yay for NLP!')
elif g == 'text analytics':
    print('Yay for text analytics!')
else:                             # Neither NLP or text analytics.
    print('OK')
```

Another way, if you only want to return some simple values in the if statement, you can also use a dictionary.

```
dictionary = {'Larry': 'larry@wall.org', 'Spammer': 'spammer@hotmail.com'}
g = input('Larry or Spammer? ')
dictionary[g]        # Error if not in `dictionary`.
dictionary.get(g)    # Works always (returns `None` otherwise).
```

The next thing we're going to cover are loops. There are two kinds of loops, and the first one is the while loop. The following loop prints `x` and decreases `x` by one each time it runs. At the beginning, it always checks whether `x` is still at least one (or greater) and if this condition is no longer fulfilled, it stops. So in total, the following loop just prints the numbers eight to one in a decreasing sequence.

```
x = 8
while x >= 1:
    print(x)
    x -= 1
```

Here's another example, where we check a condition each time inside the loop and set a variable `ct` that determines whether to continue the while loop. The condition is random. We draw a random number between 1 and 100, and if that number is less than 5 we let the loop stop. So with a 5% probability we will stop and we will continue with a 95% probability each time the loop is run. To better monitor how often the loop runs, we print "NLP" each time it runs.

```
import random
ct = True     # Variable that will determine whether to continue.
while ct:
    print('NLP')
    if random.randint(1, 100) < 5:
        ct = False
```

The next kind of loop is a for loop. It just iterates over a sequence of objects. In the following example, it just prints the numbers one to eight.

```
for i in range(1, 8):
    print(i)
```

In fact, `range()` returns a so-called iterator, whose main purpose in this case is to return one number at a time. If you want to get all numbers at the same time, you can use `list(range(8))` for example.

If you want more control over loops, you can use the `break` statement. The following example will only print the numbers one to five, because the loop will stop if `i` is larger than five.

```
for i in range(1, 8):
    if i > 5:
        break
    print(i)
```

Here's an example of how `break` works in the `while` loop. The following loop is going to print out the numbers one to eight, because it breaks for any larger number.

```
x = 0
while True:
    x += 1
    if x > 8:
        break
    print(x)
```

Sometimes you want to skip the rest of the statements in the current loop block. `continue` lets you do exactly that. As you can see, the first print statement is executed each time, but the second one only the first three times. The reason is that after the first three times, `continue` is run, so the rest of the for loop is skipped.

```
>>> for i in range(1, 8):
...     print('First print statement,  i =', i)
...     if i > 3:
...         continue
...     print('Second print statement, i =', i)
...
First print statement,  i = 1
Second print statement, i = 1
First print statement,  i = 2
Second print statement, i = 2
First print statement,  i = 3
Second print statement, i = 3
First print statement,  i = 4
First print statement,  i = 5
First print statement,  i = 6
First print statement,  i = 7
```

You can use `continue` also in a while loop. The following code will have the same output as the previous for loop.

```
x = 0
while x < 7:
    x += 1
    print('First print statement,  x =', x)
    if x > 3:
        continue
    print('Second print statement, x =', x)
```

### 6.5.7 Defining and Using Functions

You write functions because you have code that you want to reuse, maybe with different parameters. A simple function that takes its function argument `x`, squares it, and returns the squared value is given here:

```
def f(x):
    return x ** 2
```

You can check that it performs as expected:

```
>>> f(3)
9
>>> f(8)
64
```

If course, you can also do other things in functions such as print or do other calculations.

```
def f(x, y):
    print('Multiplying both arguments with each other yields', x * y)
    return x * y
```

Again it works as expected. In the first function call, the function prints the message and then returns the output value of 24, which is directly printed by Python. In the second call, we save the output value to a variable x and then take a look at that variable (which, as we find out, contains again 24).

```
>>> f(3, 8)
Multiplying both arguments with each other yields 24
24
>>> x = f(3, 8)
Multiplying both arguments with each other yields 24
>>> x
24
```

You can also omit the `return` statement, in which the function by default will return `None`.

```
def f(x, y):
    print('Multiplying both arguments with each other yields', x * y)
```

You can check its output as follows:

```
>>> x = f(3, 8)
Multiplying both arguments with each other yields 24
>>> x == None
True
```

Unless you define a variable inside a function as a global variable (which is usually a bad idea, so we are not going to cover this), it will be treated as a **local variable**. It simply means that whatever happens inside the function stays inside the function and has no effect on anything outside the function. The following code illustrates what that means concretely.

```
def f(x):
    print('Local variable x =', x)
    x += 100
    print('Local variable x changed to', x)
```

If we run it as follows, notice that the variable x outside the function is not changed at all if we run the function. The reason is that **the variable** x **inside the function is a completely separate entity than the variable** x **outside the function**.

```
>>> x = 8
>>> f(x)
Local variable x = 8
Local variable x changed to 108
>>> x       # x is still 8!
8
```

Sometimes you have functions that have many function arguments, for example:

```
def f(a, b, c, d, e, f):
    return a * b * c * d * e * f
```

Suppose that for some reason, most of these arguments are usually equal to one. In that case you can just specify the **default argument values**, which means you don't have to supply them each time.

```
def f(a = 1, b = 1, c = 1, d = 1, e = 1, f = 1):
    return a * b * c * d * e * f
```

You can now run it, and if you don't specify an argument, it will be equal to one by default.

```
>>> f()
1
>>> f(3)
3
>>> f(3, 8)
24
```

Another thing that is useful when defining or using functions is to use **keyword arguments**.

```
def f(x = 10, y = 20, z = 30):
   print('x =', x, ', y =', y, ', z =', z)
```

Now you can use it as follows:

```
>>> f(1, 2, 3)          # Normal usage.
x = 1 , y = 2 , z = 3
>>> f(x = 1, y = 2, z = 3)  # Specify keyword arguments.
x = 1 , y = 2 , z = 3
>>> f(z = 3, y = 2, x = 1)  # Order doesn't matter any more!
```

```
x = 1 , y = 2 , z = 3
>>> f(z = 3, x = 1)         # If omit one, use default values.
x = 1 , y = 20 , z = 3
>>> f(z = 3)                # If omit several, use several defaults.
x = 10 , y = 20 , z = 3
```

As we have seen Section 6.4 you can (and should) use a docstring for most of your function. It's simply a bit of documentation that lives right in the function definition and gives the user some idea of what the function does and what its purpose is.

```
def f(x):
    '''This is the docstring of this
    function. It describes what it does
    and why it does it.
    '''
    return 2 * x
```

You can now take a look at this docstring by typing `help(f)` or by typing `print(f.__doc__)`.

Sometimes you may see a function called in this way: `f(**d)`. In Python, `**` is used to **unpack a dictionary into keyword arguments for a function or method**. A dictionary passed like this will have its keys treated as parameter names and its values as corresponding argument values. (Note that this use of `**` is completely unrelated to exponentiation, i.e. use of `**` as a power operator.)

```
def f(a, b, c):    # Define example function.
    print(a, b, c)

d = {'a': 1, 'b': 2, 'c': 3}    # A dictionary.
f(**d)      # Equivalent to calling `f(a=1, b=2, c=3)`.
```

We will discuss dictionaries in greater detail in Section 6.7.5 starting on page 73.

## 6.6 Object Oriented Programming

Python is a multi-paradigm programming language. You can do object-oriented, imperative, functional, and procedural programming with Python. However, if one would have to put Python into one category only, it would probably be the object-oriented corner. We have already seen how you can assign values to variables and how to define functions. Object-oriented programming in Python takes this one step further and puts more structure on this.

### 6.6.1 Class Example #1

Let's look at a very simple introductory example. Here we define the simplest class we can think of, a class that does nothing. The pass statement in the class definition below does nothing. You can use it when a statement is syntactically required but you don't want any action to be performed. In this case, we just use it as a placeholder so that we can at least define the class c.

```
class c:
    pass       # Do nothing
```

Having defined a class is a good start, but we need to instantiate it next. The basic idea is that a class is a blueprint for something, but we first need to **instantiate** it before we can do anything with it. We instantiate two objects of that class and then print them. Both objects conform to the blueprint given by the class (i.e. do nothing), and you can see from the print statement that they reside in different addresses in computer memory, so they are really two distinct objects. The reason why we have instantiated two objects is simply to show that the class is just a blueprint, and you can create many objects from that blueprint.

```
>>> o1 = c()
>>> o2 = c()
>>> print(o1)
<__main__.c object at 0x7fcad25a4438>
>>> print(o2)
<__main__.c object at 0x7fcad25a44e0>
```

### 6.6.2 Class Example #2

Let's create a bit of a more sensible class that actually does something.

```
class c:
    def hello(self):          # Define method ‘hello()‘.
        print('Hello world.')
```

Here we create only one instance of this class for illustration, although we could, as before, in principle create as many as we wish.

```
>>> o = c()   # Create object ‘o‘, which is an instance of class ‘c‘.
>>> o.hello() # Call method ‘hello()‘ of object ‘o‘.
Hello world.
```

You can see that you can call the **method** `hello()` of the class by calling it after the object, separated by a dot. We will explain later why `self` has to be specified as an argument to the `hello()` method. For now, all you need to know is that **any method (i.e. a function defined as part of a class) has to have as its first argument** `self`, **even though it doesn't always seem to be used**.

### 6.6.3 Class Example #3

A classic introductory example is if you are dealing with people. You want to represent a person that can say "hi" and that has an age. **A class is blueprint of an object**, so for example, to create a person object, you first need to provide the blueprint.

```
class person:
    def __init__(self, n, a):
        self.name = n
        self.age = a

    def say_hello(self):
        print('Hi, my name is', self.name)

    def say_age(self):
        print(f'I am {self.age} years old')
```

Remember that indentation is important in Python. When you are inputting this class in a text editor or directly into Python, make sure to add four spaces in the empty lines between the function (in fact, "method") definitions.

We will go through this class definition (the blueprint) in more detail shortly. But before we do that, let's quickly see how to use it. In what follows we will **instantiate** two objects created from this class. These are two different persons with their respective names and ages.

```
g = person('Guido', 25)
l = person('Larry', 30)
```

We can now let these persons tell us their names and ages using the `say_hello()` and `say_age()` **methods**.

```
>>> g.say_hello()
Hi, my name is Guido
>>> g.say_age()
I am 25 years old
>>> l.say_hello()
Hi, my name is Larry
>>> l.say_age()
I am 30 years old
```

You could also access the `name` and `age` variables directly. These variables that are part of a such an object (which is an instantiated class) are called **fields**.

```
>>> g.name
'Guido'
>>> g.age
```

```
25
>>> l.name
'Larry'
>>> l.age
30
```

In general, fields and methods are collectively called **attributes** of a class.

Now let's go step by step through this class definition and explain every detail.

1. There are three function definitions in the class object, i.e. we define `__init__()`, `say_hello()`, and `say_age()`. As they are part of a class, each of these functions is called a "**method**." The reason why we use a special name und why we don't call them "function" any more is because they are closely tied to the class we just have defined, and **only work on objects of this class**.

2. What is **"self"**? It refers to the object that will be created using this class (remember, the class is just the blueprint). So why is it included in all the method definitions when you don't usually use it inside the method definitions (with the exception of `__init__()` which we'll talk about shortly)? There is a technical reason for this. Suppose you have created an instance g of the class `person`. Whenever you call `g.method(x1, x2)`, what happens is that Python behind the scenes actually calls `person.method(g, x1, x2)`. That's why the first argument of each method is always `self`. So in this call, `self` gets set to g, which is the instance of the class.

3. The `__init__()` method is also sometimes called an **initializer method** (which is a special case of a magic method, see Section 6.6.6 starting on page 63). It is called when an object of that class is instantiated. In this example, the `__init__()` method sets the field's `name` and `age` of the instantiated object.

### 6.6.4 Class Variables and Object Variables

What we have seen so far are object variables, e.g. `name` and `age`. They are specific to the object instantiated, and they can have different values for each object. In contrast, sometimes you might want to have **class variables** that are the same for all objects that are instantiated from a given class.

```
class c:
    class_var = 0                   # Class variable.
    def __init__(self, ov):
        self.object_var = ov        # Object variable.
        c.class_var += 1            # Update the class variable.
```

Now let's create to instances of this class and let's see what happens to the class and object variables. The point of the following code is to show that the class variable is the same for all objects of this class.

```
>>> x = c(8)       # Create the first instance of class ‘c‘.
>>> x.object_var   # Check the object variable.
8
>>> x.class_var    # Check the class variable.
1
>>> y = c(88)      # Let’s create another instance.
>>> y.object_var   # As expected.
88
>>> y.class_var    # Class variable has been increased.
2
>>> x.object_var   # Still the same.
8
>>> x.class_var    # Has increased as well, even though it’s a different object.
2
```

If you would like to change the class variable, there’s a small caveat to keep in mind. You cannot just “overwrite” the class variable of an object. The reason is that if you use an object variable that has the same name as class variable, it will hide the class variable. For example:

```
>>> x.class_var = 300
>>> x.class_var # What we are seeing now is the object variable of the same name.
300
>>> y.class_var    # ‘y‘ still displays the class variable.
2
```

If you want to avoid this, you can add a **class method** that takes care of dealing with the class variable.

```
class c:
    class_var = 0                   # Class variable.
    def __init__(self, ov):
        self.object_var = ov        # Object variable.
        c.class_var += 1            # Update the class variable.

    @classmethod     # Class method can work on class variable directly.
    def update_class_var(cls, new_cv_val):
        cls.class_var = new_cv_val    # Set class variable to new value.
```

Now we can use this class method whenever we want to update the class variable:

```
>>> a = c(8)
>>> b = c(88)
>>> a.class_var
2
>>> b.class_var
```

```
2
>>> c.update_class_var(300)
>>> a.class_var
300
>>> b.class_var
300
```

### 6.6.5 Callable Objects

An object with a `__call__` method is called a **callable object**. This special method, `__call__`, **allows an instance of a class to be invoked as if it were a function**. When the `__call__` method is defined in a class, you can create an instance of that class and then "call" the instance like a regular function. Behind the scenes, Python will execute the `__call__` method whenever the object is called with parentheses `()`.

```
class c:
    def __call__(self, name):
        print(f'Hello, {name}!')

# Create on object of this class. Due to the existence of the
# '__call__()' method, this object is a "callable object." It
# means we can call the object as if it is a function.
o = c()      # Create object 'o' of class 'c'.
o('Bob')     # Output: "Hello, Bob!", same as 'o.__call__('Bob')'
```

The `__call__` method is an examle of a magic method in Python, see Section 6.6.6 starting on page 63 for further details.

### 6.6.6 Magic Methods

**Magic methods**, also known as **dunder methods** (short for "double underscore"), are special methods in Python that allow you to define the behavior of objects for built-in operations such as arithmetic, comparisons, or container-like access. The special behavior of these methods is **predefined by Python**. Each magic method starts and ends with double underscores. Examples of common magic methods:

- `__init__()` initializes a new instance of a class, discussed in Section 6.6.3 starting on page 60.
- `__str__()` provides a readable string representation of an object.
- `__getitem__()` allows objects to support indexing, see example below.
- `__call__()` makes an object callable like a function as discussed in Section 6.6.5 starting on page 63.

The `__str__()` method in Python is a magic method used to define a **human-readable string representation of an object**. This method is called automatically when you use the `print()` function or the `str()` function on an object. By overriding `__str__()` in a class, you can control how your object is displayed to users, making it more descriptive and meaningful instead of showing the default representation, i.e. the memory address. This is particularly useful for debugging, logging, or providing user-friendly output.

```
class c:
    def __str__(self):
        return f'Title={self.title}'

o = c()
o.title = 'The Tale of Peter Rabbit'  # Set field `title` of object `o`.
print(o)       # Output: Title=The Tale of Peter Rabbit
str(o)         # Output: 'Title=The Tale of Peter Rabbit'
```

The `__getitem__()` magic method allows objects to emulate container types like lists, dictionaries, or tuples, **enabling indexing and slicing functionality**. Here's a brief example of `__getitem__()`:

```
class c:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, index):
        return self.data[index]

o = c([10, 20, 30]) # Instantiate class `c` to create object `o`.
o[2]                # Output is `30`, the result of `o.__getitem__(2)`.
o = c('hello')
o[1]                # Output is 'e', the result of `o.__getitem__(1)`.
```

In this code example, the class `c` uses the `__getitem__()` method to **provide index-based access** to one of its internal data structures, `self.data`. In this example the internal data structure is a list or string, making it behave like a standard Python list or string for indexing, but you can use it for any customized data structure that is suitable for indexing. For example, the `__getitem__()` **magic method is used in gensim** when indexing an object of class `TfidfModel`, see Section 16.2 starting on page 187.

### 6.6.7 Inheritance

In finance you can think of a stock and a European call option on a stock as financial instruments. Both are traded on the market and thus both have a price. What is unique about the stock is that we should also keep track of its volatility. And what is unique about the European call is that it has a strike price. (We ignore for now that the call also depends on a few other parameter, including the volatility of the stock.) So let's capture this hierarchy using different classes that build on each other.

Specifically, we first define a class `financial_instrument` that has a `price` field. Then we define a `stock` class and a `option` class. Both of them build on `financial_instrument` and can add further fields (as in the example below) or in principle further methods as well. The reason why this can be a good idea is that we don't have to reinvent the wheel twice, i.e. we don't have to deal with `price` again, as `financial_instrument` already takes care of it. In this example, `financial_instrument` is called the **base class** or **superclass**, while `stock` and `option` are called the **derived classes** or **subclasses**.

```
class financial_instrument:          # Define the base class.
    def __init__(self, prc):
        self.price = prc

class stock(financial_instrument):   # Define a derived class.
    def __init__(self, prc, vol):
        financial_instrument.__init__(self, prc)
        self.vola = vol

class option(financial_instrument):  # Define another derived class.
    def __init__(self, prc, K):
        financial_instrument.__init__(self, prc)
        self.strike = K
```

Now we can instantiate a stock and an option.

```
>>> s = stock(100, 0.20)
>>> o = option(5, 80)
>>> # Both objects ‘s‘ and ‘o‘ have a price, managed by the base class.
>>> s.price
100
>>> o.price
5
>>> # ‘s‘ has a volatility while ‘o‘ has a strike.
>>> s.vola
0.2
>>> o.strike
80
>>> # ‘s‘ does not have a strike, which makes it different from ‘o‘.
>>> s.strike
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: ’stock’ object has no attribute ’strike’
>>> # ‘o‘ does not have a volatility, which makes it different from ‘s‘.
>>> o.vola
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'option' object has no attribute 'vola'
```

### 6.6.8 Context Managers

Context managers in Python are a feature that simplify the management of resources, such as files, network connections, or database sessions, by automating setup and cleanup operations. The key advantage of context managers is that they ensure resources are properly acquired and released, even when exceptions or errors occur. This makes it easier to write reliable and clean code, particularly when working with resources that need explicit closing or cleanup, such as file handles.

In Python, the `with` statement is used to work with context managers. It automatically handles the setup and cleanup of resources, so you don't have to worry about explicitly opening and closing them. For instance, when reading from a file, you might need to open the file, read its contents, and then close it. Without a context manager, if an error occurs during the reading process, the file might not be closed properly, leading to potential resource leaks. Using the `with` statement ensures that the file is automatically closed when the block of code is completed, even if an error occurs, making the code safer and more robust.

Here are two code examples that illustrate why using a context manager makes sense. In the first example, we do not use a context manager to open and read from a file. As you can see, the code is quite verbose.

```
try:
    file = open('example.txt', 'r')
    content = file.read()
    # Simulate an error in the middle of the operation.
    x = 1 / 0  # ZeroDivisionError.
    print(content)
finally:
    # File is manually closed, but only in the `finally` block.
    file.close()
```

Now let's see how we can make this code more concise when using the `with` statement. Here we do not have to use the `try` and `finally` statements at all.

```
with open('example.txt', 'r') as file:
    content = file.read()
    # Simulate an error in the middle of the operation.
    x = 1 / 0  # ZeroDivisionError
    print(content)
```

The `with` statement works with context managers. A **context manager** is any object that implements the following two methods (which are magic methods, by the way; see Section 6.6.6 starting on page 63):

- `__enter__()` contains the code that is executed when entering the context, e.g. when opening a file.
- `__exit__()` contains the code that is executed when exiting the context, e.g. when closing a file.

In the code example above, the `open()` function creates and returns an object which is an instance of a class that (among others) implements the two context management methods `__enter__()` and `__exit__()`.

To better understand how the `with` statement works with context managers, let us implement a custom context manager from scratch for illustration:

```
class c: # Class that instantiates context manager objects.
    def __enter__(self):
        print("Entering the context.")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context.")

# Using ‘with‘ alongside the custom context manager:
with c() as my_context_manager:
    print("Inside the context.")
```

The output is:

```
Entering the context.
Inside the context.
Exiting the context.
```

This code example defines a custom context manager class `c` that implements the `__enter__()` and `__exit__()` methods, which are required for an object to be used in a `with` statement. When an instance of class `c` is used with the `with` statement, the `__enter__()` method is called first, printing “Entering the context” and returning the object itself, assigned to `my_context_manager`. The code inside the `with` block (“Inside the context.”) is then executed. After the `with` block completes—whether normally or due to an exception—the `__exit__()` method is called, printing “Exiting the context.” This ensures proper setup and cleanup around the block’s execution.

## 6.7 Data Structures

### 6.7.1 Overview

Python has the following **basic data types**:

- Boolean: Takes values `True` or `False`.

- Numeric types: `int` for integers, `float` for floating point numbers, and `complex` for complex numbers.
- Stings: Immutable sequences of Unicode code points, e.g. `'hello world'`. For more details see Section 6.5.2 and Chapter 12 starting on pages 46 and 153, respectively. You can also do slicing/indexing on strings, see Section 6.8 starting on page 76, which means that a string is a sequence data type. In fact, **a string is an iterable** (see also Section 6.12.4 starting on page 87) because it is a sequence, see:

```
from collections.abc import Sequence, Iterable
isinstance('', Sequence)  # `True`.
isinstance('', Iterable)  # `True` because all sequences are iterables.
```

Python also has the following built-in data structures called **containers**. Containers are objects that hold an arbitrary number of other objects. We will discuss each one of them in turn. Their **main purpose is to hold data**.

1. list, see Sections 6.7.2 and 6.7.3 starting on pages 68 and 71, respectively.
2. tuple, see Section 6.7.4 starting on page 71.
3. dictionary, see Section 6.7.5 starting on page 73.
4. set, see Section 6.7.6 starting on page 74.
5. collections, see Section 6.7.7 starting on page 75.

### 6.7.2 List

A **list** is a sequence type, e.g. `['item_1', 'item_2', 'item_3']`. While a list can be a heterogeneous collection (i.e. containing mixed data types such as `['hello', 8, True]`), it is most frequently used for **homogeneous** collections containing the same data types (e.g. strings as in `['NYC', 'HK', 'London']`).

Lists can be **nested**, i.e. they have another list as an item, e.g. `['hello', [1, 2, 3]]`. Lists are **mutable**, which means you can change/update them:

```
>>> l = [2, 4, 6, 8]
>>> l[0] = 10
>>> l
[10, 4, 6, 8]
```

In this course, the list is probably the most important data structure overall. They contain a sequence of objects. For example, we can use it to hold a whole corpus, i.e. a sequence of text documents.

```
['Great job. Let us move on now',   # First text document.
 'I like NLP. It is fun.']          # Second text document.
```

We can also use a list to hold the words in a single text document. It is a very powerful and versatile data structure.

```
>>> l = ['I', 'love', 'NLP']   # One text document, split up into words.
>>> len(l)            # Length of list.
3
>>> for item in l:    # Loop over the items of the list
...     print(item)
...
I
love
NLP
>>> l.append('and text analytics')  # Add new element to the end.
>>> l.sort()                        # Sort the list.
>>> l
['I', 'NLP', 'and text analytics', 'love']
>>> del l[0]                        # Delete first element.
>>> l
['NLP', 'and text analytics', 'love']
```

To manipulate lists, people often use a **list comprehension**. This is a shortcut for looping over the items of the list and doing something with them. For example, the following list comprehension creates a new list where `_hello` is added to each item, unless the item is `'NLP'`.

```
>>> l = ['I', 'love', 'NLP']
>>> [item + '_hello' for item in l if item != 'NLP']
['I_hello', 'love_hello']
```

Another example illustrating a list comprehension:

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

**zip** can be used to **iterate through any two iterables**, e.g. through two lists.

```
>>> alist = ['a1', 'a2', 'a3']
>>> blist = ['b1', 'b2', 'b3']
>>> for a, b in zip(alist, blist):
...     print(a, b)
...
a1 b1
a2 b2
a3 b3
```

Next we will show a few list methods and functions that are useful when dealing with lists. The difference between list methods and functions is that the list methods only work with an object of class list, while the functions work on objects of class list and also on objects of some other classes. The list methods would be called for example as `mylist.sort()`, while the functions would be called for example as `len(mylist)`. The following table shows useful list methods:

| *Method* | *Description* |
|---|---|
| `append()` | Add element to end of list |
| `extend()` | Add list to another list (can also done with +) |
| `insert()` | Insert item at the defined index |
| `remove()` | Remove item from the list |
| `pop()` | Removes and returns an element at the given index |
| `clear()` | Removes all items from list |
| `index()` | Returns index of first matched item |
| `count()` | Counts how many times an element has occurred in a list |
| `sort()` | Sort list in ascending order |
| `reverse()` | Reverse order of items |
| `copy()` | Returns shallow copy of list |

The followin table shows built-in functions that can be used for lists:

| *Function* | *Description* |
|---|---|
| `all()` | Returns `True` if all elements are `True` or if list is empty. |
| `any()` | Returns `True` if at least one element is `True`. `any([])` is `False`. |
| `enumerate()` | Adds a counter, so you can for example loop through the list and have an automatic counter. |
| `len()` | Length of the list (the number of items). |
| `list()` | Convert an iterable (tuple, string, set, dict) to a list. |
| `max()` | Largest item in the list. |
| `min()` | Smallest item in the list. |
| `sorted()` | Returns a new sorted list (does NOT sort the list itself), e.g. `sorted(l, key=lambda x: x[1], reverse=True)`. |
| `sum()` | Sums all elements in the list. |

### 6.7.3 Lists and References

Suppose you have a list saved in a variable and assign it to a new variable. What happens in Python is actually that **the assignment does not copy the list, but it copies the reference to the list**. A reference is just pointing to the memory location. So in the code below you have both x and y pointing to the same location in memory. If you change x, you also change y.

```
>>> x = ['hello', 'world']
>>> y = x
>>> del x[0]  # Delete first element from 'x'.
>>> y         # Also deleted from 'y'!
['world']
```

This is fine if that's what you're trying to accomplish, e.g. in order to use computer memory more efficiently. However, if you want to **copy the list**, you can use one of the following ways to create a **shallow copy**:

```
x = ['hello', 'world']
mycopy_a = x.copy()     # Use 'copy()' method.
mycopy_b = x[:]         # Slicing.
mycopy_c = list(x)      # Use built-in 'list()' function.
```

### 6.7.4 Tuple

Tuples on first sight look like lists, but they are usually used in a different way. While a **list** is usually used for objects that **appear in a sequence**, e.g. different readings in time from a sensor, with a **tuple** it's more about the **position of the elements**, e.g. in the form of coordinates such as (longitude, latitude). If you switch two elements in a list, it usually does not make a big difference. But if you switch two elements in a tuple, it might matter a lot (depending on what kind of data it contains).

A key difference in how you use tuples vs. lists is that a **tuple has structure while a list has order**. For example, you could store information about a customer, e.g. name, gender, and age e.g. `('Bob', 'male', 32)`. Or you would use a tuple to store coordinates like `(latitude, longitude)` or locations in a book such as `(page_number, line_number)`. On the other hand, you would use a list if you take a trip and record the city names you visit e.g. `['Hong Kong', 'London', 'New York']`.

A tuple is a sequence type (like a list), but unlike a list it is **immutable**. Being immutable for a tuple means that once the tuple is created, its elements cannot be changed, added, or removed. The size and content of the tuple remain constant throughout its lifetime, making it a suitable data structure for representing fixed collections of values. The immutability makes sense for a tuple, e.g. you would not want to swap longitude and latitude. However, for a list, mutability makes sense, e.g. if you change your travel plans and visit the cities in a different order. Because a tuple is immutable, it can be used as a key in a dictionary (see Section 6.7.5 starting on page 73).

The following code illustrates the immutability of a tuple:

```
t = (1, 2, 3)
t[0] = 8         # Error: Cannot change element.
t.append(7)      # Error: Cannot add element (increase size).
del t[0]         # Error: Cannot remove element.
```

Here is some example usage of tuples, with comments explaining the code inline. Keep in mind that **indexing in Python starts at zero**! Also keep in mind that you can use a tuple recursively, i.e. it can contain elements of different types, e.g. it can have another tuple as an element.

```
>>> fi = ('stock', 'call option', 'put option')   # Financial instruments.
>>> len(fi)
3
>>> new_fi = ('asian option', fi)   # Recursive usage.
>>> new_fi
('asian option', ('stock', 'call option', 'put option'))
>>> new_fi[1]
('stock', 'call option', 'put option')
>>> new_fi[1][2]
'put option'
```

**There is no tuple comprehension** like a list comprehension, but if you really need it you can use a **generator expression** (see Section 6.12.6) and convert it into a tuple again.

```
>>> tuple(i + 3 for i in (1, 2, 3, 4, 5) if i >= 2)
(5, 6, 7, 8)
```

Keep in mind that if the task at hand seems to require a tuple comprehension, you should ask yourself whether you're using the wrong data structure, i.e. whether it might be better to use a list instead of a tuple.

A word of caution: Tuples are immutable in the sense that the membership of a tuple cannot be changed, but **mutable elements (e.g. a list) of a tuple *can* be changed**. The following code works because the tuple contains a *reference* to where the contents of list `lst` are stored in computer memory. So we can modify this part of the computer memory by modifying the list `lst` itself. However, within the tuple, we cannot change the *reference* to the memory location, i.e. let it point to a different location in memory that contains another data structure.

```
>>> lst = [1, 2, 3]
>>> tpl = (lst, 7, 8)
>>> tpl
([1, 2, 3], 7, 8)
>>> lst.append(6)    # Can modify the list itself.
>>> tpl              # Tuple has changed.
([1, 2, 3, 6], 7, 8)
```

```
>>> tpl[0] = [6, 3, 2, 1] # Cannot change reference to different memory location.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You might be wondering whether you could change the tuple as follows, but it does not work:

```
>>> lst = [6, 3, 2, 1]
>>> tpl
([1, 2, 3, 6], 7, 8)
```

The reason is that we lose the pointer to the memory location if we overwrite `lst` by assigning something new to it. However, `tpl[0]` still points to the unchanged location in memory, and therefore does not show any changes.

When a tuple contains mutable element(s), you can avoid the sometimes a bit unpredictable behavior of a tuple by using `deepcopy` on the tuple.

```
>>> from copy import deepcopy
>>> lst = [1, 2, 3]
>>> tpl = (lst, 7, 8)
>>> tpl
([1, 2, 3], 7, 8)
>>> tpl = deepcopy(tpl)
>>> lst.append(6)
>>> lst              # List has changed.
[1, 2, 3, 6]
>>> tpl              # Tuple has not changed.
([1, 2, 3], 7, 8)
```

### 6.7.5 Dictionary

We have briefly seen the use of a dictionary before when we **unpacked** a dictionary for a function in Section 6.5.7 starting on page 55. Now we go into further details to explain exactly what a dictionary is.

A dictionary is like a phone book. It has a bunch of **key-value pairs**, which are like name-number pairs in a telephone book. Here is a similar example, but with email addresses. A dictionary uses curly braces, the keys and values are separated by a colon, and key-value pairs are separated by commas.

```
d = \
    {'Jeff': 'jeff@amazon.com',
     'Bill': 'bill@microsoft.com',
     'Larry': 'larry.page@google.com'}
```

Let's see what we can do with a dictionary.

```
>>> len(d)           # Number of key-value pairs.
3
>>> d['Jeff']        # Jeff's email address.
'jeff@amazon.com'
>>> del d['Bill']    # Delete Bill's entry.
>>> for key, value in d.items():     # Use ‘items‘ method.
...     print('{} has email {}.'.format(key, value))
...
Jeff has email jeff@amazon.com.
Larry has email larry.page@google.com.
>>> d['Guido'] = 'guido@python.org'   # Add Guido.
>>> if 'Larry' in d:                  # Check if it has a given key.
...     print('yay')
...
yay
```

You can have **dict comprehensions** similar to list comprehensions.

```
>>> {number: number * 2 for number in [1, 2, 3]}
{1: 2, 2: 4, 3: 6}
>>> d
{'Jeff': 'jeff@amazon.com',
 'Larry': 'larry.page@google.com',
 'Guido': 'guido@python.org'}
>>> {key: value for (key, value) in d.items() if key != 'Jeff'}
{'Larry': 'larry.page@google.com', 'Guido': 'guido@python.org'}
```

If you want to **sort a dict**, you need to use the items() method as well. While a dictionary maintains insertion order, you can sort it in various alternative ways:

```
d = {'z': 3, 'x': 4, 'y': 2}
sorted(d.items())                      # List sorted by keys.
sorted(d.items(), key=lambda x: x[0])  # List sorted by keys.
sorted(d.items(), key=lambda x: x[1])  # List sorted by values.
```

In the preceding code, the output is a list. If you would like to have a dictionary, you can convert the result again to a dictionary, for example:

```
dict(sorted(d.items()))
```

### 6.7.6 Set

Sets in Python are modeled after **sets in mathematics**. You can uses sets to test for membership. Every element can **occur only once** in a set. Also the **order of the set elements does not matter**. (If the order matters, you could use a list instead.)

```
s = set(['I', 'love', 'NLP', 'NLP'])  # Create set.
s                           # Note that duplicate elements are removed.
'I' in s
new_s = s.copy()            # See section on "References."
new_s.add('text analytics')
new_s
new_s.issuperset(s)
s.remove('NLP')
s
s.intersection(new_s)
```

There are also **set comprehensions**.

```
>>> s = {1, 3, 5}
>>> {x ** 2 for x in s}
{1, 9, 25}
```

If you want to get the intersection between a set and list:

```
myset.intersection(mylist)
```

**Difference between list and set**: lists are very nice to sort and have order while sets are nice to use when you don't want duplicates and don't care about order.

### 6.7.7 Collections

The collections module provides **specialized container data types** supplying alternatives to dict, list, set, tuple. Some useful ones for our purposes:

- **Counter** can be used for counting. (See also Section 15.4 starting on page 180 for an example how to use `Counter` for bag-of-words.) The resulting `Counter` object has a similar structure as a dictionary (it is a dict subclass).

  ```
  >>> from collections import Counter
  >>> Counter('a b A B B c'.lower().split())
  Counter({'b': 3, 'a': 2, 'c': 1})
  >>> c = Counter(['eggs', 'ham', 'eggs'])
  >>> c                     # View counts.
  Counter({'eggs': 2, 'ham': 1})
  >>> c.values()            # Only the word count numbers.
  dict_values([2, 1])
  >>> c.most_common(1)      # Most common word.
  [('eggs', 2)]
  >>> list(c.elements())    # List all the elements.
  ['eggs', 'eggs', 'ham']
  ```

  Keep in mind that `most_common` does not tell us whether there are more words with the same frequency.

- **defaultdict** is a subclass of dict, so it behaves basically like a dict. `defaultdict(int)` can be useful for counting. The basic idea is that it is very similar to a dict, but the difference is that it is initialized with a "default factory" (in this case the `int` function) for a previously nonexistent key. The `int` function simply returns zero. In other words, if a new key is added to the dict, it will be initialized with a value of zero. Alternatively we could as well write `defaultdict(lambda: 0)` which would have the same effect (it uses an anonymous function that always returns zero).

  ```
  from collections import defaultdict
  l = 'spam spam spam spam spam spam eggs spam'.split()  # List of food.
  d = defaultdict(int)    # Default value of int is zero.
  for food in l:
      d[food] += 1        # Increment value by one.
  ```

  Another more elaborate example for counting with `defaultdict(int)`:

  ```
  from collections import defaultdict
  l = [
      ("Lucy", 1),
      ("Bob", 5),
      ("Jim", 40),
      ("Susan", 6),
      ("Lucy", 2),
      ("Bob", 30),
      ("Harold", 6)
  ]
  d = defaultdict(int)
  for name, count in l:
      d[name] += count
  ```

- **namedtuple** is a tuple subclass that adds names to each field. For example, instead of the tuple `('Bob', 'male')` you would have `(name = 'Bob', gender = 'male')` with a named tuple.

## 6.8 Slicing

"Slicing" or "indexing" means extracting subsets of python objects. (See also the examples on slicing and stride in Section 12.1 starting on page 153.) It is important to remember that **indexing in Python starts with zero**. So for example `lst[0]` extracts the first item of the list `lst` from the following code.

```
>>> lst = ['hello', 'world', 'slicing', 'is', 'great']
>>> lst[0]    # First item.
'hello'
```

```
>>> lst[2]    # Third item.
'slicing'
>>> lst[len(lst) - 1] # Last item.
'great'
>>> lst[-1]   # Last item.
'great'
>>> lst[-2]   # Second to last item.
'is'
```

You can also perform slicing or indexing with strings in Python.

```
>>> s = 'hello world'
>>> s[0]        # First character.
'h'
>>> s[2:5]      # From index 2 (inclusive) to index 5 (exclusive).
'llo'
>>> s[:5]       # From beginning to index 5 (exclusive).
'hello'
>>> s[2:]       # Starting from index 2 (inclusive) to the end.
'llo world'
>>> s[-5:-2]    # From index -5 (inclusive) to index -2 (exclusive).
'wor'
>>> s[1:-1]     # Everything except for first and last.
'ello worl'
>>> s[:]        # All of it (no change).
'hello world'
```

**Stride** can be combined with slicing. Using a stride allows you to skip items. For example, it allows you to skip every second item.

```
>>> w = 'I love NLP.'
>>> w[2:9]     #
'love NL'
>>> w[2:9:1]   # Stride of one (same as before).
'love NL'
>>> w[2:9:2]   # Stride of two (skip every second).
'lv L'
>>> w[2:9:3]   # Stride of three (skip every third).
'leL'
>>> w[::-1]    # Negative stride, reverse string.
'.PLN evol I'
>>> w[::-2]    # Skip every other letter in reversed string.
'.L vlI'
```

## 6.9 Matrices

Python does not have matrices and support for tabular data (e.g. DataFrames) baked into the core language. This means that you need to use add-on packages. This is not necessarily a bad thing, but it illustrates the historically different focus Python has compared to, for example, the R programming language, which does include matrices "out of the box."

In some simple cases you could represent a matrix as a list of list without using any add-on packages. But as you can imagine, you will quickly see the limitations of this approach once you need to run more complicated computations such as matrix multiplication or matrix inversion. Besides the fact that the following code is difficult to read, it is also not computationally efficient, as it cycles through each row several times. The following code is an example of how you can transpose such a matrix using list comprehensions:

```
>>> matrix = [
...     [1,  2,  3,  4],
...     [5,  6,  7,  8],
...     [9, 10, 11, 12],
... ]
>>> [[row[col_index] for row in matrix] for col_index in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In Python you typically need add-on packages if you would like to deal with matrix data or tabular data. The most common packages are NumPy and SciPy. There is also another packages called Pandas which we discuss later in Appendix G starting on page 255. Pandas is often used for tabular data when each column can potentially have a different data type.

- For (dense) matrices you typically use NumPy, e.g. the `numpy.array` function, while SciPy is often used for sparse matrices, e.g. you can use the `scipy.sparse.csc_matrix` class (from the older matrix interface) or the `scipy.sparse.csc_array` class (from the new array interface, which is compatible with NumPy arrays). A *dense matrix* does not have many zero elements, while a *sparse matrix* contains many elements that are zero. For a sparse matrix, SciPy uses special computational techniques to increase the efficiency of storage and computations.

- NumPy is usually imported as follows:

  ```
  import numpy as np
  ```

- In contrast to NumPy, you usually do not need to import SciPy as a whole. Instead you selectively import certain subpackages from it. For example, the following code imports the `special` and `optimize` subpackages.

  ```
  from scipy import special, optimize
  ```

Alternatively you can selectively import specific classes or functions, for example:

```
from scipy.sparse import csc_array  # Import the ‘csc_array‘ class.
```

- NumPy stores data in **arrays**, which might be one-dimensional (a vector), two-dimensional (a matrix), or $n$-dimensional (`ndarray`) in general.

- For example, here we create a 2-dimensional array of size 2x3, composed of 4-byte integer elements, i.e. `np.int32`. Keep in mind when indexing the array that indices in Python start at zero (not at one like in R). `dtype` is the data type of the array.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> x
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)
>>> x.T                             # Transpose.
array([[1, 4],
       [2, 5],
       [3, 6]], dtype=int32)
>>> x[:, 0]                         # First column.
array([1, 4], dtype=int32)
>>> x[1, :]                         # Second row.
array([4, 5, 6], dtype=int32)
>>> type(x)
<class 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

- Ranges of numbers:

```
>>> np.arange(0, 1, 0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

- If you have worked before with other software such as MATLAB, don't get confused by the way NumPy stores its data. MATLAB stores data column by column (the so-called "Fortran order"), while **NumPy by default stores them row by row** (the so-called "C order"). This does not affect indexing, but may affect performance. For example, in MATLAB, an *efficient* loop will be over columns, e.g. `for n = 1:10 a(:, n) end`, while in NumPy it's preferable to iterate over rows, e.g. `for n in range(10): a[n, :]` (note that `n` is in the first position, not the last).

- SciPy has several different kinds of sparse matrices, e.g. see scipy.sparse. If you have a sparse matrix `M`, you can take a look at it by converting it to a dense matrix/array:

```
    M.toarray()   # Use the ‘toarray‘ method.
    M.A           # Access the ‘.A‘ attribute.
```

## 6.10 Saving Objects and Data to File

You can save any Python object to a file using **pickle**. The following example is from the official Python documentation. We start by creating some data and then save it to a file by using the `dump()` function.

```
import pickle
data = {           # Arbitrary collection of objects.
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}}
# 'wb' means "write in binary format."
with open('mydata.pickle', 'wb') as myfile:
    pickle.dump(data, myfile)
```

Next we can restart Python to have a “clean sheet,” on Windows and Linux by pressing `Ctrl + .` and on macOS by pressing `Cmd + .` in case you are using the Spyder IDE. Then we read the data back into Python by using the `load()` function:

```
import pickle
# 'rb' means "read from binary format."
with open('mydata.pickle', 'rb') as myfile:
    data = pickle.load(myfile)
```

We also discuss the pickle file format in Section G.3 starting on page 256 when introducing Pandas and DataFrames.

## 6.11 Modules and Packages

We have seen in Section 6.5.7 starting on page 55 how to make code reusable by wrapping it in functions. Once you start writing more code, you're likely going to have more and more self-written functions, and it is going to get increasingly difficult to manage them.

This is where the concept of a **module** comes in. You can easily create a module by putting your function (or class or variable) names into a text file whose file name ends with `.py`. Use any IDE or editor you like, for example an IDE/editor mentioned in Sections C.1 and C.2. Suppose for example you create a file named `my_first_module.py` that has the following content:

```
def sq(x):
    '''Function that squares
    its input.
    '''
    return x ** 2

__version__ = '1.0'
```

You can then import and use it as follows:

```
>>> import my_first_module
>>> my_first_module.sq(8)
64
>>> my_first_module.__version__
'1.0'
>>> help(my_first_module.sq)   # Display help, exit with ‘q‘.
```

You can specify a shortcut for the imported module. This can save a lot of typing.

```
>>> import my_first_module as mfm
>>> mfm.sq(8)
64
>>> mfm.__version__
'1.0'
```

If you don't always want to type `my_first_module.sq`, you can just import everything except names that start with `__` by using `*`. However, this practice is usually discouraged because there could be name clashes. For example, if there is another module that also has a function called `sq`, it will not be clear which function (from which module) is called when you type `sq(8)`.

```
>>> from my_first_module import *
>>> sq(8)          # Works fine.
64
>>> __version__     # Not imported because it starts with ‘__‘.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '__version__' is not defined
```

Alternatively, you can also import a few select functions, classes, or variables by explicitly specifying them, separated by commas:

```
>>> from my_first_module import sq, __version__
>>> sq(8)
64
>>> __version__
'1.0'
```

Depending on whether you import your module or run the module file standalone by itself, you sometimes might want your code to behave differently. Consider for example the file `investigate_module_name.py` with the following content.

```
print('__name__ has value', __name__)
```

If you import the module, it will have the following output:

```
>>> import investigate_module_name
__name__ has value investigate_module_name
```

On the other hand, if you run the following code on the Unix shell (i.e. you're running the module file standalone, all by itself, and don't import it), you will get a different output:

```
$ python investigate_module_name.py
__name__ has value __main__
```

You can notice here that `__name__` has a different value depending on how you run your module. You can use this to make your module behave differently depending on how it is run.

For illustration, in the following example, we import our module the usual way, but if it is run in standalone, it will call one of its functions with a specific value. Suppose you have the following content in file `mymodule.py`.

```
def sq(x):
    return x ** 2

if __name__ == '__main__':
    print('Eight squared is equal to', sq(8))
```

You now can use this module the usual way from Python when you import it. Note that it does not print our `sq(8)`.

```
>>> import mymodule as mm
>>> mm.sq(12)
144
```

And when you run it standalone, it will simply print `sq(8)`.

```
$ python mymodule.py
Eight squared is equal to 64
```

If you want to see what's inside a module, you can use the built-in `dir()` function. There are other things in there as well (such as `__name__`), but you can also see your self-written `sq` function in there. In general, `dir()` shows all functions, classes, and variables in the module.

```
>>> import mymodule as mm
>>> dir(mm)
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'sq']
```

You can also use `dir()` without an argument to see the names in the current scope. In the following code, we look at all names in the current scope, then we add a variable `x`, take a look whether it's listed in the current scope, delete it with `del`, and check that it's gone.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__']
>>> x = 8
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__',
'x']
>>> del x
>>> dir()
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__']
```

There are also packages in Python. A package is essentially a collection of modules, organized hierarchically in folders. Each folder contains a `__init__.py` file, telling Python that this folder is special because it contains Python modules.

In terms of using packages, you can import them the same way as modules. We do not further go into writing packages, as most likely it will suffice if you write module(s) for this course.

## 6.12 Iterators in Python

### 6.12.1 Overview

- Iterators pervade and unify Python. They are a powerful programming concept because they let you do **data streaming**. The basic idea is that you do not need to load the whole dataset into memory (which might be impossible if your dataset is very large). Instead, you process your data in small chunks one by one, thus potentially **saving a lot of memory**. Furthermore, you can **create long, data-driven pipelines** as you can feed generators as inputs to other generators. (A generator is a way of creating an iterator in Python.)

- **The reason why we discuss iterators is that we need them when discussing the gensim package** in Chapter 20 starting on page 205. The gensim package is very popular for doing text analytics in Python. There's also a nice discussion about iterators written by gensim's author Radim Rehurek, upon which this section is partially built.

- Iterators can be powerful, but at the beginning they might sometimes seem a bit confusing. I'll try to explain them clearly here in this book.

- Be careful: When you google around, there are lots of explanations out there about this concept, but **some of them have their terminology mixed up**, which will only confuse you, especially if you are trying to learn about iterators for the first time. If in doubt, refer to the official Python documentation or just follow along here in this book. Explanations from Python's glossary:
  - iterator
  - iterable
  - generator
  - generator iterator
  - generator expression
  - sequence
- To avoid confusion, I suggest **not to use "iterable" as a verb**, e.g. don't say: "Is this object iterable?" Instead, **"iterable" should be used as a noun**, e.g. say: "This object is an iterable." The reason is that an "iterable" has a very specific meaning in Python as we will see in Section 6.12.4 starting on page 87.
- itertools is a popular module containing a number of iterator building blocks.
- We focus on Python 3 in this course. However, in case you need to deal with iterators and would like to write code that works on both Python 2 and 3, you can use the six library. The **six library** helps with writing code that is **compatible with both Python 2.5+ and Python 3**. For example, it has an `iteritems` method that returns an iterator over a dictionary's items. In the code below, `iteritems(d)` creates an iterator that works with both Python 2 and 3. In Python 2, `d.iteritems()` creates an iterator. In Python 3, on the other hand, `d.items()` creates an iterable. In the following example, all three pieces of code below have the same output.

```
d = {'Jack': '#4098', 'Guido': '#4127'}   # Create a dictionary.
# Works in Python 2.
[key + ' has ' + value for key, value in d.iteritems()]
# Works in Python 3.
[key + ' has ' + value for key, value in d.items()]
# Following code works in Python 2 & 3 using the six library.
from six import iteritems
[key + ' has ' + value for key, value in iteritems(d)]
```

### 6.12.2 Summary of Iterables and Iterators

- Iterators and iterables were introduced into Python to provide a general, efficient, and flexible way of **iterating over collections of items**.

- An **iterable** is an object that is capable of returning its members one at a time. For example, lists, tuples, dictionaries, sets, and file objects (e.g. when opening a file using the built-in `open()` function) are iterables.
- An **iterator** is used to actually perform the iteration. Often the iterations are performed over an iterable corresponding to the iterator, but it is also possible to have an iterator without a corresponding iterable. In any case, you need an iterator because **an iterator is the object that gives you the actual stream of data** one at a time.
- For example, iterables can be used in a for loop, or as arguments for many built-in functions such as `zip()` or `map()`.

  ```
  >>> myiterable_a = [1, 2, 3]       # An iterable.
  >>> myiterable_b = [4, 5, 6]       # Another iterable.
  >>> for i in myiterable_a:
  ...     print(i)
  ...
  1
  2
  3
  >>> it = zip(myiterable_a, myiterable_b)
  >>> list(it)
  [(1, 4), (2, 5), (3, 6)]
  >>> it = map(lambda x: 2 * x, myiterable_a)
  >>> list(it)
  [2, 4, 6]
  ```

  Side note: `zip()` or `map()` return iterators, so to view their contents in the examples above, we convert their outputs back to lists.
- Every iterator is also an iterable. When calling the `iter()` function on an iterator, the iterator will always return itself. It thus satisfies the requirement of an iterable, i.e. that calling the `iter()` function returns an iterator.
- A difference between an iterable and an iterator is that (unless the iterable is an iterator) you cannot call `next()` on an iterable, while you can always call `next()` on an iterator. For example, `next([1, 2])` will not work, because the list `[1, 2]` is not an iterator.
- You can pass through an iterator only once. Afterwards, it is exhausted.
- If you use an iterable in the for loop (e.g. `for i in [1, 3, 5]:`), you can run the same for loop again and and again. On the other hand, if you use an iterator, the for loop will only work once, because the iterator is exhausted afterwards.
- An **iterator can be created** in several ways:

1. Calling the `iter()` function on an iterable, e.g. `iter([8, 10, 12])` or `iter(range(8))`. See Section 6.12.3 starting on page 86
2. Calling the `iter()` function on an iterator. (The iterator will just return itself.)
3. Define and then instantiate a custom class that has `__iter__()` and `__next__()` methods, see Section 6.12.4 starting on page 87. The `__iter__(self)` method must return `self`, i.e. it returns an iterator. This is necessary because every iterator has to be an iterable, and an iterable requires that `iter()` (or equivalently `__iter__()`) returns an iterator. The `__next__()` method returns the next item of the iterator.
4. Define and then call a generator function, see Section 6.12.5 starting on page 91.
5. Use a generator expression, see Section 6.12.6 starting on page 93.

- An **iterable can be created** by defining and then instantiating a class that has an `__iter__()` method. This `__iter__()` method must return an iterator. See Section 6.12.4 starting on page 87 and Section 6.12.7 starting on page 95.

### 6.12.3 Iterators

- An **iterator** is an object that represents a stream of data. You can move from one item to the next by calling the `next()` function. Once there are no further items produced by the iterator, a `StopIteration` exception is raised; the iterator is exhausted, i.e. empty. This shows that **you can pass through an iterator once only**. In other words, an iterator is good for one pass over the set of values. If you wanted to cycle through several times, you would have to go back to the beginning and create another iterator from the list.

- Let's create an iterator from a list to illustrate some basic concepts.

```
>>> it = iter([8, 10, 12])   # Create iterator from list.
>>> next(it)
8
>>> next(it)
10
>>> next(it)
12
>>> next(it)                  # Iterator is exhausted (empty).
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- Strictly speaking, each iterator has a `__next__()` method, which is implicitly called when using the built-in `next()` function. The example above could have equivalently been written as follows

```
it = iter([8, 10, 12])   # Create iterator from list.
it.__next__()
it.__next__()
it.__next__()
it.__next__()
```

- Likewise, the built-in `iter()` function actually calls the object's `__iter__()` method. The example above could have equivalently been written as follows.

```
it = [8, 10, 12].__iter__()
next(it)
next(it)
next(it)
next(it)
```

- Instead of basing an iterator on a list, you can use many other objects in Python to generate iterators. We will dwell on this in the following Section 6.12.4.

### 6.12.4 Iterables

- So what does a list have to do with an iterator? A list is not an iterator (you cannot call `next()` on it), but it can be used to generate an iterator. It is therefore by convention called an "iterable." An **iterable** is an object capable of returning its members one at a time. You can think of an **iterable as something that you can use to create an iterator** by calling the `iter()` function on the iterable. Specifically, an iterable is an object that manages a single pass over a sequence type (such as list, str, tuple) or some non-sequence type (such as dict, file objects, or a class you define with an `__iter__()` method).

```
from collections.abc import Iterable, Iterator
l = [8, 10, 12]            # A list is an iterable, but not an iterator.
isinstance(l, Iterable)    # True. A list is an iterable.
isinstance(l, Iterator)    # False. A list is NOT an iterator.
it = iter(l)               # Create iterator.
isinstance(it, Iterable)   # True. An iterator is an iterable.
isinstance(it, Iterator)   # True. An iterator is an iterator (duh).
```

- As we have just seen, **any iterator is also an iterable**. Why is that the case? It is because when you apply the `iter()` function on an iterator, the iterator will always return itself. So in this sense, it can generate an iterator (i.e. itself) and thus by definition is an iterable.

- Recap: All iterators are also iterables. But not all iterables are iterators. **You can think of the set of all iterables being a strict superset of the set of all iterators.** In symbols:

$$\{\text{iterators}\} \subsetneq \{\text{iterables}\}$$

- In this context, it is important to understand how the for loop works internally. Let's consider for illustration the following example, which simply prints the numbers 8, 10, and 12 based on `my_list` (which, as we know, is an iterable).

  ```
  my_list = [8, 10, 12]
  for x in my_list:
      print(x)
  ```

  What happens behind the scenes (done automatically by the for loop) is that **an iterator is created** by calling `iter(my_list)`. This iterator has a `__next__()` method, and each time this method is called, its return value is assigned to `x` and the loop is executed. This goes on until a `StopIteration` exception is raised when calling `__next__()`, in which case the iterator is empty and the loop exits.

- **So the for loop in Python internally uses an iterator!**

- As we have noted before, an iterator is also an iterable, so let's see **what happens when we use an iterator in a for loop**. The first time, everything works as expected. The for loop calls `iter(my_it)`, which returns itself (i.e. `my_it`) and then uses `next()` to get each successive item and assigns it to `x`. However, the next time we run the loop, nothing happens! The reason is simply that the iterator is exhausted, so there are no items left in the iterator. If we want to get some output, we need to get new iterator (e.g. run `my_it=iter(my_list)` again).

  ```
  >>> my_list = [8, 10, 12]
  >>> my_it = iter(my_list)   # Create iterator.
  >>> for x in my_it:         # Works as expected.
  ...     print(x)
  ...
  8
  10
  12
  >>> for x in my_it:         # No output! Iterator is exhausted!
  ...     print(x)
  ...
  >>> my_it = iter(my_list)   # Need to get a new iterator ...
  >>> for x in my_it:         # ... to get some output.
  ...     print(x)
  ...
  8
  10
  12
  ```

- You can have a similar example when you want to convert the iterator back to a list. The second time, the iterator is exhausted and you get an empty list.

```
>>> my_list = [8, 10, 12]
>>> my_it = iter(my_list)    # Create iterator.
>>> list(my_it)              # Works as expected.
[8, 10, 12]
>>> list(my_it)              # Empty list because iterator is exhausted!
[]
>>> list(my_it)              # ... still empty!
[]
```

- Let's create an iterator manually. As mentioned, an iterator is what we ultimately care about because **an iterator represents a stream of data**. For illustration, we implement an iterator that is similar to `range()` (which is an immutable sequence type and as such it is an iterable), with the difference that our implementation creates an iterator directly (unlike `range()`, our implementation does not create an iterable). As mentioned in Section 6.12.3, it needs a `__iter__()` method, which simply returns itself. It also needs a `__next__()` method, which yields the next item (or throws a `StopIteration` exception at the end). The first thing we do therefore is to define a class with those two methods (and an additional `__init__()` initializer method to initialize some variables when the class is instantiated).

```
class my_iterator:
    def __init__(self, n):  # Initialize some variables.
        self.i = 0
        self.n = n

    def __iter__(self):     # An iterator needs to return itself here.
        return self

    def __next__(self):     # This method yields the next item.
        if self.i < self.n:
            self.i += 1
            return self.i - 1
        else:
            raise StopIteration() # Raise StopIteration exception when exhausted
```

Second, we need to instantiate this class and assign it to a variable, which will contain the iterator.

```
it = my_iterator(3) # Create new iterator by instantiating the class.
```

Third, we can check that the iterator works as expected.

```
>>> next(it)
0
>>> next(it)
```

```
1
>>> next(it)
2
>>> next(it)                            # Iterator is exhausted.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 15, in __next__
StopIteration
```

Also, as expected, the iterator is exhausted after finishing a given for loop.

```
>>> it = my_iterator(3)      # Create iterator.
>>> for x in it:
...     print(x)
...
0
1
2
>>> for x in it:             # Iterator is exhausted, no output.
...     print(x)
...
>>>
```

- How could we **turn this iterator into an iterable** similar to range()? Remember that, using its __iter__() method, an iterable returns an iterator. That's exactly what we are going to do in the following class definition.

```
class my_iterable:
    def __init__(self, n):
        self.n = n

    def __iter__(self):             # Return an iterator here.
        return my_iterator(self.n)
```

Then all we have to do is instantiate this class to actually create an iterable.

```
itrble = my_iterable(3)   # Create an iterable.
```

And we can check that it works as expected.

```
>>> it = iter(itrble)     # Create an iterator from the iterable.
>>> next(it)
0
>>> next(it)
1
```

```
>>> next(it)
2
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 14, in __next__
StopIteration
```

Because `itrble` is an iterable (and not an iterator), we can now for example use it several times in a for loop. Each time another for loop starts, a new iterator is created for that loop, because the for loop internally calls `iter(itrble)`.

```
>>> for x in itrble:    # Creates new iterator for this for loop.
...     print(x)
...
0
1
2
>>> for x in itrble:    # Creates another iterator for this for loop.
...     print(x)
...
0
1
2
```

### 6.12.5 Generator Functions (Generators)

- **A generator is a tool for creating an iterator with relative ease.** Instead of manually creating an iterator like we have done in Section 6.12.4 starting on page 87, it is often easier to create an iterator from a generator.
- Anything that can be done with generators can be done with class-based iterators (e.g. as done in `my_iterator` in Section 6.12.4). However, what makes generators so compact is that they create the `__iter__()` and `__next__()` methods automatically.
- Strictly speaking, **when talking about a generator we usually mean a *generator function*, which creates a *generator iterator* object when called**.
- Sometimes people also use the term *generator* to refer to a *generator iterator*, which depends on the context and admittedly is a bit of a sloppy terminology. If you want to use sloppy terminology, I think it is better to call a generator iterator simply "iterator" (because that's what it really is).
- Here's a simple example illustrating a generator function and a generator iterator. We begin by defining a generator function. This generator function determines how the iterator (which will later be derived from it) will behave.

```
def gf():       # Define a "generator function".
    yield 1
    yield 2
    yield 3
```

Next we create a generator iterator from the generator function by calling the generator function. The key thing is that **you are pausing each time there is a** `yield` **expression until the following call to** `next()` **is made**. As you can see, at the end, the generator iterator is exhausted as calling `next()` results in an error, so **you can only consume the values of a generator iterator once**. This is exactly the same behavior as we have seen for iterators in Section 6.12.3 starting on page 86.

```
>>> gi = gf()       # Create a "generator iterator."
>>> next(gi)        # Retrieve next item from the iterator.
1
>>> next(gi)        # Retrieve next item from the iterator.
2
>>> next(gi)        # Retrieve next item from the iterator.
3
>>> next(gi)         # Raises StopIteration since iterator is empty.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> gi = gf()       # Again create a "generator iterator".
>>> next(gi)        # Retrieve next item from the iterator.
1
>>> next(gi)        # Retrieve next item from the iterator.
2
>>> next(gi)        # Retrieve next item from the iterator.
3
>>> next(gi)         # Raises StopIteration since iterator is empty.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- For illustration, you can also use a generator function in a for loop. As usual, the for loop internally takes care of creating the iterator and calling `next()`. We have briefly discussed this in Section 6.12.4 starting on page 87. As you can see, you can go through several for loops because, based on the generator function, a new generator iterator is created each time a new for loop is encountered. In other words, each time a generator function is called, it creates a new iterator.

```
>>> for value in gf():
...     print(value)
```

```
...
1
2
3
>>> for value in gf():
...     print(value)
...
1
2
3
```

- Here is another way to write and use a generator function. What we do here is put the `yield` expression into a for loop. Everything else, e.g. the creation of the generator iterator and its behavior, is the same as before.

```
>>> def gf():                # Define the generator function.
...     for x in range(3):
...         yield x ** 2
...
>>> gi = gf()                # Create a generator iterator.
>>> next(gi)
0
>>> next(gi)
1
>>> next(gi)
4
>>> next(gi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> for value in gf():       # Using the generator function in a for loop.
...     print(value)
...
0
1
4
```

### 6.12.6 Generator Expressions

- **Generator expressions** are shorthand ways to create generator iterators.

- A generator expression is equivalent to creating an anonymous generator function and calling it (thus creating a generator iterator).

- **A generator expression is conceptually related to a list comprehension**, but the key difference is that it conserves memory by creating the items one at a time, and only when they are needed.
- Here's an example, where we create a generator iterator gi using a generator expression:

```
>>> gi = (i ** 2 for i in range(10)) # No calculations done at this stage!
>>> sum(gi)       # Calculations done here using the generator iterator.
285
>>> sum(gi)       # Generator iterator is empty.
0
```

- Often you would use generator expressions and the resulting calculations in one line of code. In this case, you can omit one set of brackets (this is just syntactic sugar to save some typing).

```
sum((i ** 2 for i in range(10)))
sum(i ** 2 for i in range(10))     # Equivalent. Syntactic sugar.
```

- Keep in mind that when you put a generator iterator into a for loop, it will be exhausted (i.e. empty) after the first run. This is the same behavior we saw with iterators in Section 6.12.4 starting on page 87.

```
>>> gi = (word + '!' for word in 'I love NLP'.split())
>>> for val in gi:
...     print(val)
...
I!
love!
NLP!
>>> for val in gi:
...     print(val)
...
>>>
```

- On the other hand, if you put a generator expression into the for loop, it will supply a new generator iterator each time a new for loop is run.

```
>>> for val in (word + '!' for word in 'I love NLP'.split()):
...     print(val)
...
I!
love!
NLP!
```

```
>>> for val in (word + '!' for word in 'I love NLP'.split()):
...     print(val)
...
I!
love!
NLP!
```

### 6.12.7 Creating Iterables the Easy Way Using Generators

- This section is the culmination of our previous efforts in understanding iterators. Specifically, **in this section we describe the probably easiest way to write a custom iterable**.
- Remember the iterable `my_iterable` defined earlier in Section 6.12.4 starting on page 87. Simplifying this example, we are going to **use a generator function to create an iterable**.
- What we did in Section 6.12.4 was to define a class with an `__iter__()` method that returned an iterator. Specifically, it returned `my_iterator` which was our custom-built iterator defined separately before. Instead of going through all the trouble and defining a custom iterator, we can simply create one using a generator function.
- Specifically, we write `__iter__()` as a generator function, which (as we know from Section 6.12.5) always returns a generator iterator.

```
class my_iterable2:
    def __init__(self, n):
        self.n = n

    def __iter__(self):         # Generator function.
        for x in range(self.n):
            yield x ** 2
```

- We have thus written a class for an iterable, because a call to the `__iter__()` method creates a new iterator (by using a generator function).
- We can now instantiate this class in order to create an iterable.

```
itrble2 = my_iterable2(3)
```

- This iterable can be used in the usual way. Each time a new for loop starts, the iterable `itrble2` is used to create a new iterator for that loop.

```
>>> for x in itrble2:
...     print(x)
...
```

```
0
1
4
>>> for x in itrble2:
...     print(x)
...
0
1
4
```

- Equivalently, instead of using a generator function, we can also **use a generator expression to create an iterable**. Specifically, the `__iter__()` method simply returns the call to an generator expression, for which we know from Section 6.12.6 starting on page 93 that this is an iterator.

```
class my_iterable3:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        # Return an iterator created from a generator expression.
        return (x ** 2 for x in range(self.n))
```

## 6.13 Functional Programming

While Python is often used for object-oriented programming (see Section 6.6 starting on page 58), it is also possible to write your programs using functional programming. The basic idea is to decompose a problem into a set of functions.

### 6.13.1 map

The `map` function goes through an iterable and applies a given function to each element. For example:

```
>>> mylist = ['I', 'love', 'NLP']
>>> result = map(len, mylist)      # Apply then ‘len‘ function to ‘mylist‘.
>>> for x in result:
...     print(x)
...
1
4
3
```

Another example illustrates a function with two arguments:

```
>>> def f(x, y):
...     return x + y
...
>>> result = map(f, ['hello', 'world'], ['foo', 'bar'])
>>> for x in result:
...     print(x)
...
hellofoo
worldbar
```

As discussed in Section G.4 starting on page 258, there is an `apply()` method for DataFrames in Pandas, which works in a similar fashion as `map`.

### 6.13.2 Anonymous Functions

Sometimes you just need to use a function once, in which case it doesn't make a lot of sense to give it a name. Thus, Python has anonymous functions, i.e. functions without a name. Consider the following example, where we define a function and just use it once. It is a "waste" of time and space to define this function and assign it to a name if we just plan to use the function once.

```
>>> def myfunction(z):
...     return z + 5
...
>>> result = map(myfunction, [10, 11, 12])
>>> for x in result:     # Show the result of the calculation.
...     print(x)
...
15
16
17
```

Instead, we can omit giving a name to the function, and instead just use an anonymous function, indicated by `lambda`. The following code produces the same output as the previous code, but does it in a much more concise way.

```
>>> result = map(lambda z: z + 5, [10, 11, 12])
>>> for x in result:     # Show the result of the calculation.
...     print(x)
...
15
16
17
```

Note that it is possible to give an anonymous function a name, if you wish. For example, the function named `f` works exactly the same way as the function `myfunction` defined above:

```
>>> f = lambda z: z + 5   # Give a name ‘f‘ to the anonmyous function.
>>> f(10)
15
>>> myfunction(10)
15
```

### 6.13.3 filter

Sometimes you have an iterable and would like to remove some elements. For example, you have stock price returns and would only like to keep the ones that are 1% or higher.

```
>>> stock_returns = [0.03, -0.02, 0.003, 0.02]
>>> def f(x):
...     if x >= 0.01:        # Check if 1% or higher.
...         return True      # Values with ‘True‘ will be kept by ‘filter‘.
...     else:
...         return False     # Values with ‘False‘ will be removed by ‘filter‘.
...
>>> result = filter(f, stock_returns)
>>> for x in result:
...     print(x)
...
0.03
0.02
```

Another example that uses an anonymous function to keep only the odd numbers. Remember that % refers to the modulo operation, i.e. it finds the remainder after division. For example, `x % 2` divides `x` by two and returns the remainder. Below we check whether the remainder is not equal to zero, which means that the number `x` is odd. In this case, the anonymous function returns a `True` value, telling `filter` to keep the corresponding element of `sequence`.

```
>>> sequence = [2, 3, 4, 5, 6, 7]
>>> result = filter(lambda x: x % 2 != 0, sequence)  # Keep odd numbers.
>>> for x in result:
...     print(x)
...
3
5
7
```

### 6.13.4 reduce

Sometimes you would like to summarize an iterable (e.g. a list) with a single value. For example, you would like to sum all elements in a list, or you would like to compute the maximum. Oftentimes, there are already functions available in Python that do these things, e.g. `sum()` and `max()`. Still, sometimes you would like have customized behavior, in which case `reduce` comes in handy.

In the following examples, we add all elements in the list and we also find the maximum. The purpose of these examples is to illustrate how `reduce()` works. The purpose is not to show how to do summation or finding the maximum, as these could be done more easily using `sum()` and `max()` as discussed above.

The way `reduce()` works is that it has a function of two arguments, and iteratively applies this function to an iterable (e.g. a list). In the summation example below, it would first compute $5+1=6$, and then it would compute $6+8=14$, which is the end result. Or in the case of the maximum, it would first compute $\max\{5,1\}=5$, and then it would compute $\max\{5,8\}=8$.

```
>>> import functools
>>> mylist = [5, 1, 8]
>>> def myaddition(x, y):
...     return x + y
...
>>> def mymax(x, y):
...     if x > y:
...         return x
...     else:
...         return y
...
>>> functools.reduce(myaddition, mylist)
14
>>> functools.reduce(mymax, mylist)
8
```

Equivalently, the code above could have been written more succinctly using anonymous functions:

```
>>> import functools
>>> mylist = [5, 1, 8]
>>> functools.reduce(lambda x, y: x + y, mylist)
14
>>> functools.reduce(lambda x, y: x if x > y else y, mylist)
8
```

One could also use the operator module, as it provides some commonly-used functions. For example, `operator.add(x, y)` does the same thing as the expression `x + y`.

```
>>> from functools import reduce
>>> from operator import add, mul
>>> mylist = [5, 1, 8]
>>> reduce(add, mylist)     # Addition.
14
>>> reduce(mul, mylist)     # Multiplication.
40
```

# Chapter 7

# Basic Concepts of Text Analytics and NLP

## 7.1 Difference Between Text Analytics and NLP

Natural Language Processing (NLP) is a subfield of artificial intelligence and linguistics that focuses on enabling computers to understand, interpret, and generate human language. It encompasses a wide range of tasks, such as part-of-speech tagging, syntactic parsing, named entity recognition, coreference resolution, and machine translation. NLP aims to bridge the gap between human communication and computer understanding by developing techniques and algorithms that can process and analyze language data effectively. This field has seen significant advancements with the advent of deep learning and neural networks, leading to increased accuracy and efficiency in various language-related tasks.

Text analytics, often considered a subset of NLP, is the process of analyzing and extracting insights from unstructured text data. It utilizes various NLP techniques and algorithms to perform tasks such as sentiment analysis, keyword extraction, document clustering, text summarization, and information retrieval. While text analytics focuses specifically on textual data, NLP covers a broader spectrum of language processing tasks, some of which go beyond the scope of text analytics. For example, NLP includes tasks such as speech recognition, which involves converting spoken language into written text; dialogue systems and chatbots, which enable computers to engage in conversation with humans; and natural language generation, where computers generate human-like text or narratives. In summary, text analytics represents a specific application area within the larger field of NLP, and both fields are intrinsically linked through their shared goal of understanding and processing human language.

## 7.2 Limitations and Potential Pitfalls

It is important to keep in mind that **text analytics is not an “exact” science** in the sense that there is always room for interpretation. For example, if you use two different analysis methods on the same text document, you will receive two different answers.

Furthermore, even the preprocessing steps can make a difference in the answers you obtain. Therefore it is always important to keep in mind that **working with textual data is messy**. It therefore of utmost importance to:

1. Obtain data of the highest possible quality.
2. Clean and preprocess the data with great care and attention to detail.
3. Conduct your analysis in a sensible way and use the right tools for the job.

Even if you have a valuable input and are careless with preprocessing and analyzing your textual data, you will end up with "garbage out." If you are looking for a quick fix and are cutting corners, the analysis performed will more often than not be of low value added. Therefore **a thorough understanding of your textual data is of crucial importance.**

Another important item to keep in mind is the number of documents you have and how big each document is. For example, if you have downloaded millions of websites, you typically have relatively short documents, but you have many of them. On the other hand, if you are analyzing books, you might have only two lengthy books (i.e. documents), but each one of them is very long. Depending on which case you are in, you might want to use different approaches to analyzing text, so it is important to keep in mind that there is no "magic" text analysis method that works well in all cases. **Depending on the text data at hand, you should carefully select the appropriate text analytics methods.**

Related to this point is that it's always important to **have enough text data** so that whatever algorithm you use actually works reliably. Even the best algorithm won't help you if you don't have enough data or if your data is of low quality.

## 7.3 "Terms" or "Words?"

In text analytics and NLP, you sometimes hear people say "term" instead of "word." Which one should you use? If you are just dealing with words, then by all means say "words."

On the other hand, sometimes you might for example be dealing with bigrams as in the example on page 170 in Chapter 13. In this case, "I live" would be a single term (consisting of two words) and you would feed this term into your further analysis. In this case, to avoid misunderstandings, it is better to talk about "terms" rather than "words."

Another example is if you have transformed your document by singular-value decomposition (SVD), so the new (transformed) entries do not really correspond to single words any more. Instead, they correspond to linear combinations of other words. In this case again it makes more sense to say "terms" when referring to those transformed entries.

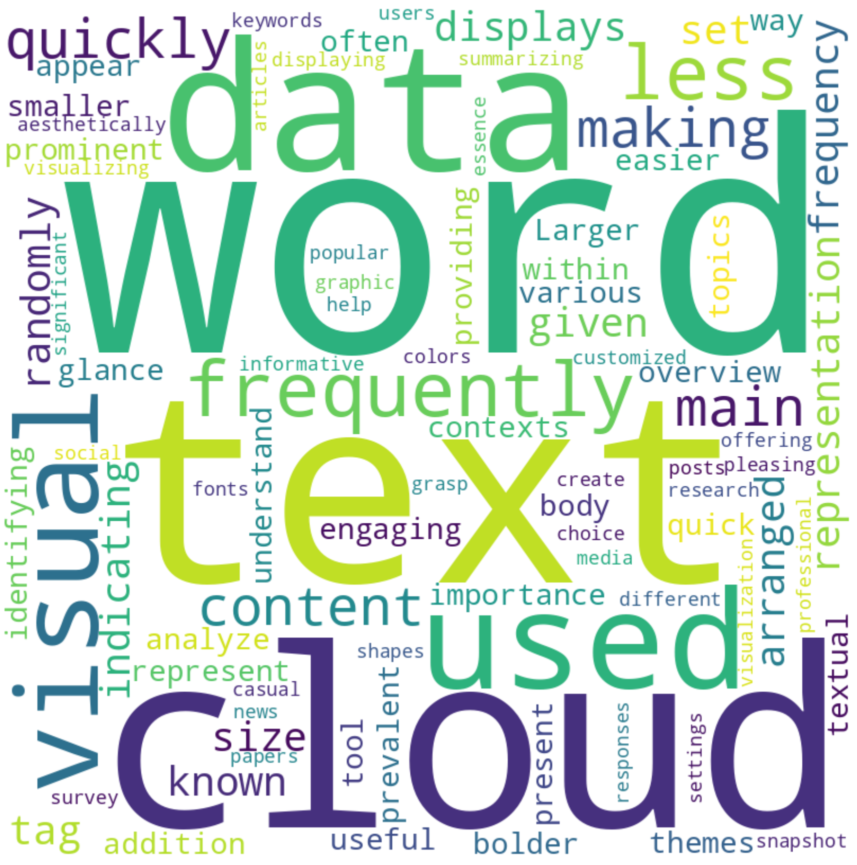
# Chapter 8

# Word Clouds

A word cloud, also known as a tag cloud or text cloud, is a visual representation of text data that displays the most frequently used words in a given text or set of texts. The words are arranged randomly, with the size of each word indicating its frequency or importance in the text. Larger and bolder words represent those that appear more frequently, while smaller and less prominent words are used less often. Word clouds are a useful tool for quickly identifying the main themes or topics within a body of text, making it easier to analyze and understand the content at a glance.

In addition to providing a quick overview of the most prevalent words in a text, word clouds can also be an engaging way to present textual data. They can be used in various contexts, such as visualizing survey responses, summarizing social media posts, or displaying the main keywords in news articles or research papers. Word clouds can be customized with different colors, shapes, and fonts to create an aesthetically pleasing and informative graphic. By offering a visual snapshot of the most significant words in a text, word clouds help users to quickly grasp the essence of the content, making them a popular choice for data visualization in both professional and casual settings.

For example, the previous two paragraphs would be represented by the following word cloud:



In the following code snippet, we will demonstrate how to generate a word cloud in Python using the "wordcloud" library. This library provides a simple and effective way to create word clouds from a given text. We will start by importing the necessary modules, then preprocess the text by removing stop words, and finally create and visualize the word cloud using the `WordCloud` class and `matplotlib` library for displaying the result. The code below can be found in the file `code-wordcloud.py` on the course website.

```
# This script shows how to generate a word cloud (also known as tag
# cloud).
from wordcloud import WordCloud, STOPWORDS
import matplotlib.pyplot as plt

# Prepare the text data with repeated words.
text = '''Python Python wordcloud wordcloud example code
programming AI assistant tutorial library text data visualization
data data data visualization visualization visualization AI AI AI
assistant assistant'''

# Set up the word cloud configuration. Words that show up more often
# will appear larger in the word cloud.
wc = \
    WordCloud(
        width=800,
        height=800,
        background_color='white',
        stopwords=STOPWORDS,  # Remove stop words.
        min_font_size=10).\
    generate(text)
# Using ‘words_‘ attribute to take a look at the words and their
# frequencies.
wc.words_
# Display the word cloud using matplotlib.
plt.figure(figsize=(8, 8), facecolor=None)
plt.imshow(
    wc,              # Matplotlib calls ‘wc.__array__()‘ for plotting.
    interpolation='bilinear')
plt.axis('off')
plt.tight_layout(pad=0)
plt.show()
```

# Chapter 9

# Sentiment Analysis

Sentiment in general refers to opinions, feelings, and emotions, and in a financial context to optimism and pessimism in financial markets. As such, sentiment is of subjective nature, but it is nonetheless important because it can drive human decision making in finance. A key example is that positive sentiment might put upward pressure on prices, resulting in stock returns that are at least temporarily higher.

## 9.1 TextBlob Sentiment Score

A straightforward way to do sentiment analysis is through the TextBlob package, which is built using the NLTK and Pattern libraries. It aims to simplify text processing such as sentiment analysis, but also many other tasks such as part-of-speech tagging, noun phrase extraction, classification, and translation. TextBlob can calculate the polarity score, which captures how positive the text in a string is. It ranges from minus one (very negative) to one (very positive). It can also calculate the subjectivity score, which captures how subjective the string is. This score ranges from zero (very objective) to one (very subjective). By default, TextBlob uses the dictionary-based sentiment calculation from the Pattern library:

```
from textblob import TextBlob
s = TextBlob('I love NLP.').sentiment
s.polarity      # How positive the text is.
s.subjectivity  # How subjective the text is.
```

TextBlob can alternatively use machine learning (a Naive Bayes analyzer from NLTK) that is trained on a dataset of movie reviews:

```
from textblob import TextBlob
from textblob.sentiments import NaiveBayesAnalyzer
b = \
    TextBlob(
        'I love NLP.',
        analyzer=NaiveBayesAnalyzer())
b.sentiment
```

## 9.2 NLTK Vader

NLTK has Vader, which is very good at analyzing sentiment on social media. Vader has been integrated into NLTK based on code from vaderSentiment.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer
sid = SentimentIntensityAnalyzer()
sid.polarity_scores('I love NLP.')
```

## 9.3 afinn

Afinn does sentiment analysis using a dictionary. It was developed for analyzing the language found in microblogs such as Twitter. The dictionary itself is also available separately, see the Section 9.4 starting on page 108. The following example is taken from the Afinn documentation:

```
>>> from afinn import Afinn
>>> a = Afinn()                 # Instantiate the ‘Afinn‘ class.
>>> a.score('I love NLP.')
3.0
```

## 9.4 Dictionaries

There are various ways one can measure sentiment. The easiest way is to look at dictionaries (or “lexicons”) containing certain vocabulary that is associated with positive or negative emotions. All you do is to count how many positive or negative words appear in a document relative to its length.

Furthermore you can divide the positive or negative score by the document length, i.e. the number of words contained in the document. The rationale is that if there are a lot of positive words in a document, it could be because the document has a positive content, or because it is simply a long document with lots of words in it (some of which will be positive). To eliminate this effect, you can divide by the total number of words in the document.

Here is a list of popular dictionaries:

1. SentiWordNet is a lexical resource for opinion mining based on WordNet
2. NRC Emotion Lexicon from Saif Mohammad and Peter Turney
3. Sentiment lexicon from Bing Liu and collaborators
4. AFINN lexicon of Finn Arup Nielsen
5. qdapDictionaries has various dictionaries, e.g. labMT for a happiness score (polarity)

6. Loughran-McDonald sentiment lexicon
7. Affective Norms for English Words (ANEW)

The R community has been quite active in collecting some of these dictionaries and making them easily available, for example in the following R packages: tidytext, textdata, SentimentAnalysis, syuzhet, and quanteda.dictionaries. You can install these packages in R using `install.packages`, e.g. type `install.packages('tidytext')` to install the tidytext R package (you do not need to be root for that).

To import these dictionaries into Python, it's probably easiest to write the dictionary to a CSV file in R, then import this CSV file into Python. The following example imports the Loughran-McDonald sentiment dictionary into Python:

```
# In R:
library('tidytext')
DF = get_sentiments('loughran')
write.csv(DF, 'sentiment.csv', row.names = FALSE)
# In Python:
import pandas as pd
DF = pd.read_csv('sentiment.csv')
```

## 9.5 Machine Learning

Another way is based on supervised learning, which is a form of machine learning, see Section 11.2 starting on page 117. Either you read a few text documents manually and assign them labels such as “positive” or “negative,” or alternatively you look e.g. for market reactions (i.e. whether the market interprets something as “positive” or “negative”). You then feed these labels into a machine learning classifier of your choice (e.g. naive Bayes or others) and let it learn which text it should regard as “positive” or “negative.” After a few observations for training, your machine learning classifier should then (hopefully) be able to reliably label hitherto unseen text documents as either “positive” or “negative.”

The advantage of the dictionary approach is that it is really simple and therefore easy to understand and easy to interpret. People will not come charging at you and accuse you of using a “black box.” Instead, even people who have no idea about text analytics and NLP can easily understand what you are doing. Using a dictionary approach can therefore be useful if your application requires convincing someone that what you are doing makes sense.

The advantage of the machine learning approach is that (depending on the quality of the training data) it might be more accurate. Another advantage is that it can deal better with different contexts. For example, the vocabulary used in financial applications might be quite different from applications in psychology. As a consequence, if you use dictionaries that are not specifically made for the context you're dealing with, you might misclassify some texts. Using the machine learning appraoch lets you avoid many of these issues, assuming that you can get hold of a good and suitable training set.

# Chapter 10

# Named Entity Recognition (NER)

The purpose of named entity recognition (NER) can be used to identify important named entities in a text. Examples include people, places, and organizations. It can also deal with identifying other things such as dates. **NER can help answer: Who? What? When? Where?**

Besides its obvious purpose of extracting named entities, NER can also be used to increase the precision of other tasks such as classification. For example, instead of using a vector that represents the document (e.g. a BoW vector) as an input to the classifier, you can augment this vector by adding the extracted named entities occurring in the document. A very simple application for illustration: If you are trading the U.S. and UK markets, you could add an indicator variable that equals one if any entity related to New York (e.g. "New York," "NYSE," "Nasdaq," etc.) shows up in the document, and zero if any entity related to London (e.g. "London," "London Stock Exchange," "LIFFE," etc.) shows up. This concept can be generalized by adding a categorical variable with values corresponding to all possible named entities in the corpus, or various indicator variables, each representing the occurrence of a separate named entity.

Before looking at named entity extraction, we typically run **part-of-speech tagging, also called POS tagging, PoS tagging, or POST**. (We have briefly mentioned POST in Chapter 13 starting on page 161.) POST allows for a better NER. What does POST do specifically? It tries to figure out whether a word is a proper noun, pronoun, adjective, verb, or another part of speech, based on the English grammar. For example, "NNP" in NLTK means "proper noun, singular."

Popular packages for performing NER:

- NLTK
- SpaCy
- Stanza, which is the recommended way to use Stanford CoreNLP (a Java library) in Python.
- Polyglot, especially if you are dealing with languages other than English.

## 10.1 NLTK and NER

Here is an illustration how it works in NLTK. The code can be found in the file `code-nltk-NER.py` on the course website. You will see when running the code that the MTR or the HK Space Museum are identified as organizations while Andy Hayler is identified as a person. The code can be found in the file `code-nltk-NER.py` on the course website.

```
# This file shows how to do named entity recognition (NER) using NLTK.
import nltk
sentence = '''In Hong Kong, I like to ride the MTR to visit the HK Space Museum
              and some restaurants rated well by Andy Hayler.'''
to_sentence = nltk.word_tokenize(sentence)
ta_sentence = nltk.pos_tag(to_sentence) # Tag the sentence for parts of speech.
ta_sentence[:3]
# Returns sentence as a tree, with named entities such as PERSON,
# ORGANIZATION, etc.
nltk.ne_chunk(ta_sentence)
# Extract stems of the tree with 'NE' tags, i.e. we're getting all the
# named entities.
ner_sentence = nltk.ne_chunk(ta_sentence, binary=True) # Tags named entities as "NE
for chunk in ner_sentence:
    if hasattr(chunk, 'label') and chunk.label() == 'NE':
        print(chunk)

# If you have more than one sentences, you can adapte the above
# workflow as follows:
article = '''I like riding the MTR.
             And sometimes I visit the HK Space Museum.
             Andy Hayler rates restaurants'''
sentences = nltk.sent_tokenize(article)
token_sentences = [nltk.word_tokenize(sent) for sent in sentences]
pos_sentences = [nltk.pos_tag(sent) for sent in token_sentences]
# chunked_sentences = [nltk.ne_chunk(sent, binary=True) for sent in pos_sentences]
chunked_sentences = nltk.ne_chunk_sents(pos_sentences, binary=True) # Using a gener
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label') and chunk.label() == 'NE':
            print(chunk)

# We can also plot a pie chart showing how often each named entity
# type appears in the text. For counting we use as usual a
# defaultdict.
from collections import defaultdict
import matplotlib.pyplot as plt
chunked_sentences = nltk.ne_chunk_sents(pos_sentences)
ner_categories = defaultdict(int)
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label'):
            ner_categories[chunk.label()] += 1

labels = list(ner_categories.keys())
values = [ner_categories.get(l) for l in labels]
plt.pie(
    values,
    labels=labels,
    autopct='%1.1f%%',           # Add percentages to chart.
    startangle=140)              # Rotate initial start angle.
plt.show()
```

## 10.2 SpaCy and NER

Yet another alternative to NLTK NER is SpaCy, a relatively young but fast-growing and popular Python package. In general, SpaCy is good for creating NLP pipelines to generate models and corpora. It has has extra libraries and tools, e.g. in the following link you can visualize named entities in your text online using displaCy.

- spaCy.io
- displaCy

You can do many things with spaCy, and here we are going to use it for NER. It has different entity types and often labels entities differently than NLTK. It can also deal better with informal corpora, e.g. tweets and chat messages. You can find the following code in the file `code-spacy-NER.py` on the course website.

```
# This script shows how to do named entity recognition (NER) with
# spaCy. The example is taken from the spaCy documentation.
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp('Apple is looking at buying U.K. startup for $1 billion')
doc.ents                        # Document attribute called `ents`.
# Investigate labels of first entity using `label_` attribute.
print(doc.ents[0], doc.ents[0].label_)
print(doc.ents[0].text, doc.ents[0].label_) # Same effect.
# Iterate over `doc.ents` and print out the text, start and end positions,
# and labels.
for ent in doc.ents:
    print(ent.text, ent.start_char, ent.end_char, ent.label_)
```

## 10.3 Polyglot and NER

There is also the Polyglot package. Its main advantage is that it can deal with many different languages. The following code can be found in `code-polyglot-NER.py` on the course website.

```
# This script shows named entity recognition (NER) with polyglot.
from polyglot.text import Text
text = '''Cervantes es un conocido escritor español,
          cuya obra más famosa, "Don Quijote de la Mancha",
          tiene lugar en la región de Castilla-La Mancha.'''
ptext = Text(text) # No need to specify language here; recognized automatically.
ptext.entities # `entities` attribute; see a list of chunks (with label).
for ent in ptext.entities:      # Print each of the entities found.
    print(ent)
type(ent)                       # Print the type of the (last) entity.
ent.tag                         # Tag of (last) entity.
'Castilla' in ent               # Check is 'Castilla' is in the (last) entity.
# List comprehension to get tuples. First tuple element is the entity
# tag, the second is the full string of the entity text (separate by
# space).
[(ent.tag, '␣'.join(ent)) for ent in ptext.entities]
# The `pos_tags` attribute queries all the tagged words.
for word, tag in ptext.pos_tags:
    print(word, tag)
```

# Chapter 11

# AI and Machine Learning for NLP

This chapter provides a conceptual overview of artificial intelligence (AI) and machine learning and illustrates them with introductory code examples. AI and machine learning have many subfields, with perhaps the two most well-known subfields being supervised learning and unsupervised learning. What distinguishes those two approaches is whether the machine learning model/algorithm can find things out by itself (unsupervised learning) or whether it needs initial guidance (supervised learning), e.g. in the form of labeled data. In addition to supervised and unsupervised learning, we also discuss semi-supervised learning, neural networks and deep learning, and reinforcement learning.

From the outset, it is important to clarify some terminology. There is often confusion when people talk about artificial intelligence (AI), machine learning (ML), and deep learning. The best way to think about it is that AI is the most general topic, followed by ML, followed by deep learning. In other words, **deep learning is a subfield of ML, and ML is a subfield of AI**. Using subset notation:

$$\text{Deep learning} \subseteq \text{Machine Learning} \subseteq \text{Artificial Intelligence}$$

Another way of jokingly thinking about the difference between AI and machine learning is as follows: If it's written in Python, it's probably machine learning. If it's written in PowerPoint, it's probably AI.

Machine learning is a cornerstone for NLP and text analytics. The usual application of ML is that once you have converted your text into a numeric representation (e.g. BoW, tf-idf, doc2vec, or others, see Chapter 14 starting on page 173), you need to *do* something with it. One thing would be to classify documents (e.g. whether they are "positive" or "negative" along some dimension such as the outlook of a stock) or you could cluster similar documents together (e.g. if you're trying to figure out the different topics discussed). Most or all of these tasks require machine learning.

## 11.1 A Brief History of NLP and AI

There are basically two different historical approaches to NLP, which happen to parallel the developments in artificial intelligence (AI). In fact, **one could consider NLP a**

**subfield of AI.**

Before going into the details, it is important to keep in mind that there were **two major "AI winters,"** the first one approximately **1974–80** and the second one approximately **1987–93**. Daniel Crevier writes, "it is a chain reaction that begins with pessimism in the AI community, followed by pessimism in the press, followed by a severe cutback in funding, followed by the end of serious research." Basically it is a pattern of excessive hype and the following bust that has occurred in other emerging technologies as well, e.g. the railway mania or the dot-com bubble. The general point to be taken away is (a) not to get carried away if everyone around you is euphoric and (b) not to think all is lost when everyone around you is pessimistic. Of course this is easier said than done, but it is still worth bearing in mind.

In any case, in the early days of NLP and AI (roughly the time before the mid-1980s), people tried to develop a set of logical rules (if-then rules) that tried to help understand what was going on. Often these rules were **hand-written**. Logic programming languages such as **Prolog, ASP, and Datalog** were very popular in those areas and were used to formalize the logical rules. Another very popular programming language for AI was (and still is to some extent) the venerable **Lisp**, because it supports the implementation of software that computes with symbols very well. The strand of AI that uses **logical theory** is often also termed "**symbolic AI**."

It turned out that a problem with this approach to NLP and AI is that the rules often do not generalize (e.g. if you are originally analyzing the French language and want to instead analyze the English language, you need a whole new set of rules), are often fragile to small changes (e.g. if you give it a sentence it has never seen before it might not work even close to as expected), and often require a lot of understanding of the specific problem domain (e.g. what are the quirks of the French language vs. the quirks of the English language).

After the second AI winter another approach gained traction, so-called "**sub-symbolic AI**", whose most important subfield, "**connectionism**," models cognitive processes using neural networks, inspired by the way human brains operate. The basic idea is to take a statistical approach (i.e. machine learning) and let the data speak. Suppose for example you want to write a software that translates French to English. With symbolic-AI, you would write an elaborate set of rules to translate each sentence. On the other hand, sub-symbolic AI you would for example take a large set of documents that are issued simultaneously in both languages for the European Union, and run machine learning models over both of these sets without significant human intervention. It turns out that this second approach using sub-symbolic AI works much better and requires less work, as long as you have a sufficiently large set of data to learn from.

Nowadays when people talk about AI, they are almost always referring to machine learning. However, it is important to keep in mind that AI is strictly speaking a much broader field, even though most applications nowadays are from machine learning. An interesting area of AI that combines ideas from both symbolic and sub-symbolic AI is **inductive logic programming (ILP)**, which is a subfield of machine learning that uses logic programming to represent hypotheses, examples, and background knowledge.

## 11.2 Supervised Learning

The basic idea is to give the model/algorithm a few examples it can learn from, and afterwards the model can perform the task by itself. For example, you have two kinds of text documents, one is a movie review, the other is a cooking recipe. You go through a few, say, 100 documents, read them, and tell the model for each of these 100 documents what it is (a review or a recipe). The model thus has 100 examples where it knows the input (the document text) and the desired output (whether it is a review or a recipe). Based on these examples (which are called a **training set**), the model then tries to “understand” what the relation between the input and the output is. This is called **model fitting**. Once it has learned a reasonably accurate representation thereof, it can then classify documents by itself into reviews or recipes. For example, if you give the model a new text it has never seen before, the model then can tell you (with a hopefully high degree of certainty) whether this document is a movie review or a cooking recipe.

Model fitting can include both parameter estimation and variable selection. **Parameter estimation** means finding out the model parameters that guarantee the best model fit. **Variable selection** means finding out which predictor variables actually matter and which ones don’t (and “select” them, i.e. keep, only those variables that matter). For example, the word “and” might not contain a lot of information in a text document. If we think about this word as a predictor variable for whether the document is about a movie review or a cooking recipe, it might actually be a good idea to remove this word from the text document because it just introduces unwanted noise.

The main idea of supervised learning is that it needs a few examples to learn from (i.e. the training dataset). So it needs some form of initial guidance, which may or may not involve human input. For example, if you want to know whether a tweet was written in a happy mood, you might have to read a few tweets yourself to determine which ones seem to be happy and which ones are not. On the other hand, if you want to identify tweets that predict a steep downward move in the stock market, you simply select a few relevant tweets that were posted before a few down moves from the past and use those as a training set. No human interaction is required here (in the sense that nobody has to manually go through these tweets and manually read them).

How large should the training set be? There is no fixed rule for that, since it always depends on the specific application at hand and the specific algorithm you use. For example, naïve Bayes typically needs relatively few examples to learn from (in some cases as few as 100), while deep learning is often said to require hundreds of thousands or even millions of observations in the training set. In any case, a general rule is that **the larger the training set, the better the model fit**.

A key problem we might face is **overfitting**, which means that the model fits the training dataset very well, but as soon as you provide a new data point the model has not seen before (in the training set), it fails miserably. We discuss this problem further in Section 11.11 starting on page 147, but to get an intuitive understanding let’s consider the following example. Suppose for example that you run a company whose sales are a linear function of GDP plus some small random fluctuations (noise). You could fit two models, a linear model and a polynomial model. The polynomial model would have a much better fit on the training set than the linear model. But if you look at a new sample that was not used for model fitting before, the linear model would perform better. The polynomial model in this case suffers from overfitting, i.e. it seems to work great on the training sample, but not on any new data points.

A simple but very popular and powerful model from supervised learning is the **naive Bayes model**. It answers the question of: Given a particular piece of data, how likely is a particular outcome? For illustration we use the multinomial naive Bayes classifier from sklearn, `MultinomialNB`. This classifier normally requires integer feature counts, so it is suitable for bag-of-words. (You could alternatively use the `NaiveBayesClassifier` from the NLTK library which uses a different implementation of the naive Bayes model.) For tf-idf `MultinomialNB` might also work (even though it is not specifically tailored for floats), but for tf-idf you might also want to try support vector machines (SVM) or linear models. In any case, here we focus on integer inputs and multinomial naive Bayes. The code below can be found in the file `code-sklearn-naive-bayes.py` on the course website.

```
# This script illustrates how to use the naive Bayes classifier from
# the SciKit-Learn library.
import numpy as np
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics     # To evaluate model performance.

# Load the 20 Newsgroups dataset with 3 categories for demo purposes.
categories = ['comp.graphics', 'sci.space', 'rec.sport.baseball']
data = \
    fetch_20newsgroups(
        subset='all',
        categories=categories,
        remove=('headers', 'footers', 'quotes'))

X_text = data.data              # Raw text data.
y = data.target # Labels (0: comp.graphics, 1: rec.sport.baseball, 2: sci.space)

# Convert text data into word counts (discrete, non-negative
# features). Here we limit ourselves to 100 features for simplicity.
vectorizer = CountVectorizer(max_features=100)
X = vectorizer.fit_transform(X_text) # Shape: (n_samples, n_features)

# Split up the data into training and testing.
count_train, count_test, y_train, y_test = \
    train_test_split(
        X, # Each observation: 100 independent variables (word frequencies).
        y, # Each observation: One dependent variable.
        test_size=0.20,         # 20% of dataset used for testing.
        random_state=42) # Seed for random shuffling of data for train & test.
```

```
# Instantiate the ‘MultinomialNB‘ class. Here "multinomial" refers to
# the features, i.e. you can have a multinomial distribution such as
# word counts (non-negative integers) as inputs to the model. If the
# result is suboptimal, you can change the ‘alpha‘ parameter.
nb = MultinomialNB()            # MultinomialNB(alpha=0.5)
# Determine internal parameters based on dataset. Pass the training
# count vectors first, and training labels second.
nb.fit(count_train, y_train)
# Get the class labels.
nb.classes_
# Log-probability of features occurring, given a class (in this case
# the first class, i.e. the class having 0-th index).
nb.feature_log_prob_[0]
# Make predictions of the label for test data.
pred = nb.predict(count_test)

# Test the accuracy of the predictions.
metrics.accuracy_score(y_test, pred)
# The confusion matrix shows correct and incorrect labels. On the
# output array, the MAIN DIAGONAL shows the true scores, i.e. the
# labels that have been CORRECTLY predicted. The TRUE LABELS
# correspond to the ROWS, while the PREDICTED LABELS correspond to the
# COLUMNS. For example, assuming you're looking at movie reviews, if
# you have in the second row corresponding to the label "sci-fi" and
# in the first column corresponding to the label "action", then the
# entry in the second row, first column shows you the number of movie
# reviews that are truly about sci-fi movies, but were incorrectly
# classified by the naive Bayes model as reviews of action movies.
metrics.\
    confusion_matrix(
        y_test,                 # Actual labels.
        pred,                   # Labels predicted by naive Bayes model.
        labels=[0, 1, 2])       # Reorder the resulting matrix.
```

Below is a list of some of the most popular supervised learning models and methods. These approaches are widely used for various tasks across domains. The models highlighted in bold are particularly practical and effective for natural language processing (NLP), making them strong candidates for tasks such as text classification, sentiment analysis, and more. Section E.11 starting on page 237 contains a list of useful Python packages.

- **Naive Bayes** is a very robust supervised learning method. It does not always give the very best results, but it is usually surprisingly close to the best available method. For text classification it is often a good idea to start with naive Bayes as it easy to use and usually gives good results in a wide variety of scenarios.
- Ordinary least squares (OLS) regression
- **Logistic regression**

- **Support vector machines (SVMs)**
- $k$-nearest neighbors ($k$-NN or KNN, approximate nearest neighbor)
- Classification and regression trees (CART)
- Ensemble methods: Bayesian averaging, bagging (e.g. **random forests**), boosting (e.g. AdaBoost, **LightGBM**, **XGBoost**). LightGBM and XGBoost are very popular these days, e.g. on Kaggle competitions. For example, LightGBM has won the M5 Forecasting Competition. It is important to keep in mind that XGBoost often does not work well if the training set is too small relative to the number of predictor variables. In NLP applications, this means for example that the number of observations in your training set should be larger than the number of terms. To achieve this, you can for example, remove terms that occur in fewer than 0.5% of all documents in order to reduce the number of terms. Furthermore, to be on the safe side, you should always compare the performance of XGBoost to other methods such as naive Bayes or deep learning.

## 11.3 Unsupervised Learning

Unsupervised learning tries to extract information from the data completely without any human supervision or input. A classic example is **clustering**, where you have a set of objects and you're trying to group similar objects together. For example, you could have a set of newspaper articles, and the clustering algorithm would try to put newspaper articles together into groups such that similar articles are in the same group. The key thing is that the algorithm looks at the data and tries to identify the groups by itself. Importantly, you do not specify these groups in advance, so for example you do not tell the algorithm that you want to group the articles into politics vs. business. The groups found by the clustering algorithm should in the end hopefully somehow be interpretable, but the interpretation might be different than what you would expect from the outset. For example, maybe instead of politics vs. business, the algorithm might find it more convincing to group the articles into what could be interpreted as art vs. science.

Another key are of unsupervised learning is **anomaly detection**. The basic idea is to find observations that do not conform to expected patterns in a dataset. An example from NLP is to find errors in text. An example from finance is to identify bank fraud.

We illustrate unsupervised learning using k-means clustering, which is one of the most popular unsupervised learning algorithms. The basic idea is that the algorithm partitions the data into several groups (so-called "clusters") without using any training data. Each cluster should contain similar data. However, one has to specify the number of clusters before running the algorithm. It is therefore sometimes useful to experiment with different numbers of groups. The code below can be found in the file `code-sklearn-k-means-clustering.py` on the course website.

```python
# This script illustrates how to use k-means clustering in
# SciKit-Learn.
from sklearn.cluster import KMeans
```

```
import numpy as np
# We create a numpy array with some data. Each row corresponds to an
# observation.
X = \
    np.array(
        [[1, 2],
         [1, 4],
         [1, 0],
         [10, 2],
         [10, 4],
         [10, 0]])
# Perform k-means clustering of the data into two clusters.
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
# Show the labels, i.e. for each observation indicate the group
# membership.
kmeans.labels_
# Ifyou have two new observations (i.e. previously unseen by the
# algorithm), to which group would they be assigned?
kmeans.predict(
    [[0, 0],
     [12, 3]])
# Show the centers of the clusters as determined by k-means.
kmeans.cluster_centers_
```

Below is a list of some of the most popular unsupervised learning models and methods. These approaches are fundamental for discovering patterns and structures in unlabeled data. The methods highlighted in bold are particularly useful and practical for natural language processing (NLP) tasks, such as topic modeling, clustering, and dimensionality reduction, making them essential tools for understanding and organizing text data. Section E.11 starting on page 237 contains a list of useful Python packages.

- Apriori algorithm
- Clustering algorithms: Centroid-based (e.g. ***k*-means clustering**), connectivity-based (e.g. **hierarchical clustering**), density-based (e.g. DBSCAN or OPTICS), probabilistic (e.g. Gaussian mixture models), neural nets/deep learning (e.g. self-organizing map). Note that often it is useful to perform dimensionality reduction before running a clustering algorithm!
- Dimensionality reduction: Singular value decomposition (**SVD**, see Section 19.1 starting on page 197), principal component analysis (PCA)
- Topic modeling: **Latent Dirichlet allocation (LDA)**
- Independent component analysis (ICA)

## 11.4 Semi-Supervised Learning

The main idea of semi-supervised learning is to leverage both labeled and unlabeled data for training machine learning models. This approach helps improve the model's performance, especially when labeled data is scarce or expensive to obtain. Semi-supervised learning combines elements of supervised learning (using labeled data) and unsupervised learning (using unlabeled data), aiming to achieve better results than using either method alone. This is typically done by exploiting the underlying structure or patterns in the unlabeled data to refine the model's predictions and enhance its generalization capabilities. This is typically done in one of two ways:

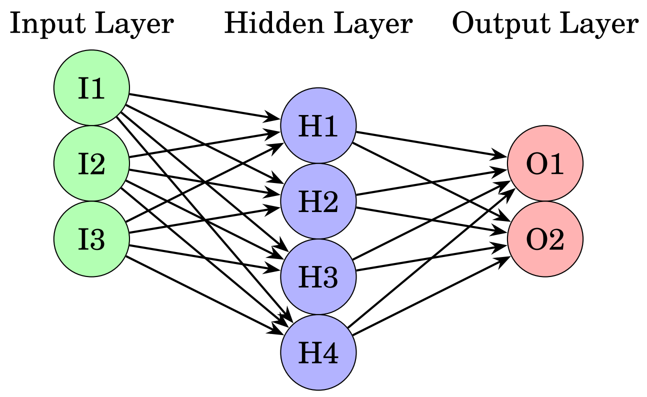
1. **Self-training (or self-labeling)**: In this method, the model is initially trained with the labeled data using a supervised learning approach. Then, the model is used to predict labels for the unlabeled data. The most confident predictions are treated as "pseudo-labels" and added to the training set along with the original labeled data. The model is then retrained using this expanded dataset, which can help improve its performance.

2. **Consistency regularization**: This approach encourages the model to produce consistent predictions for similar or related data points, even if they are unlabeled. This is based on the assumption that similar inputs should have similar outputs. By enforcing this consistency, the model can learn a more robust representation of the data, which can lead to better generalization and improved performance on unseen data.

Both of these methods aim to incorporate the information present in the unlabeled data to improve the model's predictions. By doing so, semi-supervised learning can often achieve better results than using only supervised or unsupervised learning techniques.

## 11.5 Neural Networks and Deep Learning

Neural Networks consist of interconnected neurons organized into layers. Typically, a neural network comprises an input layer that receives external data, one or more hidden layers that process the data, and an output layer that produces the final result. Neural networks and deep learning models can be trained using supervised, semi-supervised, or unsupervised learning methods.



Traditionally, researchers focused on a specific class of neural networks known as **shallow neural networks**, which typically have one or two layers between the input and output layers. In contrast, **deep neural networks**, often associated with **deep learning**, feature multiple hidden layers between the input and output layers, allowing for more complex data representations and processing. Section E.12 starting on page 237 contains a list of useful Python packages for deep learning.

A **feedforward neural network (FFN)** is a type of artificial neural network where connections between the nodes do not form a cycle, making information flow in one direction: from the input layer through hidden layers to the output layer. Each layer consists of a set of neurons, and each neuron processes input by applying a weighted sum followed by an activation function to introduce non-linearity. Depending on the number of hidden layers in an FFN it can be classified as either a shallow or deep learning model. FFNs are widely used for tasks like classification, regression, and function approximation, as they are effective in mapping complex input-output relationships. In the following code example we illustrate sentiment classification using an FFN. The code below can be found in the file `code-keras-imdb-sentiment.py` on the course website.

```python
# Introduction to feedforward neural networks (FFNs) in NLP. Here we
# analyze the sentiment of movie reviews from the IMDb database.

from tensorflow import keras
from tensorflow.keras import layers

# Number of words to consider as features.
vocab_size = 10000
# Load the data, keeping only the top 10,000 (`vocab_size`) most
# frequent words.
(train_data, train_labels), (test_data, test_labels) = \
    keras.datasets.imdb.load_data(
        num_words=vocab_size)
# Inspect the first review.
print("First review (as word indices):", train_data[0])
print("First review label (0=negative, 1=positive):", train_labels[0])
```

```
# Get the word index file, mapping words to indices.
word_index = keras.datasets.imdb.get_word_index()
# Reverse the word index to obtain a mapping from indices to words.
reverse_word_index = {value: key for key, value in word_index.items()}
# Decode the first review. Unknown words are marked with `???` and we
# need to shift the index by `-3` to get the correct words.
print(' '.join([reverse_word_index.get(i - 3, '???')
                for i in train_data[0]]))
```

```
# Set the maximum review length.
maxlen = 500
# Pad sequences with zeros (post-padding) to ensure they all have the
# same length.
train_data = \
    keras.preprocessing.sequence.pad_sequences(
        train_data,
        value=0,
        padding='post',
        maxlen=maxlen)
test_data = \
    keras.preprocessing.sequence.pad_sequences(
        test_data,
        value=0,
        padding='post',
        maxlen=maxlen)
print("Shape of train data:", train_data.shape)
print("Shape of test data:", test_data.shape)
```

```
# Building the feedforward neural network (FFN): We build a simple FFN
# with an embedding layer, followed by flattening and two dense
# layers.
#
# Embedding Layer: Transforms each word index into a dense vector of
# fixed size (32 in this case, but could also be a smaller number such
# as 16 for example). This layer learns word embeddings during
# training.
#
# Global Average Pooling 1D layer: Reduces the sequence of embeddings
# into a single vector by averaging, effectively summarizing the
# information. Results in a shorter vector (compared to flattening),
# but may lose some information.
```

```
#
# Dense layer with 16 units and ReLU activation introduces
# non-linearity.
#
# Dropout layer with a dropout probability of 0.5. This means that
# during training, 50% of the neurons in the layer will be randomly
# set to zero (i.e. "dropped out") in each training iteration. This is
# an example of regularization. The primary purpose of dropout is to
# prevent overfitting, which occurs when a model learns noise from the
# training data rather than the underlying patterns. By randomly
# dropping neurons, the model is forced to learn more robust features
# that are less dependent on specific neurons.
#
# Alternative A (commented out): Flatten layer: Flattens the 2D
# embeddings into a 1D vector to feed into the dense layers. Might
# result in a very long vector.
#
# Alternative B (commented out): For better sequence modeling, switch
# to a recurrent layer (LSTM or GRU). This will capture long-range
# dependencies and typically yields good results on sequence data.
#
# Alternative C (commented out): A small CNN with global max pooling
# can be quite effective for sentiment classification.
#
# Output Dense layer with 1 unit and sigmoid activation outputs a
# probability between 0 and 1.
model = \
    keras.Sequential([
        layers.Embedding(input_dim=vocab_size, output_dim=32),
        #
        layers.GlobalAveragePooling1D(),
        layers.Dense(16, activation='relu'),
        layers.Dropout(0.5),    # Optional: Reduce overfitting.
        #
        # # Alternative A:
        # layers.Flatten()
        # layers.Dense(16, activation='relu'),
        #
        # # Alternative B:
        # layers.LSTM(16),   # RNN LSTM.
        #
        # # Alternative C:
        # layers.Conv1D(            # CNN.
        #     filters=32,
        #     kernel_size=3,
        #     activation='relu'),
        # layers.GlobalMaxPooling1D(),
        #
        layers.Dense(1, activation='sigmoid')])
model.summary()
# We'll compile the model with the binary cross-entropy loss function
# and the Adam optimizer. We'll also track accuracy as a metric.
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy'])



# Before training, it's good practice to set aside a validation set to
# monitor the model's performance on unseen data during training. Here
# we are setting aside 10,000 samples for validation, and use the rest
# for training. (Note that you could alternatively use
# `train_test_split` from `sklearn` for splitting the data.)
x_val = train_data[:10000]       # For validation.
y_val = train_labels[:10000]
partial_x_train = train_data[10000:] # For training.
partial_y_train = train_labels[10000:]
# Train the model.
history = \
    model.fit(
        partial_x_train,
        partial_y_train,
        epochs=20,
        batch_size=256,
        validation_data=(x_val, y_val),
        verbose=1)
model.summary()



# Visualizing the training process can help in understanding the
# model's performance and detecting overfitting.
import matplotlib.pyplot as plt
history_dict = history.history
loss = history_dict['loss']
val_loss = history_dict['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'bo', label='Training␣loss')  # 'bo' = blue dots.
plt.plot(epochs, val_loss, 'b', label='Validation␣loss')  # 'b' = solid blue line.
plt.title('Training␣and␣Validation␣Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
# After training, evaluate the model on the test dataset to see how
# well it generalizes.
results = model.evaluate(test_data, test_labels, verbose=2)
print("Test␣Loss:", results[0])      # Lower loss is better.
print("Test␣Accuracy:", results[1])  # Higher accuracy is better.
```

```
# Predict the probabilities for the test set.
predictions = model.predict(test_data)
# Since the model outputs probabilities, convert them to binary labels
# (0 or 1).
predicted_labels = (predictions > 0.5).astype(int).flatten()
# Compare predicted labels with actual labels.
for i in range(10):
    print(f'Review␣{i+1}:')
    print('Predicted␣Sentiment:␣' + \
          ('Positive' if predicted_labels[i] == 1 else 'Negative'))
    print('Actual␣Sentiment:␣' + \
          ('Positive' if test_labels[i] == 1 else 'Negative'))
    print('-' * 50)
```

```
from sklearn.metrics import confusion_matrix, classification_report
# Confusion Matrix: Shows the number of true positives, true
# negatives, false positives, and false negatives.
cm = \
    confusion_matrix(
        test_labels,
        predicted_labels)
print("Confusion␣Matrix:")
print(cm)
# Classification Report: Provides precision, recall, f1-score, and
# support for each class.
report = \
    classification_report(
        test_labels,
        predicted_labels,
        target_names=['Negative', 'Positive'])
print("\nClassification␣Report:")
print(report)
```

```
# You might want to predict sentiments for new reviews that are not
# part of the dataset. To do this, you need to preprocess the new text
# in the same way as the training data. Here's how:
from nltk.tokenize import word_tokenize
reviews = [
    "I absolutely loved this movie! The performances were outstanding and the plot 
    "This was the worst film I have ever seen. Completely boring and a waste of tim
    "An average movie with some good moments but overall not impressive."
]
for review in reviews:
    print(review)
    tokens = word_tokenize(review.lower()) # Better tokenizer.
    # '2' is typically the index for unknown words
    # (i.e. out-of-vocabulary word, OOV). And we need to shift the
    # index by '+3' to get the correct words, because the first few
    # indices are reserved for special tokens.
    sequence = \
        [word_index.get(word, 2 - 3) + 3 \
         for word in tokens]
    # Prepend <START> Token: Adds '1' at the beginning to signify the
    # start of the sequence.
    sequence = [1] + sequence
    processed_review = \
        keras.preprocessing.sequence.pad_sequences(
            [sequence],         # Must be a list of iterables.
            value=0,
            padding='post',
            maxlen=maxlen)
    prediction = model.predict(processed_review, verbose=0)[0][0]
    predicted_label = 'Positive' if prediction > 0.5 else 'Negative'
    print(f'Predicted Sentiment: {predicted_label} (Probability: {prediction:.4f})'
    print('-' * 80)
```

The well-known **word2vec** model can be considered a specialized form of an FFN. It is an example of a shallow neural network, as it consists of only two layers: a hidden layer for learning word embeddings and an output layer for predicting context words. Although word2vec is not considered a deep learning model, it is extensively used as a pre-training step for various deep learning architectures in NLP tasks. The word embeddings generated by word2vec can serve as input features for deep learning models, enhancing their performance on a range of NLP applications. For more information on word2vec, refer to Section 20.8 on page 216.

Deep learning has revolutionized NLP by introducing architectures capable of capturing complex patterns and representations in data. Among the most prominent deep learning models are recurrent neural networks (**RNNs**) and their variants, such as long short-term memory (**LSTM**) networks and gated recurrent units (**GRUs**), which excel at modeling sequential data. Convolutional neural networks (**CNNs**), initially designed for image processing, have also been adapted for NLP tasks to capture local dependencies in text. More recently, **transformer-based models**, such as bidirectional encoder representations from transformers (**BERT**) and generative pre-trained transformers (**GPT**), have dominated the field by leveraging attention mechanisms to process sequences in parallel and capture contextual information effectively. These models have pushed the boundaries of NLP by achieving state-of-the-art results across a wide range of applications, from sentiment analysis to machine translation and text generation. Next, we list some of the most common use cases for various deep learning architectures in natural language processing (NLP).

**Recurrent Neural Networks (RNNs):**

- Sequence-to-sequence tasks like translation (e.g. older models before transformers, such as Google Neural Machine Translation).
- Text generation (e.g. poetry, stories, by predicting sequences of words).
- Named Entity Recognition (NER) in sequential text (e.g. identifying names, locations).
- Sentiment analysis with sequential dependence (e.g. analyzing long reviews by understanding context).

**Convolutional Neural Networks (CNNs):**

- Text classification (e.g. sentiment analysis, spam detection).
- Feature extraction (e.g. extracting hierarchical features from text).
- Document-level categorization (e.g. summarizing key features for categorization).

**Transformers:**

- Language modeling and generation (e.g. GPT, BERT, generating human-like text).
- Machine translation (e.g. Google Translate, MarianMT, replacing RNN-based models).
- Sentiment analysis and text classification (e.g. fine-tuning for specific tasks with state-of-the-art results).
- Text summarization and paraphrasing (e.g. generating concise summaries or paraphrased versions).
- Question answering and conversational AI (e.g. chatbots, virtual assistants generating contextually relevant responses).

When **deciding between RNNs, CNNs, or transformers** for NLP tasks, the choice depends on the nature of the problem and the data. **RNNs** are well-suited for sequential data and tasks requiring context from previous tokens, such as language modeling and time-series prediction, though they can struggle with long-term dependencies. **CNNs** excel in capturing local patterns, such as word collocations, short phrases, or nearby dependencies, and are efficient for tasks like text classification or sentiment analysis, where understanding small, localized parts of the text is crucial. **Transformers**, powered by self-attention mechanisms, are the go-to choice for most modern NLP tasks, including translation, summarization, and question answering, as they effectively capture both local and global dependencies in parallel and scale well to large datasets. Notably, **transformers have largely supplanted both RNNs and CNNs** in many NLP applications due to their superior performance and efficiency, leading to a decline in the usage of these earlier architectures in contemporary NLP research and practice.

## 11.6 Transformers: An Overview

Transformers are a deep neural network architecture that have become the de facto standard in NLP. They differ from earlier sequence models (e.g. RNNs, LSTMs) by using an attention mechanism—in particular, self-attention—to process entire sequences in parallel. Some of the key advantages of transformers include:

- Parallelization: Unlike RNNs, transformers process tokens in parallel, leveraging GPUs more effectively and reducing training time.
- Long-Range Dependencies: Self-attention allows direct connections between any two positions, enabling the model to capture long-term dependencies without degradation.
- Scalability: The architecture scales well with increased data and model size, leading to state-of-the-art performance in various NLP tasks.
- Flexibility: Transformers are versatile and have been adapted for numerous tasks beyond translation, such as text generation, summarization, and even non-NLP tasks like image processing.

Traditional NLP tasks often used recurrent neural networks (RNNs) or long short-term memory (LSTM) networks to handle sequential data. These rely on hidden states that get updated token by token, which can be slow to train and may struggle to capture long-range dependencies. Transformers remove recurrence entirely and instead use attention to allow every token in a sequence to “see” every other token directly, without having to pass information step by step. This approach not only speeds up training (since it is more parallelizable) but also can capture global context more effectively.

The **transformer** model consists of two main parts:

- **Encoder**
- **Decoder**

Each encoder and decoder is composed of several (often identical) **layers**, with each layer performing the following steps:

- Multi-head attention: Self-attention in the encoder, while in the decoder, there is both self-attention and encoder-decoder attention. This mechanism allows the model to weigh the importance of different tokens in a sequence relative to each other. It effectively captures dependencies and relationships between tokens, enabling the model to build contextual representations.
- Feed-forward network (FFN). While attention handles interactions between different positions in the input sequence, the FFN focuses on transforming the features of each individual token. Specifically, the FFN applies a non-linear transformation to each position's representation, enhancing the model's ability to capture complex, non-linear patterns within the data.
- Residual connections + layer normalization around each sub-layer

Before we discuss the details of attention, we should clarify the **input** to the encoder. Suppose we have a sequence of **tokens** (for example, words or subwords):

$$\text{sequence} = [t_1, t_2, \ldots, t_n].$$

Each token $t_i$ is mapped to a trainable embedding, i.e. a learned vector. We then add positional encoding to reflect each token's position in the sequence. (Positional encoding is usually done using sine and cosine functions, e.g. through traditional sinusoidal positional encoding or through rotary positional embeddings (RoPE); we skip the details here for brevity.) Transposing and stacking these "**embedding vectors**" row by row, we obtain a matrix:

$$X \in \mathbb{R}^{n \times d},$$

where

- $n$ is the sequence length, i.e. the number of tokens,
- $d$ is the dimension of each embedding vector.

Hence, each row of $X$ corresponds to the embedding vector (including positional encoding) of token $t_i$. Each column of $X$ corresponds to one of the $d$ embedding features.

Let us focus now on the **encoder**. We start with **single-head attention** (and will discuss multi-head attention next). Within each encoder layer, we compute **self-attention**. This is where the concepts of **query** ($Q$), **key** ($K$), and **value** ($V$) matrices come into play. Formally, we project the input $X$ into $Q$, $K$, and $V$ by multiplying with learned parameter matrices:

$$Q = XW^Q \in \mathbb{R}^{n \times d_{\text{attn}}}, \quad K = XW^K \in \mathbb{R}^{n \times d_{\text{attn}}}, \quad V = XW^V \in \mathbb{R}^{n \times d_{\text{attn}}},$$

where

- $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_{\text{attn}}}$ are trainable weight matrices,

- $d_{\text{attn}}$ is the dimension of the query/key/value representations (often approximately equal to $\frac{d}{\text{number of heads}}$ if we use multiple heads)

Then the **scaled dot-product attention** is computed as:

$$\text{SelfAttention}(Q,K,V) := \text{softmax}\left(\frac{QK^T}{\sqrt{d_{\text{attn}}}}\right)V \in \mathbb{R}^{n\times d_{\text{attn}}}.$$

- The multiplication $QK^T$ captures how similar each query is to each key, producing a "compatibility" score, where $K^T$ is the transpose of $K$.

- Then we apply a softmax across the *row* dimension to get weights that sum up to 1 for each *query* position. The softmax function is defined as

  $$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

  which normalizes the vector $z$ so that all entries are positive and sum to 1.

- The division by $d_{\text{attn}}$ stabilizes the magnitude of the dot-products so they do not become too large for bigger embeddings.

- Finally, we multiply by $V$ to get the output representations, which are weighted sums of the value vectors.

- Intuitively, the query vector (i.e. row vector of $Q$), representing one token, asks: "Which other tokens in the sequence are relevant to me?" The key vector (i.e. row vector of $K$), representing the same or another token, answers: "Am I relevant to the query?" (The product $QK^T$ indicates similarity.) And finally all the value vectors (i.e. row vectors of $V$), representing all tokens, are weighted by the softmax and are the "content" that gets returned if the key vector is a good match for the query vector.

The three projection matrices $W^Q, W^K, W^V$ that map the input $X$ into query ($Q$), key ($K$), and value ($V$) matrices, together with the computation of the scaled dot-product attention, constitutes a single **attention head**.

In practice, rather than using just one attention head, we use multiple heads in parallel, giving us **multi-head attention**. Intuitively, this lets the model learn to attend to different types of relationships or patterns. Each head uses different parameter matrices $W^Q, W^K, W^V$. The final outputs of each head (i.e. the scaled dot-product attentions) are then concatenated and linearly projected:

$$\text{MultiHeadSelfAttention} = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

After self-attention (either single-head or multi-head attention), each token's representation is transformed by a **feed-forward network (FFN)**, often referred to as a multi-layer perceptron (MLP) with a non-linear activation function. A typical example is a two-layer MLP with a rectified linear unit (ReLU) activation in between:

$$\text{FFN}(Z) = \text{ReLU}(ZW_1 + b_1)W_2 + b_2,$$

where

- $W_1, b_1, W_2, b_2$ are trainable parameters and
- $\text{ReLU}(x) = \max(0, x)$ zeroes out negative values and keeps positive values unchanged. It allows the model to learn and represent non-linear relationships in the data.

Both the self-attention output and the feed-forward output are each wrapped with residual connections (i.e. they are added to the original input of that sub-layer) followed by a layer normalization step. Layer normalization involves normalizing the input by its mean and standard deviation, followed by scaling and shifting using learnable parameters. This is referred to as "add & norm," and is done to **stabilize training**.

- Add & norm after self-attention:

$$Y = \text{LayerNorm}(X + \text{MultiHeadSelfAttention}(X))$$

- Add & norm after the FFN:

$$X_{\text{out}} = \text{LayerNorm}(Y + \text{FFN}(Y))$$

The **decoder** is similar to the encoder but has a few key differences:

- **Masked self-attention** (usually in the form of masked multi-head self-attention): For text generation tasks (e.g. translation or chatbot), the decoder needs to ensure it does not "peek" at future tokens it has not generated yet. Hence, it uses masked self-attention: each token can only attend to tokens at earlier positions, preventing leakage of information about upcoming tokens/words. Mathematically, the same attention formula applies as in the encoder, but a mask is applied to the $K^T$ in the dot product so that positions beyond the current token are set to negative infinity (or some large negative value) before applying softmax. The input to masked self-attention of the first decoder layer consists of the embedding vectors of the tokens. For subsequent decoder layers, the input consists of the output of the previous decoder layer.
- **Encoder-decoder attention** (usually in the form of encoder-decoder multi-head attention): In addition to looking at the tokens generated so far, the decoder also attends to the encoder's outputs. After the encoder has processed the entire input sequence into a contextual representation, the **decoder can query those encoder outputs** via attention:

  $$\text{EncoderDecoderAttention}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}}),$$

  where

  - the query matrix $Q_{\text{decoder}}$ is derived from the decoder's previous sub-layer (typically "add & norm" preceded by masked multi-head attention),

  - the key and value matrices $K_{\text{encoder}}, V_{\text{encoder}}$ come from the encoder's final output.

  This allows the decoder to use information from the entire input sequence at each step of its generation process (e.g. to decide which words in the source sentence correspond to words in the translated output).

- **Feed-forward network (FFN)**: Same as in encoder.
- **Add & norm**: Same as in encoder.

Connecting the encoder and decoder, and putting it all together for a sequence-to-sequence task such as translation:

- The encoder processes the input sentence in parallel, generating a contextual embedding for each token
- The decoder then:
  - Uses masked self-attention on the output sequence so far
  - Uses encoder-decoder attention to attend to the encoder's output representations
  - Produces the next token in an auto-regressive manner. "**Auto-regressive**" means the model generates text one token at a time. Each token is predicted based on all previously generated tokens.

This two-part design (encoder-decoder) is particularly useful for tasks such as machine translation, where you first want to understand (encode) the entire input, and then generate (decode) the output in another language.

Since the original transformer model, numerous **variants and improvements** have been proposed, including:

- **BERT** (bidirectional encoder representations from transformers): Focuses on encoder representations for tasks like classification and question answering.
- **GPT** (generative pre-trained transformer): Uses the decoder architecture for generative tasks like text completion.
- **Transformer-XL**: Introduces recurrence to handle longer sequences by maintaining a memory of past computations.
- **T5** (text-to-text transfer transformer): Frames all NLP tasks as text-to-text problems, leveraging both encoder and decoder.

In the context of transformers, **pre-training** refers to the initial phase where a model is trained on a large corpus of unlabeled data to learn general language patterns and representations. During this phase, the model develops a foundational understanding of language without being tailored to any specific task. For instance, models like BERT and GPT are pre-trained using techniques such as masked language modeling or next-token prediction, allowing them to grasp syntax, semantics, and context. This pre-trained model can then be **fine-tuned on smaller, task-specific datasets to adapt its capabilities for particular applications like sentiment analysis or translation**. This approach allows practitioners to adapt a pre-trained model to a specific task using a smaller dataset, which significantly **reduces the time and computational power** required for training.

The following code sets up a text classification pipeline using the Hugging Face **Transformers library** to analyze movie reviews from the **IMDb dataset**. It loads a subset of the dataset, tokenizes the text using the pre-trained **DistilBERT** model, and prepares the data for fine-tuning (i.e. training on a smaller datasets to fix the remaining model weights). A sequence classification model is instantiated to predict sentiment (positive or negative) based on the reviews. The code then trains the model using specified parameters and evaluates its performance, ultimately providing insights into how well it can classify movie sentiments. The code below can be found in the file `code-transformers-imdb.py` on the course website.

```
# This script uses a BERT model from the Hugging Face `transformers`
# library to analyze movie reviews from the IMDb dataset.

from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
    TrainingArguments,
    Trainer)
from datasets import load_dataset

# Use a pretrained BERT model. Here we use 'distilbert-base-uncased'
# which is a smaller model than 'bert-base-uncased'.
bert_model = 'distilbert-base-uncased'

# Load the IMDb dataset.
dataset = load_dataset('imdb')
# Select the first few samples for training and testing. This is done
# purely to speed up model training for demonstration. For production,
# you would skip this step and use the whole dataset.
dataset['train'] = dataset['train'].select(range(1000))
dataset['test'] = dataset['test'].select(range(200))

# Instantiate the tokenizer from `transformers` using the vocabulary
# of a pretrained BERT model.
tokenizer = AutoTokenizer.from_pretrained(bert_model)
# Tokenization function. This is just a wrapper around the `tokenizer`
# with a few parameter values set. For the number of tokens that can
# be generated from the text, `max_length`, we use a very small value
# so that model training will not take too long. For production you
# would use something like `max_length=256`.
```

```
def tokenize_function(example):
    return \
        tokenizer(
            example['text'],
            padding='max_length',
            truncation=True,
            max_length=32)
# Apply tokenization on `dataset` by applying the tokenization
# function to each element of `dataset`.
tokenized_datasets = \
    dataset.\
    map(
        tokenize_function,
        batched=True)

# Set the format of the datasets for PyTorch. We use PyTorch because
# it is tightly integrated with the `transformers` library.
tokenized_datasets = \
    tokenized_datasets.\
    rename_column(
        'label',
        'labels')  # Necessary for comatibility with `Trainer` class.
tokenized_datasets.\
    set_format(
        'torch',
        columns=['input_ids', 'attention_mask', 'labels'])

# Instantiate the pre-trained BERT model for sequence
# classification. There will be a message that some weights were not
# initialized. We will fix this in a later step when we train the
# model.
model = \
    AutoModelForSequenceClassification.\
    from_pretrained(
        bert_model,
        num_labels=2)

# Initialize the `Trainer`, which is a training and eval loop for
# PyTorch, optimized for `transformers`.
trainer = \
    Trainer(
        model=model,
        args=TrainingArguments(
            output_dir='./data-NLP-teaching', # Can be deleted later.
            eval_strategy='epoch',
            per_device_train_batch_size=8,
            per_device_eval_batch_size=8,
            num_train_epochs=2,
            weight_decay=0.01),
        train_dataset=tokenized_datasets['train'],
        eval_dataset=tokenized_datasets['test'])

# Train the model. This step may take several minutes or hours to
# complete, depending on the computing power available.
trainer.train()

# Evaluate the model.
results = trainer.evaluate()
print(results)
```

Next we'll explore how to leverage a pre-trained transformer model for text classification tasks using the **Keras library**. We're using the **AG News dataset**, which consists of news articles sorted into four categories. The process involves tokenizing the text data into a format that neural networks can process, specifically by using a tokenizer from the DistilBERT model. After tokenization, we'll convert these sequences into TensorFlow datasets to handle them in batches during training. We then fine-tune a pre-trained DistilBERT model by adding a classification layer on top to categorize each news article based on its content. This approach demonstrates how **transfer learning** can be applied in natural language processing, where we use a model pre-trained on a vast corpus of text to adapt to a specific classification task with relatively little data. The code below can be found in the file `code-keras-ag_news.py` on the course website.

```
# This code uses Keras to wrap a BERT model from Hugging Face
# `transformers` to analyze the AG News dataset. The outputs from the
# `transformers` model (specifically the `last_hidden_state`) are
# further processed through Keras layers such as `Dropout` and
# `Dense`.

from transformers import AutoTokenizer, TFAutoModel
# Distinguish the Hugging Face `Dataset` class (from the `datasets`
# library) from the TensorFlow `Dataset` class:
from tensorflow.data import Dataset as TF_Dataset
from tensorflow import keras
from datasets import load_dataset

# Load the AG News dataset.
dataset = load_dataset('ag_news')

# Instantiate the tokenizer.
tokenizer = \
    AutoTokenizer.\
    from_pretrained(
        'distilbert-base-uncased')
# Tokenization function. This is just a wrapper around the `tokenizer`
# with a few function arguments set.
```

```
def tokenize_function(example):
    return \
        tokenizer(
            example['text'],
            padding='max_length',
            truncation=True,
            max_length=128)
# Apply tokenization. For development (to minimze model training
# time), you could add `.select(range(1000))` and
# `.select(range(200))` before `.map` to the train and test
# datasets. This would extract only the beginning of the datasets.
tokenized_train = dataset['train'].map(tokenize_function, batched=True)
tokenized_test = dataset['test'].map(tokenize_function, batched=True)

# Convert the tokenized training and test data to TensorFlow
# datasets. `from_tensor_slices` creates a Dataset where each element
# corresponds to one row from the input tensors. Here, it's used to
# create a dataset from tokenized_data. 'input_ids': These are
# typically the tokenized sequences of text where each token has been
# converted into an ID. 'attention_mask': This mask indicates which
# tokens should be attended to (1) or ignored (0), often used in
# models like BERT. The second element is 'label', which contains the
# labels for each example in the dataset. `shuffle(10000)`: This
# operation shuffles the dataset with a buffer size of
# 10,000. Shuffling is important for training to randomize the order
# of the data, helping to prevent overfitting by ensuring the model
# doesn't learn the sequence of data. `batch(16)`: This groups the
# dataset elements into batches of size 16. Batching is essential for
# training as it processes multiple examples at once, which can
# significantly speed up computation on hardware like GPUs.
def convert_to_tf_dataset(tokenized_data):
    return \
        TF_Dataset.from_tensor_slices(
            ({  'input_ids':      tokenized_data['input_ids'],
                'attention_mask': tokenized_data['attention_mask']},
             tokenized_data['label'])).\
        shuffle(10000).\
        batch(16)

train_ds = convert_to_tf_dataset(tokenized_train)
test_ds = convert_to_tf_dataset(tokenized_test)

# Load the pre-trained transformer model, in this case a BERT model
# consisting only of the encoder (i.e. without the decoder).
transformer = \
    TFAutoModel.\
    from_pretrained(
        'distilbert-base-uncased')

# Define model inputs. `keras.Input` is used to define an input layer
# for a Keras model. `shape=(128,)`: This specifies that the input
# will be a 1D tensor (vector) of length 128. This is common in NLP
# tasks where each input is a sequence of tokens. `dtype='int32'`:
# This sets the data type of the input to 32-bit
# integers. `name='input_ids'` or `name='attention_mask'`: This is a
# string identifier for the layer. Naming layers can be useful for
# model architecture visualization, debugging, and when you need to
# reference this layer later in the model definition or when loading
# weights. The `attention_mask` ensures that the model only considers
# the actual content of the input, excluding any padding that was
# added for batch processing, thus optimizing both performance and
# accuracy.
input_ids = \
    keras.Input(
        shape=(128,),
        dtype='int32',
        name='input_ids')
attention_mask = \
    keras.Input(
        shape=(128,),
        dtype='int32',
        name='attention_mask')

# Get transformer outputs.
transformer_outputs = \
    transformer(
        input_ids,
        attention_mask=attention_mask)
# Last hidden state is of shape: (batch_size, sequence_length,
# hidden_size)
hidden_state = transformer_outputs.last_hidden_state
# Take only first element from the `sequence_length` dimension, which
# corresponds to the CLS token ("classification token"). Result is of
# shape: (batch_size, hidden_size)
cls_token = hidden_state[:, 0, :]
# Add a dropout and a dense layer for classification (4 classes in AG
# News).
x = keras.layers.Dropout(0.3)(cls_token)
output = keras.layers.Dense(4, activation='softmax')(x)

# Define the model.
model = \
    keras.Model(
        inputs=[input_ids, attention_mask],
        outputs=output)
# Compile the model.
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=3e-5),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
model.summary()

# Train the model. This may run for a long time.
history = \
    model.fit(
        train_ds,
        validation_data=test_ds,
        epochs=3)

# Evaluate the model
loss, accuracy = model.evaluate(test_ds)
print(f'Test␣Loss:␣{loss}')
print(f'Test␣Accuracy:␣{accuracy}')
```

## 11.7 Evaluating Language Models: The Role of Perplexity

When working with language models (LMs), especially deep learning architectures like *Transformers*, it is important to evaluate how well a model predicts sequences of text. A key metric for this purpose is the **perplexity score**.

Perplexity measures how confidently a language model predicts the next word or token in a sequence. Formally, it is defined as:

$$\text{Perplexity} = e^{\text{Cross-Entropy Loss}}$$

where cross-entropy loss measures how well the model's predicted probabilities match the actual next token. A **lower perplexity** indicates better predictive performance, while a **higher perplexity** suggests that the model is more "surprised" by the data. Perplexity is commonly used to:

- Evaluate pretrained or fine-tuned language models.
- Compare models across datasets.
- Monitor training progress and detect overfitting.

In finance, language models are often fine-tuned on domain-specific corpora such as SEC filings (10-K, 10-Q), earnings call transcripts, Federal Reserve statements, or financial news articles. For example, when fine-tuning a Transformer-based model on SEC reports to generate summaries or detect risk factors, perplexity serves as a useful diagnostic metric to assess whether the model is learning the structure and language of financial documents. However, while perplexity is valuable during training, it does not fully capture text coherence or task-specific success. Real-world financial NLP tasks typically require task-specific evaluation metrics such as accuracy, F1-score, or ROC-AUC.

The following Python code demonstrates how to compute perplexity using a pre-trained GPT-2 model on a sample financial sentence. Note that this example illustrates the computation, but fine-tuning on financial data would lower the perplexity for better performance. For suitable datasets, see Chapter 3 starting on page 23.

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel
import torch
import math

model = GPT2LMHeadModel.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model.eval()
text = ('The Federal Reserve decided to maintain '
    'its accommodative stance to support economic recovery.')
inputs = tokenizer(text, return_tensors='pt')
with torch.no_grad():
    outputs = model(**inputs, labels=inputs['input_ids'])
    loss = outputs.loss

perplexity = math.exp(loss.item())
print(f'Perplexity: {perplexity:.2f}')
```

Since GPT-2 is trained on general web data, perplexity on financial texts may be relatively high. Fine-tuning on domain-specific corpora such as SEC EDGAR filings or FOMC statements (see Chapter 3 starting on page 23) can improve performance. For example:

- SEC EDGAR: Annual and quarterly reports.
- FOMC Minutes: Monetary policy statements.
- Earnings Calls: Available via Seeking Alpha or Capital IQ.
- Financial News: Factiva or Kaggle datasets.
- Twitter Finance Feeds: Sentiment and event-driven data.

For large-scale experiments, also explore the Hugging Face Datasets Hub, which includes datasets suitable for NLP tasks in finance.

## 11.8 Applications of LLMs: APIs and RAG

**Large language models (LLMs)** such as GPT and related Transformer architectures deliver state-of-the-art performance across a vast range of NLP tasks. However, we have seen that pre-training them may require extremely large datasets and massive computational resources. In practice, most organisations and individual practitioners do *not* train or fine-tune such models from scratch. Instead, they access pre-trained LLMs through an **Application Programming Interface (API)**. This inference-only workflow is lightweight: you transmit a prompt to a remote server, which runs the model and returns a response. Because the platform provider manages hardware, scaling, and model updates, developers can integrate advanced text analysis into applications—including financial ones—without owning specialised GPUs.

The following code example illustrates a **typical API workflow**. After securely reading an API key from a separate file, the script specifies a target model (here, a freely available OpenRouter model) and builds a list of conversational `messages`. This message history provides context, enabling the model to respond coherently across turns. Next, a `POST` request is sent to the OpenRouter `/chat/completions` API endpoint using the standard `requests` library. The JSON response from OpenRouter contains the model's final answer (and, if available, intermediate reasoning), which we then print to the console. This basic pattern—construct prompt, send request, parse response—is central to building end-to-end NLP applications that interact with production-grade LLMs.

The code below can be found in the file `code-API-OpenRouter.py` on the course website.

```python
# This script connects to OpenRouter through the OpenRouter API and
# obtains a reply from one of the free LLM models. You can obtain a
# free API key from https://openrouter.ai/settings/keys, then enter
# the API key into the file named `code-API-key-openrouter.txt`.

import requests

# Read API key from file or paste as text string, for example:
# API_KEY = 'paste_here'
with open('../code-API-key-openrouter.txt', 'r') as f:
    API_KEY = f.read().strip()

# You change the LLM model here.
model = 'minimax/minimax-m2:free'

# Example history: You can build it in a loop.
messages = [
    {'role': 'user',
     'content': 'First question: What is the capital of the UK?'},
    {'role': 'assistant',
     'content': 'London.'}, # From previous response.
    {'role': 'user',
     'content': 'Thanks, second question: And Spain?'}] # New prompt.
```

```
response = requests.post(
    'https://openrouter.ai/api/v1/chat/completions',
    headers={
        'Authorization': f'Bearer {API_KEY}',
        'Content-Type': 'application/json'},
    json={
        'model': model,
        'messages': messages})

data = response.json()['choices'][0]['message']
print(
    'Model reasoning:\n',
    data['reasoning'],
    '\n',
    sep = '')
print(
    'Model reply:\n',
    data['content'],
    sep = '')
```

Despite their power, LLMs face **two fundamental limitations** in finance.

1. **Stale knowledge:** Because LLMs are trained on fixed corpora, their parametric knowledge reflects information available only up to the training cutoff. They typically lack awareness of recent corporate events, macroeconomic developments, or real-time market data.

2. **Hallucination:** LLMs may generate confident but incorrect statements, including fabricated figures or nonexistent sources. In settings where factual accuracy is mandatory—e.g. compliance, research notes, or investment commentary—these weaknesses are serious liabilities.

A widely adopted solution is **Retrieval-Augmented Generation (RAG)**. Conceptually, RAG converts LLMs from taking a *closed-book* exam (relying solely on internal parameters) to an *open-book* exam (with access to external documents). The workflow is straightforward: When the user asks a question, the system first *retrieves* relevant, up-to-date documents from external sources (e.g. news articles, reports, or transcripts), appends these documents to the prompt, and only then asks the LLM to answer using that context. By injecting current information into the prompt, we implicitly "ground" the model's output, increasing factual reliability and relevance.

Consider a simple practical pipeline for financial text analytics. Suppose a user asks: "What is the sentiment on Apple stock today?" First, we retrieve recent articles about AAPL. Here, the `code-API-NewsDataIO.py` example integrates with a news API to fetch headlines and full-text descriptions within the past 48 hours (see Section 3.4 starting on page 28). Alternatively, we could retrieve SEC filings, earnings call transcripts, or analyst commentary scraped from the web. Second, we construct an *aug-*

*mented* prompt instructing the LLM to base its answer strictly on the provided documents. Third, we send this enriched prompt to the LLM API (using the same interface as e.g. in the OpenRouter example code shown before), yielding a summarised sentiment assessment grounded in retrieved evidence rather than the model's outdated or invented knowledge.

This way, RAG unlocks a powerful and pragmatic workflow for applying LLMs in financial applications. It combines real-time data retrieval with advanced natural-language reasoning, allowing practitioners to build tools that summarise earnings calls, extract risk factors from filings, or generate timely sentiment analyses. Critically, the approach alleviates stale knowledge and hallucination by anchoring generation to verifiable, external text. This makes RAG a promising and accessible framework for student projects, research prototypes, and industry-facing financial analytics systems.

Beyond freshness and hallucination, RAG faces a major **scalability challenge**. The simple workflow shown just now works well when only a handful of recent documents are involved. But many financial applications require searching across *thousands* of documents—e.g. all SEC filings over the past 20 years, including 10-K, 10-Q, and Form 8-K disclosures. Clearly, we cannot paste decades of text into a single prompt. Moreover, naive keyword search is insufficient: Simple string matching would entirely miss conceptual relationships (e.g. querying for "company risks" would fail to retrieve a section titled "Forward-Looking Statements and Market Hazards"). To scale RAG to large corpora and capture semantic similarity, we use **vector databases**. These systems store embeddings—vector representations learned from models such as BERT—and allow efficient similarity search. In this representation space, terms like "company risks" and "market hazards" lie near one another, enabling retrieval based on meaning rather than wording.

A modern **Advanced RAG** workflow proceeds in two phases. *Phase 1: Indexing (one-time setup).* First, the document corpus (e.g. all 10-K filings) is segmented into short chunks (e.g. paragraphs). Each chunk is encoded into an embedding vector using a pre-trained model such as BERT or Doc2Vec, then stored in the vector database. *Phase 2: Retrieval and generation (query-time).* When a user asks a question—e.g. "What are Apple's main competitive risks?"—we embed the question into a vector and query the vector database for the most similar chunks (e.g. top 5). Only those highly relevant passages are added to the LLM prompt; the LLM then produces a concise, contextually grounded answer. This hybrid design combines the semantic recall of vector search over massive text libraries with the reasoning and summarisation capabilities of modern LLMs, yielding accurate, scalable financial NLP systems.

## 11.9 Reinforcement Learning

What is reinforcement learning? You can think of it as a little robot who is in a room where there are lots of balls of different colors. The robot needs to find a red ball. It can perform certain actions such as walking around, stretching out its arm, and tightening his fingers to grab something (i.e. hopefully a red ball). Whenever it performs an action, the robot interacts with the environment and gets a reward based on that interaction (e.g. if it found the red ball, its batteries get recharged). Also, after performing an action, the little robot observes how its state changed. For example, if it made a step forward, it observes that it is now standing in a different position in the room. Through trial and error, the little robot learns what to do and what not to do, e.g. bump against a wall or fall down. In a nutshell, this is what reinforcement learning is all about.

As you can see, reinforcement learning is fuzzier than supervised learning. Although there is some reward, correct input/output pairs are never presented and sub-optimal actions are never explicitly corrected. The basic idea is that sometimes, due to the complexity of many situations encountered in real life, exact methods become infeasible, so we need to find a balance between exploration and exploitation. Exploration means to take a look at uncharted territory (e.g. the little robot walks over to another corner of the room he hasn't examined before), while exploitation in this context means to take advantage of current knowledge (e.g. it has found a way to walk around on this side of the room without bumping into its walls).

There are also related questions as to what payoff the little robot gets and the strategies it should use. As we have seen before, it gets rewarded when it finds a red ball and picks it up. But there are many red balls scattered throughout the room and the little robot should try to maximize the expected sum of *all* balls it picks up. So how should it go about to pick up the balls? Should it just go in a straight line, which might give the highest payoff in the short run? Or should it circle around the room, which might initially have a lower payoff but in the long run performs better. My point is that cutting corners might work in the short run, but not in the long run, and this fact should somehow be picked up by the little robot's reinforcement learning. It is really non-trivial how the robot's actions relate to its overall payoff which is the expected sum of all balls picked up.

Another point about the payoff is that there are two kinds of payoffs. The first one is the payoff that can be "seen" by the robot. If it picks up a red ball, it gets rewarded. But there is more to that. Let's say there is one red ball left on the other end of the room and the little robot makes a step towards it. There is no immediate reward, but it is definitely (and literally) a step in the right direction, because it brings the little robot closer to finally grabbing the ball. This is a different kind of payoff because the robot is doing the right thing, even though it does not directly result in a red ball in its hand. So my point is that it is difficult to specify a concrete payoff function that gives our little robot the correct information right after each action it takes. This is what I mean when I say that it is difficult or impossible to fully specify a payoff function that rewards or punishes each decision taken by the little robot.

Related to this are also strategic contexts, where there are two or more players. For example, you could put another little robot into that room and let both robots compete to find most red balls. How should our little robot react now? Should it just try to be faster? Or should it try to push the other robot off its feet?

Competitive self-play expands on this idea. The basic idea is that in order to become better and improve, you let the little robot play against a copy of itself. The good thing is that playing against itself has just the right difficulty level. It's not too difficult (so that the little robot doesn't always loose and doesn't learn anything) and not too easy (so that the little robot doesn't always win and doesn't learn anything either). There have been advances, e.g. with AlphaGo, where GU Li (a very famous Go player) said that "AlphaGo's self play games are incredible—we can learn many things from them."

How does this apply to text analytics and NLP in finance? The basic idea of reinforcement learning is that you're trying to find a solution to a very complex optimization problem. Through exploration and exploitation you may be able to find words or combination of words and other features that capture what you're looking for. For example, if you are using Twitter to predict the stock market, you might not only look at the words, but you might also want to look at the person who is writing the tweets and his position in the network of followers. The relations here can be very complicated (and noisy due to the stock market's reaction), so this is a very complex optimization problem that you might be able to solve approximately with reinforcement learning.

One key challenge is that the settings of reinforcement learning are usually experimental and not observational. This means in the example above that you can clean up the room, put the little robot back into its starting position, and start all over again, thus repeating the experiment. In finance, however, things are usually more complicated because experiments are often difficult to come by. If we take the example of the stock market, we can only observe from the past what happened in the stock market. We cannot readily observe an experiment, for example what would have happened if the financial crisis of 2007–2008 would not have occurred.

Another problem is that reinforcement learning is often formulated as a Markov decision process. The basic idea is that as long as you know today's state of nature, you don't need to know what happened yesterday. As long as the little robot knows where it stands and where the balls are, in that moment, it doesn't matter whether it got there by walking forwards or backwards in its previous move. But in financial markets, a longer history often matters. If a stock has been going up for the last year, chances are higher that it will keep going up in the future. This phenomenon is called "momentum." So a Markov decision process might not yield the optimal results because it is misspecified.

Depending on how much data the algorithm needs, you could still let it learn over time. For example, you give all the data available on Monday evening, let it make its decision (e.g. which stocks to buy or sell), and the reward would be how much money it made on Tuesday (the following trading day). You then move the algorithm forward by one trading day to give it all the data on Tuesday; rinse and repeat. The only limitation here might be the amount of data the algorithm needs. If you're looking at daily data and we assume we have approximately 100 years worth of data, we have about 25,500 data points (100 times 255, the number of trading days per year). If this amount of data is sufficient, reinforcement learning could potentially work.

When combined with deep learning, reinforcement learning is referred to as deep reinforcement learning (**DRL**). Typically, reinforcement learning is considered distinct from both supervised and unsupervised learning. It does not fall under supervised learning because it does not rely on a labeled training dataset. Conversely, it is not categorized as unsupervised learning since the reinforcement learning process is guided by expected rewards. Some of the most popular algorithms in reinforcement learning include **Q-learning**, deep Q-networks, and proximal policy optimization (PPO).

## 11.10 Standardized Variables

When using machine learning models, it is often important to transform the input and/or output variables in order to boost the performance of the model. A seemingly trivial but often effective method is to "standardize" some variables (sometimes also called calculating the "$z$-score" or the "standard score"). Often machine learning algorithms work better if they are provided with standardized variables, as their statistical properties fit the algorithm better. The basic idea is to "center" the variables so that they have mean zero:

$$z = x - \mu,$$

where $x$ is the raw score and $\mu$ is the mean of the sample or population.

Additionally, you can scale a variable so that it has a standard deviation of one:

$$z = \frac{x - \mu}{\sigma},$$

where $\sigma$ is the standard deviation of the sample or population.

There are several ways how to do this in Python, one of them is using the function `scale()` from the Scikit-Learn package. In the following example we think of each *column* as containing realizations of a random variable.

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X_train = np.array([[ 1., -1.,  2.],
...                     [ 2.,  0.,  0.],
...                     [ 0.,  1., -1.]])
>>> X_std = preprocessing.scale(X_train)
>>> X_std
array([[ 0.  ..., -1.22...,  1.34...],
       [ 1.22...,  0.  ..., -0.27...],
       [-1.22...,  1.22..., -1.07...]])
>>> X_std.mean(axis=0)
>>> np.mean(X_std, axis=0) # Check the mean of each column.
array([0., 0., 0.])
>>> X_std.std(axis=0)
>>> np.std(X_std, axis=0)  # Check the standard deviation of each column.
array([1., 1., 1.])
```

## 11.11 Overfitting and What to Do About It

Overfitting can be a a big problem in NLP and machine learning in general. It occurs when you fit a model too closely to a given training dataset, and the fitted model then works very badly on any new data it hasn't seen before (in the training set). We have already talked about an overfitting example on page 117.

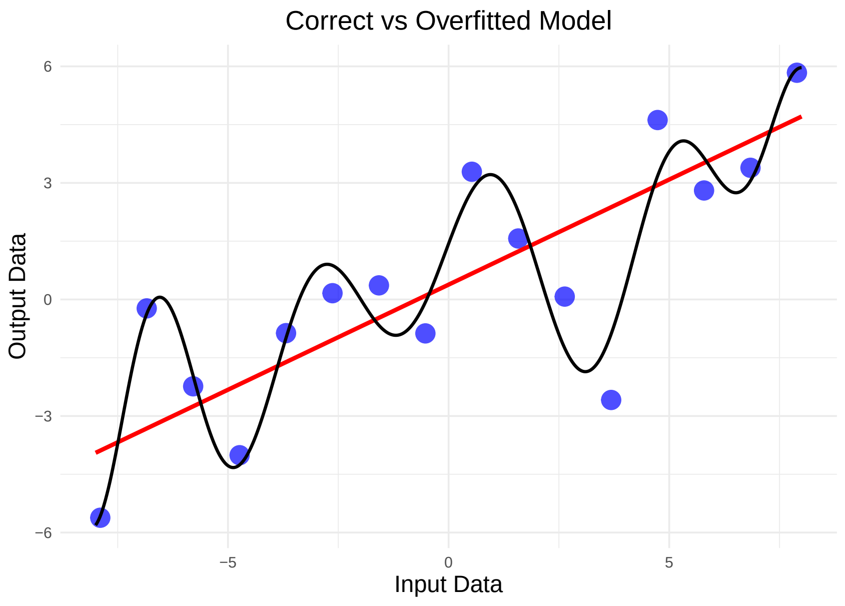
For illustration, we will use a very simple model that has one variable $x$ as input, one variable $y$ as output, and both variables are connected by a polynomial function, for example

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \qquad \text{or} \qquad y = b_0 + b_1 x + b_2 x^2.$$

These polynomials have different degrees, i.e. the highest power exponent. For example, the first polynomial's degree is 3 while the second polynomial's degree is 2. We can view the degree of each polynomial as a **hyperparameter**, i.e. a parameter we need to manually choose before model training begins. The degree of the polynomial (i.e. its hyperparameter) corresponds to the polynomial's complexity. As such, the model complexity of the first polynomial ($a_0 + a_1 x + a_2 x^2 + a_3 x^3$) is higher than that of the second model ($b_0 + b_1 x + b_2 x^2$). That is, the first polynomial can potentially fit more complex "shapes" of data. However, as we will see in this section, this is not always desirable because it can lead to overfitting.

The polynomial model is useful for illustration, but in machine learning we mostly use other models, e.g. LASSO adds a regularization hyperparameter to OLS regression, or an artificial neural network for deep learning has the number of hidden units as a hyperparameter. In any case, the principles discussed for avoiding overfitting and finding the best hyperparameters stay the same.

Let us start with the basic idea that there is some "true" model, but you can not see that model directly and instead can only see it through noisy observations of that model. For example, suppose the true model is linear as in the following figure, but all you can see are the data points:



What you could do with those data points is to fit a few models and hope that these models somehow capture the underlying "true" model. It would be tempting to fit for example a linear model (i.e. a special case of a polynomial model with degree 1) and a polynomial model (with degree larger than 1) to the data. As you can see, the polynomial model fits the training set's data points quite well (although not necessarily perfectly well), while the linear model does not. So only by looking at the training set, it seems that you should go with the polynomial model.

Of course, if you later obtain new observations that have not been included in the training set, you will find out that the linear model fits much better than the polynomial model on these new data points. You thus realize that you have **overfitted the polynomial model**, because it does not work very well out-of-sample. By **out-of-sample** we refer to a model's performance on new data not in the original training set/sample.

In general, overfitting can occur if you fit a model that is too general or if you have very limited training data (your training set is too small). The real tradeoff is between the true (but unknown) model and the noise. Ultimately you want to extract information about the true model and you want to ignore the noise as much as possible, but if you overfit, you end up doing the opposite, i.e. putting too much emphasis on the noise.

Before moving on, let us clarify what model fitting means and what model error means. **Model fitting** means the algorithm is finding those parameters of the model that make the model fit best to a given dataset. **Model error** shows how well the predictions made by a model fit a specific dataset. A low model error means that the model fits the dataset well. It is important to understand that for any model, the model error can be computed for any set of parameters on any dataset. In particular, it is possible to follow this three-step procedure:

1. Fit the model on dataset A

2. Save the fitted model parameters

3. Use these parameters to compute the model error on dataset B

The basic idea is that we want to find out how well the model generalizes to data it has not "seen" before during the model fitting. We will use this idea in the following procedure to detect when overfitting occurs and to avoid it.

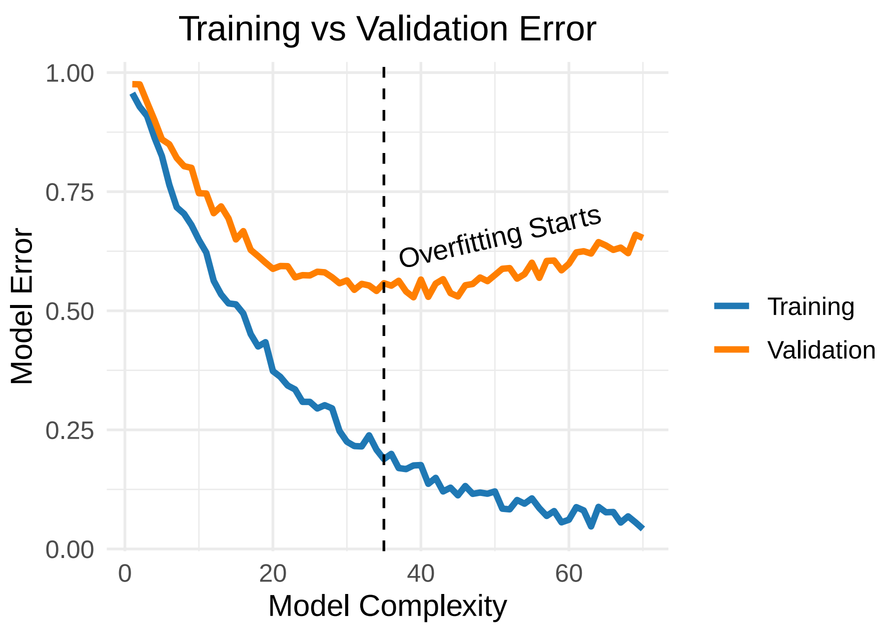
So **how do you avoid overfitting?** The basic idea is to **split the dataset you have available for fitting the model into two or even three non-overlapping subsets**. The terminology generally used can sometimes be a bit confusing because the terms are not always consistently applied. In any case, we use the following **conventions** in this book:

- If we are dealing with all **three datasets**, we call them "training set," "validation set," and "test set."

- If we omit the optional last dataset and just deal with the first **two datasets**, we call them "training set" and "validation set," or alternatively we call them "training set" and "test set."

In what follows we will focus on the case of three datasets. The basic idea is to have three independent samples from the same population. We will use these datasets to minimize the amount of overfitting of our model. The first dataset, the **training set** is used for model fitting, i.e. we are finding the model parameters that allow the model to fit the training set as best as possible. The second dataset, the **validation set** is used to validate your model in the sense that you check how well the fitted model performs on the new data, e.g. by looking at a statistic that summarizes the error of the model on the validation set. For example, you could look at various measures of goodness of fit or different loss functions for classification, such as **mean squared error (MSE)**, hinge loss, logistic loss, or cross entropy loss. At this part of the process you can iterate and fit different models, e.g. in the example above you could fit polynomials with different degrees. As you increase the model complexity (e.g. use polynomials with higher degrees), you would typically see a decrease in the training error (on the training set), which initially is mirrored by a decrease in the validation error (on the validation set). However, as the complexity of your model increases, at some point you would start overfitting the model, which means that the validation error would start to increase (while the training error keeps decreasing). This is the point where you stop increasing the model complexity because you have found the model that performs best out-of-sample.

The figure below shows on the $x$-axis the model complexity, on the $y$-axis the error. The decreasing blue line is the training error, while the u-shaped orange line is the validation error. By "training error" we refer to the error of the model on the training set. By "validation error" we refer to the error of the same model (which was fitted before to the training set) on the validation set. **The best predictive model is obtained at the minimum of the validation error.**



Finally, to get a better understanding for how the performance of your "best" model will be like in reality, you use the third dataset, the **test set**. Using this third dataset is optional, but can provide additional insights about your model's performance on new data. You can again compute the MSE or related measures. The basic idea is to compute a measure that tells us how the model is likely going to perform in reality on datasets we have not seen before. The reason why you don't want to use the validation set is that it is crucial to avoid using any dataset that has previously been used for model building (such as the training or validation set). Otherwise you will not get an unbiased estimate of your model's true out-of-sample performance.

One way to think about this whole procedure is that **the model has parameters and hyperparameters**. Let's consider again for illustration the polynomial model, which can be written as follows:

$$\sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

In this example, $\{a_0, a_1, \ldots, a_n\}$ are model parameters while $n$ is a hyperparameter. This is an example of a model with a single hyperparameter, but there are other models with no hyperparameters (e.g. linear regression) or with several hyperparameters. In any case, **the key characteristic of a hyperparameter is that it is set *prior* to the commencement of the model fitting.** So for example, if you fit a quadratic model on the data, it means you fix $n = 2$ before the model fitting, and then fit the model on the training set to estimate $\{a_0, a_1, \ldots, a_n\}$. **The iterative procedure described above is about pinning down the hyperparameter(s)** of your model, i.e. you go back and forth between the training set and validation set until you have found the hyperparameter(s) with the best out-of-sample performance.

Alternatively, overfitting can also be conceptualized in terms of the **duration of the training process**, especially when working with models that have an exceptionally large number of parameters. Take deep learning models, for example; they often contain billions of parameters, making it almost impossible to perfectly fit the training data in just one pass. As a result, the training process typically involves multiple iterations over the dataset, known as epochs. During each epoch, the model's parameters are incrementally adjusted, gradually moving closer to an optimal configuration. However, because the model has so many parameters, there is a significant risk that it will start to memorize the training data rather than generalize from it, leading to overfitting. To mitigate this, it is crucial to monitor the training process and implement strategies such as early stopping—halting the training before overfitting begins. By carefully controlling the number of training epochs, we can balance the model's ability to learn from the data without allowing it to overfit.

In terms of code implementation, it is often not necessary to "manually" split up a given dataset, as the Sklearn package already has this capability. The code below can be found in the file `code-sklearn-train-test-split.py` on the course website.

```python
# This script shows how to split up your data into training and
# testing subsets.
import numpy as np
from sklearn.model_selection import train_test_split
# Create data that is for illustration evenly-spaced.
X, y = np.arange(10).reshape((5, 2)), range(5)
X
list(y)
# Split up the data. `test_size` is the proportion of the dataset to
# be included in the test split. `random_state` is the seed of the
# random number generator (so that you get reproducible results when
# you run the code the next time).
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.33, random_state=42)
# Take a look at the training set.
X_train
y_train
# Take a look at the test set.
X_test
y_test
# Alternative way to split up the data. Do NOT shuffle the data before
# splitting.
train_test_split(y, shuffle=False)
```

# Chapter 12

# Text Processing

This chapter assumes basic Python knowledge. If you need to brush up your Python skills, check out the introduction to Python in Chapter 6 starting on page 41. Specifically, for string basics see **Sections 6.4 and 6.5.2 starting on pages 44 and 46, respectively**.

## 12.1 Common String Operations

- **String replication**:

  ```
  >>> 'hello ' * 8
  'hello hello hello hello hello hello hello hello '
  ```

- Length of a string (will include letters, numbers, whitespace, etc.):

  ```
  len('hello world')   # 11
  ```

- **Indexing** (or "**slicing**") of strings can be done in the usual way, see **Section 6.8 starting on page 76**. For example:

  ```
  >>> s = 'hello world'
  >>> s[2:5]
  'llo'
  ```

- **str.count** counts how often a character or a character sequence occurs. This can be used for example for bag-of-words (BoW), see Section 15.1 starting on page 179.

  ```
  >>> w = 'Hello World!'
  >>> w.count('o')
  2
  >>> w.count('h')   # None because it's case sensitive.
  0
  ```

```
>>> w = 'Andrew likes to run and he likes to cycle.'
>>> w.count('like')
2
>>> w.count('like ')
0
```

- **str.find** finds the first occurrence of a character or character sequence:

```
>>> w = 'Danny likes to swim and he likes to smile.'
>>> w.find('m')
18
>>> w.find('likes')
6
>>> w.find('likes', 9)   # Start looking after index number 9.
27
>>> w.find('likes', 9, -3) # ... but stop looking at index -3.
27
>>> x = 'hello world'
>>> if x.find('llo') != -1:  # Check whether can find 'llo' anywhere.
...     print('contains string')
...
contains string
```

- Converting to **upper or lower case**:

```
w = 'Hello World!'
w.upper()      # 'HELLO WORLD!'
w.lower()      # 'hello world!'
```

- **Boolean methods**, i.e. checking whether some properties of a string are true:

| *Method* | True *if* |
|---|---|
| `str.isalnum` | Only alphanumeric characters |
| `str.isalpha` | Only alphabetic characters |
| `str.islower` | All lower case |
| `str.isnumeric` | Only numeric characters |
| `str.isspace` | Only whitespace characters |
| `str.istitle` | String is in title case |
| `str.isupper` | All upper case |

- For example:

```
'hello'.isalnum()    # True
'hello#'.isalnum()   # False
'Hello'.isupper()    # Fase
'HELLO'.isupper()    # True
```

- **str.join**:

  ```
  w = 'I love NLP.'
  ''.join(w)            # Same string as before (no changes made).
  ' '.join(w)           # 'I   l o v e   N L P .'
  ''.join(reversed(w))  # '.PLN evol I'
  ','.join(['hello', 'world'])  # Combine list of strings into single string.
  ```

  Here's another example, where we join strings together with a given separator, in this example "_":

  ```
  >>> '_'.join(['hello', 'world', 'this', 'is', 'great'])
  'hello_world_this_is_great'
  ```

- **str.split** can be used as a rudimentary tokenizer, but it's usually better to use a more sophisticated solution, e.g. from Section 13.1 starting on page 161:

  ```
  >>> w = 'I have a balloon.'
  >>> w.split()      # Split on whitespace.
  ['I', 'have', 'a', 'balloon.']
  >>> w.split('a')   # Remove letter 'a'.
  ['I h', 've ', ' b', 'lloon.']
  ```

- **str.splitlines** splits a string using the newline delimiter, for example:

  ```
  >>> 'foo\nbar\n'.splitlines() # Equivalent to str.splitlines('foo\nbar\n')
  ['foo', 'bar']
  ```

  In fact, this is (in this case) preferable to `str.split` because the final newline results in an empty string:

  ```
  >>> 'foo\nbar\n'.split('\n')
  ['foo', 'bar', '']
  ```

- **str.replace**:

  ```
  >>> w = 'I have a balloon.'
  >>> w.replace('have', 'had')
  'I had a balloon.'
  ```

- **str.startswith**

```
>>> x = 'hello world'
>>> x.startswith('hello')    # Check whether it starts with 'hello'.
True
```

- **Check whether a character is in a string**:

```
>>> x = 'hello world'
>>> if 'e' in x:             # Check whether 'e' in string.
...     print('yay')
...
yay
```

## 12.2 Regular Expressions

- Sometimes it is difficult to accomplish your text preprocessing tasks using the common string operations discussed in Section 12.1 starting on page 153. In this case regular expressions might help.
- Regular expressions, or **regex** for short, can take your text processing skills to the next level.
- Regex are strings with a special syntax which allow you to match patterns in other strings.
- You can use regex to find things in text documents (e.g. web links), parse email addresses, and remove/replace unwanted characters.
- In general, regex can be very useful if you have to **preprocess text**.

## 12.3 Regex and the re Module

The `re` module in Python provides regular expression matching operations. It allows you to operationalize some of your text preprocessing needs using regular expressions.

- You can load it with `import re`.
- The functions in the `re` module work in the same way. Always pass **pattern first** and **string second**. The functions may return a string, match object, or an iterator.
- The **pattern** tells the function what part of the string to search or replace.
- The **string** is contains the text the function operates on.

We next introduce some of the most important functions and patterns, and then illustrate how to use the functions and patterns together. We begin by presenting the **most important functions**:

- `sub` replaces each match of a regex pattern in a string with a replacement string.
- `split` splits a string based on a regex pattern. Can be used for tokenization (e.g. with the `\s+` or `\W+` patterns).
- `match` finds a match of the regex pattern in a string. The match has to occur starting from the beginning of the string.
- `search` searches for a regex pattern. Similar to `match`, but doesn't require to match from the beginning of the string. Like `match`, it stops after the first match.
- `findall` finds all regex patterns in a string.

It is often good practice to use a raw string (see Section 6.5.2 starting on page 46) to define the regex pattern, i.e. **prefix the string containing your regex patterns** with `r` (e.g. `r'\n'`). Below we have an example to demonstrate that using a raw string or a "normal" string for the regex pattern can make a difference.

```
import re
text = 'This is a string with a \\n in the middle.'
print(text) # This is a string with a \n in the middle.
re.search(r'\\n', text)  # Successfully found the `\n`.
re.search('\\n', text)   # Looks for a newline, not what we want.
```

As we have just seen, whether or not we use a raw string for the regex pattern can make a difference. Note that it is possible to get the same result with a "normal" string, although it would involve escaping each backslash with the escape character (i.e. another backslash) as we have seen in Section 6.5.2 starting on page 46:

```
re.search('\\\\n', text)   # Same result as with raw string `r'\\n'`.
'\\\\n' == r'\\n'          # `True`.
```

Next we provide an overview of **important regex patterns**. These patterns control the behavior of the methods and functions we have just introduced. The patterns tell Python what kind of text you are looking for, or which part of the text you would like to replace with something else.

| *Pattern* | *Matches* | *Example* |
|---|---|---|
| `\d` | Digit | 8 |
| `\s` | Space | `' '` |
| `\S` | Anything that is NOT a space (capital letters negates) | `'no_spaces'` |
| + or * | Greedy match (grabs repeats of single letters or whole patterns) | `'aaaaa'` |
| `\w+` | Word characters, including alphanumeric characters and "_" | `'Hello_world7'` |
| `\W` | Opposite of `\w`, anything that is not a word character | |
| `.*` | Wildcard (match any letter or symbol) | `'H3llo world'` |
| `[]` | Character ranges | |
| `[a-z]+` | Lowercase group (creates group of characters by putting them inside square brackets) | `'abcdefb'` |
| `[A-Za-z]+` | Upper and lowercase English alphabet | `'ABCdef'` |
| `[0-9]` | Number from 0 to 9 | 8 |
| `\|` | Logical OR | |
| `\d+\|[a-z]+` | Digits or lowercase characters | `hello` |
| `()` | Define a group, used to find explicit set of characters | |
| `\w+(ab\d+)` | Extract `ab88` from `hello_ab88world` | `ab88` |

Now we tie everything together, the functions and patterns, to find out how they work together. The following code shows examples on how to use regex (see the file `code-re-regex.py` on the course website). Run the code to see what happens.

```python
# This file contains a few examples of how to use the re module and
# how to deal with regular expressions.
import re                           # Import 're' module.

# Replace pattern in string.
re.sub(r'H', 'h', 'Hello␣world!')
re.sub(r'\s+', '␣', 'This␣␣␣␣␣is␣a␣␣\t␣␣\n␣␣test')

# Simple tokenization.
re.split(r'\s+', 'This␣is␣a␣␣␣test.')
re.split(r'\W+', 'This␣is␣a␣␣␣test.')
# Split into sentences.
re.split(r'[.?!]', "Hello␣world!␣Let's␣write␣regex.␣Isn't␣this␣great?")
re.match(r'abc', 'abcdef')        # Match starting at beginning of string.
re.match(r'bc', 'abcdef')         # Match starting at beginning of string.
re.match(r'\w+', 'Hello␣world!') # Match a word.
re.match(r'[a-z0-9␣]+', 'lowercase␣and␣nums␣like␣8,␣but␣no␣commas.')

re.search(r'cde', 'abcdef') # Fill also match in the middle of string.
m = re.search(r'coconuts', 'I␣love␣coconuts.')
print(m.start(), m.end())        # Print start and end indices.
# Find square bracket containing a word (but no space or anything else).
re.search(r'\[\w+\]', 'Hello␣[wind␣bla]␣this␣is␣[nice].')

re.findall(r'\w+', 'This␣is␣a␣␣␣test.') # Find all words.
# Find all capitalized words.
re.findall(r'[A-Z]\w*', 'Hello␣world,␣I␣love␣Hong␣Kong.')
re.findall(r'\d+', 'The␣novel␣1984␣was␣published␣in␣1949.')
# Match digits and alphabetic characters.
re.findall(r'\d+|[A-Za-z]+', 'Hello,␣there␣are␣30␣cats!')

# Use groups `()` to capture strings that match the groups and ignore
# the rest of the string outside the groups. Result will be a list of
# strings (if one group) or a list of tuples (if multiple groups).
re.findall(r'\w+(abc\d+)=\s+', 'hello_abc7=␣abc8=␣hello_abc9=')
re.findall(r'(\w+)=(\d+)', 'The␣option␣has␣price=7␣and␣strike=30.')
```

# Chapter 13

# Text Preprocessing

This chapter delves into common preprocessing steps for text analytics. These steps would be done before conducting further text analytics or NLP. Keep in mind that **not all of these steps are necessary at all times; depending on the application you may omit some of them.** Furthermore, some of these steps depend on the language you are analyzing, e.g. stemming might be valid for English but not for Chinese.

## 13.1 Tokenization

The basic idea of tokenization is to split up the whole text document into smaller parts, the so-called *tokens*. A token can be a single word, but it can also be an n-gram, a sentence, a paragraph, or all hashtags (e.g. in a tweet). Tokenization can be important to get a deeper understanding of words. Consider for example the following sentence:
`I don’t like Jeff’s gloves.`
After tokenization, this sentence becomes (with tokens indicated in brackets for clarity):
`(I) (do) (n’t) (like) (Jeff) (’s) (gloves) (.)`
You can see the negation of “do” as well as the “’s” to indicate the possessive noun.

There are several libraries that can help you with tokenization such as NLTK and gensim. For an example of tokenization using gensim’s tokenizer `gensim.utils.tokenize` see Section 20.5.2 starting on page 210.

The NLTK library provides a few tokenizers that are specialized for certain applications, such as word or sentence tokenization. Furthermore, there is a special tokenizer for tweets.

- `word_tokenize` to tokenize text into words.
- `sent_tokenize` to tokenize text into sentences.
- `regexp_tokenize` to tokenize text based on a regex pattern.
- `TweetTokenizer` for tweet tokenization, to separate hashtags, mentions, and repeated punctuation marks!!!

Here are some examples on how to use tokenization with the NLTK package (see the file `code-nltk-tokenization.py` on the course website). Please run the code and take a look at its output.

```
# This file shows tokenization examples using the NLTK package.
from nltk.tokenize import \
    word_tokenize, sent_tokenize, regexp_tokenize, TweetTokenizer
word_tokenize("Hi there!")
sent_tokenize('Hello world. I love HK!')
# Make set of unique tokens.
set(word_tokenize('I love HK. I love NYC.'))
# Tokenize based on regular expression.
regexp_tokenize(
    'ACTOR #1: Have you found them?',
    r'\w+|#\d+|\?|!')
# Find hastags in tweets.
regexp_tokenize(
    'This is a great #NLP exercise.',
    r'#\w+')
# Find mentions and hashtags in tweets.
regexp_tokenize(
    'great #NLP exercise from @user123.',
    r'[#@]\w+')             # Alternative: r'#\w+|@\w+'
tt = TweetTokenizer()      # Instantiate the `TweetTokenizer` class.
[tt.tokenize(tweet)
 for tweet in
 ['thanks @user123', '#NLP is fun!']]
```

Byte pair encoding **(BPE) is a subword tokenization** technique that effectively bridges the gap between character-level and word-level tokenization. Character-level tokenization breaks text into individual characters (e.g., “running” → `'r'`, `'u'`, `'n'`, `'n'`, `'i'`, `'n'`, `'g'`), offering high granularity but often lacking meaningful representation of larger linguistic units. Word-level tokenization, on the other hand, splits text into words (e.g., “running is fun” → `'running'`, `'is'`, `'fun'`) but struggles with out-of-vocabulary (OOV) words and languages with complex word formation rules, such as those in morphologically rich languages. While English is not heavily morphologically rich, it still exhibits some word variations through prefixes, suffixes, and inflections. For example, the root word “run” can appear as “runs,” “running,” or “runner,” depending on grammatical context. Word-level tokenization would treat each of these as separate tokens, increasing vocabulary size and complicating rare word handling.

BPE addresses these challenges by starting with character-level tokens and **iteratively merging the most frequent adjacent character pairs to form subword units**. For instance, in a dataset containing “runs,” “running,” and “runner,” BPE might first merge `r` and `u` to form `ru`, then merge `ru` and `n` to create `run`, and eventually learn the subword units `run`, `ning`, and `ner`. This process continues until the desired vocabulary size is reached, balancing coverage and efficiency. By encoding frequent patterns as subwords, **BPE captures both common words as single tokens, and rare or complex words as combinations of smaller units**, making it highly effective for handling diverse vocabularies in modern NLP tasks.

The following code demonstrates how to create byte pair encoding (BPE) using the SentencePiece library (see the file `code-sentencepiece-BPE-tokenization.py` on the course website). Please run the code and take a look at its output.

```python
# This script uses the SentencePiece library to demonstrate byte pair
# encoding (BPE).
import sentencepiece as spm

# Train a BPE model. In real-world applications, the vocabulary size
# (`vocab_size` parameter) is usually between 10,000 and
# 30,000. However, in the "small" example we just choose 50 for
# demonstration purposes.
spm.SentencePieceTrainer.train(
    input='data-example-text-document.txt',
    model_prefix='data-bpe-model',
    vocab_size=50,              # Adjust vocabulary size as needed.
    model_type='bpe')           # Use BPE (Byte Pair Encoding).

# Load the trained model.
sp = spm.SentencePieceProcessor(model_file='data-bpe-model.model')

# Tokenize a sentence.
sentence = 'Gardening brings peace and joy, connecting people with nature.'
# Note that during real-world applications, you could use
# `out_type=int` which produces output tokens that are integer indices
# instead of strings.
tokens = sp.encode(sentence, out_type=str)
# Take a look at the BPE tokens of the example sentence. Note that `_`
# is a special character used by spm to indicate a word boundary.
print(tokens)

# Decode the tokens back to a sentence.
print(sp.decode(tokens))

# Clean up files generated during model training.
import os
os.remove('data-bpe-model.model')
os.remove('data-bpe-model.vocab')
```

## 13.2 Stop Word Removal

Stop word removal means we exclude some terms, e.g. “the,” “a,” “is,” “at,” “which,” or “on” (so-called “**stop words**”). These are words that do not contribute a lot to a deeper understanding of the text in consideration as they do not have significant meaning. You can see programming examples of stop word removals in Section 15.4 starting on page 180 and in Section 20.5 starting on page 208. **Keep in mind that stop word removal may or may not be advisable depending on the context and techniques used.** For example, it is usually not recommended to use stop word removal before BPE.

## 13.3 Stemming and Lemmatization

Here you combine different grammatical forms of the same words, e.g. "travel," "traveling," and "traveled." Again the basic idea is to remove noisy data (in this case the suffix) that are of limited use in understanding the text. The difference between stemming and lemmatization is that stemming often uses a crude heuristic process, while lemmatization tries to do things properly with the use of a vocabulary and morphological analysis of words. For example, when considering the token "saw," stemming might just return "s" whereas lemmatization would attempt to return either "saw" or "see" depending on whether the token is used as a noun or verb.

The **Porter stemming algorithm** is an efficient rule-based approach to reduce words to their root or base form, known as the 'stem.' Developed by Martin Porter in 1980, the algorithm consists of a series of systematic steps that iteratively remove various affixes, such as prefixes, suffixes, or inflections, from a given word. These steps, referred to as phases, involve the application of a sequence of conditional rules that depend on the structure and length of the word. The algorithm terminates when no more rules can be applied or when all five phases have been completed. Due to its simplicity and effectiveness, the Porter stemming algorithm has become one of the most widely used stemming techniques in natural language processing and information retrieval. By converting words to their stems, the algorithm enables search engines, text analysis tools, and other applications to better understand the underlying meaning of a text, group similar words together, and return more relevant results to users.

```
from nltk.stem import PorterStemmer
ps = PorterStemmer()
ps.stem('program')           # program
ps.stem('programs')          # program
ps.stem('programer')         # program
ps.stem('programing')        # program
ps.stem('programers')        # program
```

The original Porter stemming algorithm was specifically designed for the **English** language. Its rules and heuristics are tailored to handle the morphological structures and affixes commonly found in English words. However, inspired by the success of the Porter stemmer, researchers and developers have created adaptations of the algorithm for other languages. These adaptations, often referred to as "**Snowball stemmers**" (after the Snowball language developed by Martin Porter for creating stemming algorithms), include stemmers for languages such as **French, German, Spanish, Dutch, Italian, Portuguese**, and many more. While the core idea of reducing words to their stems remains consistent, each adapted stemmer has its own set of language-specific rules and heuristics to handle the unique morphological structures of the respective language.

The **WordNet lemmatizer** is a natural language processing tool that reduces words to their base or dictionary form, known as 'lemmas.' Based on the extensive lexical database called WordNet, this lemmatizer utilizes a large collection of English words, their meanings, and relationships to identify the appropriate lemma for a given word. Unlike stemming algorithms, such as the Porter stemmer, which rely on heuristics to systematically remove affixes, the WordNet lemmatizer takes into consideration the context and part of speech of a word to produce more accurate and meaningful results. By returning words to their lemmas, the WordNet lemmatizer enables search engines, text analysis tools, and other applications to better understand the semantics of a text, group related words together, and deliver more relevant results to users. Due to its linguistic and contextual awareness, the WordNet lemmatizer is often preferred over stemming algorithms when working with higher-level text analysis tasks, where preserving the meaning of words is crucial.

```
from nltk.stem import WordNetLemmatizer
wnl = WordNetLemmatizer()
wnl.lemmatize('dogs')        # dog
wnl.lemmatize('churches')    # church
wnl.lemmatize('aardwolves')  # aardwolf
wnl.lemmatize('abaci')       # abacus
wnl.lemmatize('hardrock')    # hardrock
```

## 13.4 Lower Case Conversion

Lower case conversion means you convert all words to lowercase. This can be important, e.g. some words appear at the beginning of a sentence and are thus capitalized, but their content is the same as their lower case version.

```
>>> 'Walk' == 'walk'    # To Python those strings are not the same!
False
>>> x = 'Example string.'
>>> x.lower()           # Convert to lower case.
'example string.'
```

## 13.5 Synonyms

For example "begin" and "commence" have nearly the same meaning and therefore could be treated the same. You could thus replace all synonyms, e.g. you would replace "commence" and "start" by "begin." For a given word (such as "begin" in the code below), you can find all its synonyms as follows:

```
from nltk.corpus import wordnet
```

```
syns = []
for s in wordnet.synsets('begin'):
    for lem in s.lemmas():
        syns.append(lem.name())

print(set(syns))
```

The output is as follows:

```
{'set_about',
 'get_down',
 'Begin',
 'start',
 'start_out',
 'begin',
 'set_out',
 'commence',
 'get',
 'lead_off',
 'Menachem_Begin'}
```

## 13.6 Special words

For example "Microsoft Windows" has nothing to do with the common use of the term "Windows." Or "New York" should really be treated as a single term instead of the two separate words "New" and "York." If you end up having a lot of these kinds of words in your specific application, you can use named entity recognition (NER) as discussed in Chapter 10 starting on page 111. Alternatively you might consider concatenating them e.g. to "MicrosoftWindows" and "NewYork" to make sure they are treated correctly. Of course, this involves some hand-coding and in many applications might after all not be necessary. But it might make an important difference in some specific applications.

```
>>> s = 'New York and California'
>>> s.replace('New York', 'NewYork')
'NewYork and California'
```

## 13.7 Part-of-Speech Tagging and Parsing

**Part-of-speech (POS) tagging** is an essential component in the field of Natural Language Processing (NLP), serving as a crucial preprocessing step for various language understanding tasks. POS tagging involves assigning grammatical categories such as nouns, verbs, adjectives, and adverbs to individual words or tokens within a given text. By providing syntactic context, POS tagging enables more accurate interpretation of text and enhances the performance of downstream NLP tasks, such as named entity recognition (see Chapter 10 starting on page 111), sentiment analysis (see Chapter 9 starting on page 107), and machine translation. Various techniques have been developed for POS tagging, ranging from rule-based methods and statistical models to state-of-the-art deep learning approaches.

For example the sentence "And now for something completely different" becomes:

```
And/CC now/RB for/IN something/NN completely/RB different/JJ
```

In this example, CC is a coordinating conjunction, RB is an adverb, IN is a preposition, NN is a noun, and JJ is an adjective.

The following code demonstrates the use of the Natural Language Toolkit (NLTK) library in Python for POS tagging and noun phrase parsing. NLTK is a widely-used and powerful library that provides essential tools and resources for NLP tasks. In this example, we will first tokenize and apply POS tagging to a given text, and then use a predefined grammar to extract noun phrases from the tagged text. The code leverages NLTK's built-in methods, such as `word_tokenize`, `pos_tag`, and `RegexpParser`, to efficiently accomplish these tasks, ultimately showcasing the ease of implementing POS tagging and noun phrase parsing using NLTK. The code below can be found in the file `code-nltk-POS-and-parsing.py` on the course website.

```python
# Part-of-speech tagging and noun phrase parsing using NLTK.
from nltk import pos_tag, RegexpParser, DependencyGraph, word_tokenize
# Input sentence.
sentence = 'The quick brown fox jumps over the lazy dog.'
# Tokenize the sentence.
tokens = word_tokenize(sentence)
# Part-of-speech tagging.
pos_tags = pos_tag(tokens)
print('Part-of-speech tags:')
print(pos_tags)
# Define a noun phrase chunk grammar. Here we define a noun phrase as
# an optional determiner (such as "the," "a," or "an"), followed by
# zero or more adjectives, and ending with a singular noun. More
# complex rules can be created to capture different types of noun
# phrases or other syntactic structures.
grammar = 'NP: {<DT>?<JJ>*<NN>}'
# Create a RegexpParser object.
chunk_parser = RegexpParser(grammar)
# Parse the sentence using the chunk grammar.
chunk_tree = chunk_parser.parse(pos_tags)
print('\nNoun phrases:')
chunk_tree.pretty_print()
```

**Dependency parsing** is a fundamental NLP task that seeks to analyze and represent the syntactic structure of a sentence by identifying the relationships between words. At the core of this technique lies the concept of a dependency grammar, which posits that each word in a sentence has a specific function, or "role," in relation to other words. The outcome of dependency parsing is a tree-like structure, known as a dependency tree or parse tree, where each word is a node and the edges represent the dependency relationships between them. By capturing these relationships, dependency parsing enables a deeper understanding of the meaning and context of a sentence, facilitating numerous NLP applications such as information extraction, sentiment analysis, machine translation, and question-answering systems. As a result, dependency parsing has become a critical component in the development of efficient and intelligent language processing tools and systems.

NLTK does not have a built-in dependency parser. NLTK is more focused on providing tools for rule-based parsing, such as context-free grammar parsers and regular expression parsers for chunking. While NLTK is a powerful library for many natural language processing tasks, it does not include a modern dependency parser.

Stanza, on the other hand, is built on top of the Stanford CoreNLP project and provides a neural network-based dependency parser, which is more accurate and efficient than rule-based parsers. If you want to perform dependency parsing, using Stanza or another library like SpaCy is a better choice than NLTK. Both Stanza and SpaCy offer pre-trained models for dependency parsing in multiple languages, making it easier to perform this task without having to define your own grammar rules or training a parser from scratch. The code below can be found in the file `code-stanza-dependency-parsing.py` on the course website.

```
# Dependency parsing using Stanza, which provides a neural
# network-based dependency parser.
import stanza
# Initialize the Stanza pipeline. By specifying ‘processors=‘ you can
# control which NLP tasks are performed by the pipeline.
nlp = stanza.Pipeline(lang='en', processors='tokenize,mwt,pos,lemma,depparse')
# Process the input sentence.
doc = nlp('The␣quick␣brown␣fox␣jumps␣over␣the␣lazy␣dog.')
# Extract and print dependency relations.
for sent in doc.sentences:
    for dep in sent.dependencies:
        head = dep[0]
        relation = dep[1]
        dependent = dep[2]
        print(f'{dependent.text}␣({head.id}-{dependent.id})␣->␣{relation}␣({head.te
```

In the following code, we showcase the use of the spaCy library, a popular and high-performance Python library designed for advanced NLP tasks. With a focus on POS tagging and dependency parsing, this example demonstrates the ease and efficiency of utilizing SpaCy's powerful functionalities. Leveraging the library's pre-trained models, the code will perform POS tagging and analyze the syntactic relationships between words in a given text. By employing SpaCy's `load` function to access the language model and the `Doc` object to process the input text, this example highlights the simplicity and effectiveness of incorporating SpaCy for POS tagging and dependency parsing in NLP projects. The code below can be found in the file `code-spacy-POS.py` on the course website.

```
# Part-of-speech (POS) tagging, noun phrase extraction, and dependency
# parsing using the spaCy library.
import spacy
# Load the pre-trained language model.
nlp = spacy.load('en_core_web_sm')
# Process the text.
text = "The quick brown fox jumps over the lazy dog."
doc = nlp(text)            # Get a `Doc` object via the `nlp` object.
# POS tagging.
print("POS tagging:")
for token in doc:
    print(f"{token.text:10} {token.pos_:10}")
# Noun phrase extraction (noun chunks).
print("\nNoun phrases:")
for chunk in doc.noun_chunks:
    print(chunk.text)
# Dependency parsing.
print("\nDependency parsing:")
for token in doc:
    print(f"{token.text:10} {token.dep_:10} {token.head.text:10}")
```

TextBlob can be considered as a wrapper around NLTK and Pattern libraries for some of its functionalities. TextBlob simplifies the process of working with textual data by providing an easy-to-use API, making it more accessible to beginners and users who want to perform quick NLP tasks without diving deep into the complexities of NLTK or other advanced NLP libraries. For example, when you perform part-of-speech tagging using TextBlob, it internally uses NLTK's POS tagger. However, TextBlob also uses Pattern for some other tasks, such as sentiment analysis. The code below can be found in the file `code-textblob-POS.py` on the course website.

```
# Part-of-speech tagging and noun phrase extraxtion using
# TextBlob. TextBlob doesn't provide direct dependency parsing.
from textblob import TextBlob
text = "The quick brown fox jumps over the lazy dog."
# Create a TextBlob object.
blob = TextBlob(text)
# Part-of-speech (POS) tagging.
print("Part-of-speech tagging:")
print(blob.tags, '\n')
# Noun phrase extraction.
print("Noun phrases:")
print(blob.noun_phrases, '\n')
```

## 13.8 Reducing the Number of Words in the Text Corpus

Often you end up with a text corpus that contains a large amount of different words. In some cases we may want to reduce the set of words we are looking at in our investigation, e.g. because some of these words may be nonsensical (e.g. due to an error in the database) or because they have little meaning. To reduce the set of words in your text corpus, you can use one or several of the following methods:

- Discard all one- or two-letter words, e.g. "a" or "to."
- Exclude numbers.
- For example, you could remove words that occur only in 0.5% or less of all documents. (The threshold level of 0.5% is arbitrary and can be chosen differently depending on your application.) So if a word does not occur in at least 99.5% of all documents, you remove it from your text corpus. To decide whether to remove a given word, you would cycle through all documents, check whether the word occurs in each document, and then divide the number of documents it occurred in by the total number of documents. If this number is less than 0.995, you would remove that word.
- Alternatively you could for example only keep the 2000 most commonly-used words and discard the rest. The basic idea is that you can express yourself pretty well if you have a vocabulary of 2000 words (or even less), even though the English language for example has more than 100,000 words in total. Of course, whether you use 2000 as the cutoff depends on your specific application and you might want to use different cutoff levels.
- Use a dictionary and only keep words that occur in this dictionary, i.e. remove a word if it does not have a matching entry in the dictionary.

## 13.9 n-Grams

The basic idea of n-grams is to combine several words. Consider for example the sentence "I live in HK." If you set $n = 2$ (so-called bigrams), you will end up with: (I live), (live in), (in HK). So the basic idea is to not look at single words, but instead to look at **combinations of words** in order to capture more aspects of the language structure. The advantage of n-grams are that you can take care of word order, e.g. if the text says "not happy," you could capture the negation in a bigram. The disadvantage is that your vectors have a high dimensionality and tend to be very sparse. The following is an example of bigrams using the NLTK package:

```
from nltk.tokenize import word_tokenize
from nltk.util import ngrams
s = 'I live in HK.'
```

```
ngs = ngrams(word_tokenize(s), 2)
[' '.join(ng) for ng in ngs]
# Output: ['I live', 'live in', 'in HK', 'HK .']
```

An alternative implementation using TextBlob:

```
from textblob import TextBlob
ngs = TextBlob('I live in HK.').ngrams(n=2)
[' '.join(ng) for ng in ngs]
# Output: ['I live', 'live in', 'in HK', 'HK .']
```

In practice you might also consider using the `ngram_range` argument to Sklearn's `CountVectorizer` function. For an example usage of `CountVectorizer`, see Section 15.6 starting on page 181.

## 13.10 Whitespace Elimination

Whitespace elimination means that you remove excessive whitespace so that you end up having a contiguous sequence of words. Oftentimes you do not have to worry too much about whitespace elimination as it is done by the tokenizer, see Section 13.1 starting on page 161. But if you would like to eliminate whitespace yourself you could use some of the following examples (the last example uses regular expressions, see Section 12.3 starting on page 156):

```
>>> x = '    Test of leading and trailing whitespace   \n   '
>>> x.strip()
'Test of leading and trailing whitespace'
>>> x = '   Test   of several     whitespace    \n   '
>>> ' '.join(x.split())
'Test of several whitespace'
>>> import re                  # Use regular expressions.
>>> re.sub(r'\s+', r' ', x)
' Test of several whitespace '
```

# Chapter 14

# Converting Text to Numbers

Ultimately we need numerical representations of data that we can use as input for further analysis, typically via machine learning as described in Chapter 11 starting on page 115. The key question is then, after preprocessing the text (Chapter 13 starting on page 161), how do we convert the text to numbers? As we will see in this chapter, we can convert text to **numerical vectors** to represent the information contained in the text. To this end, we can decide whether we would like a vector to represent an entire document or an individual word.

Document-level and word-level vector representations of text serve different purposes in natural language processing. **Document-level vector** representations capture information about the entire document, focusing on term frequencies (e.g. Section 14.2) and their importance within the document (e.g. Section 14.3). These models are well-suited for tasks such as document classification, clustering, and information retrieval. On the other hand, **word-level vector** representations refer to techniques that capture information about individual words in a text. These representations can range from basic methods like one-hot encoding (Section 14.4), which assigns a unique high-dimensional, sparse vector to each word without capturing semantic relationships, to more advanced techniques known as word embeddings, such as word2vec, GloVe, and FastText. **Word embeddings** (Section 14.5) are dense, low-dimensional representations that focus on capturing semantic meaning and contextual relationships between words, enabling more fine-grained analysis of language and better reasoning about the meanings and relationships between specific words and phrases. Overall, the main conceptual difference between document-level and word-level vector representations lies in the granularity of the information they capture (e.g. representing a whole text document or a single word) and their specific applications in NLP tasks.

A **vector space** in the context of natural language processing provides a structured framework that connects document-level or word-level vectors within a unified geometric space. While document-level and word-level vectors individually represent textual data as numerical vectors, the vector space allows these vectors to be organized, compared, and manipulated in a coherent and meaningful way.

The vector space is essential for several reasons:

1. Comparability: By placing vectors within a shared vector space, it becomes possible to measure the similarity or distance between them using various metrics, such as cosine similarity or Euclidean distance, see Chapter 18 starting on page 193. This enables tasks like document classification, clustering, information retrieval, and word similarity analysis.

2. Mathematical operations: The vector space allows for the application of mathematical operations like addition, subtraction, or scalar multiplication on the vectors. This is particularly useful in word-level vector spaces, where operations can reveal semantic relationships between words or help in solving analogies. For further details see Section 14.5 starting on page 177.

3. Dimensionality reduction: The vector space framework facilitates the use of dimensionality reduction techniques like latent semantic analysis (LSA) or t-distributed stochastic neighbor embedding (t-SNE) to analyze and visualize high-dimensional vectors in lower-dimensional spaces. This can help in understanding the underlying structure and relationships between the textual data. For further details see Chapter 19 starting on page 197.

4. Algorithmic processing: Many machine learning algorithms and natural language processing models rely on the vector space structure to process and reason about textual data. For example, neural networks require continuous numerical inputs, which can be provided by mapping words and documents into a vector space. While the vectors alone can indeed serve as inputs for these algorithms, the vector space framework ensures compatibility by maintaining a consistent dimensional structure and feature representation across all vectors. Additionally, the vector space facilitates the use of various preprocessing, normalization, and feature engineering techniques, such as tf-idf weighting (Section 14.3) or vector normalization (i.e. scaling vectors to a unit length such as Euclidean length of 1), which can improve the performance of the algorithms in certain tasks.

In summary, the vector space concept not only connects document-level or word-level vectors within a shared geometric space, but also provides a consistent and structured environment for organizing, comparing, and manipulating these representations. This framework enables various natural language processing tasks and applications, as well as the development of algorithms that can reason about and work with textual data effectively.

## 14.1 Binary Vector (Document-level)

The simplest way is to simply check whether each word is contained in the text document. If a word occurs in the document, the word gets a value of one (or `True`), if it is absent it gets zero (or `False`). So each document can be represented by a **binary vector**, where each vector element corresponds to a different word, indicating whether or not this word occurs in the document.

To begin with, we typically have a list of words that are somehow relevant to our task at hand, for example, the 3,000 most common words of the English language. This list is often called **lexicon** or **dictionary**, representing the vocabulary or the set of unique words present in the corpus. We could then cycle through this lexicon to check whether each of these words is contained in our document:

```
>>> lexicon = ['love', 'like', 'HK', 'NYC']
>>> document = 'I love HK'.split()
>>> binary_document_vector = [word in document for word in lexicon]
>>> binary_document_vector
[True, False, True, False]
```

## 14.2 Bag-of-Words (Document-level)

Another simple way is to count the words in each document, which is the **bag-of-words** (BoW) text representation. This way, you can think of one text document being represented by a **vector containing the word counts**. And several documents are just a collection of several such vectors. We cover several code examples in **Chapter 15 starting on page 179** to demonstrate how to compute BoW.

## 14.3 Word Weighting (Document-level)

Another way is to use a different **word weighting**. The basic idea is that instead of simply using the word counts, we give those words that are more important a higher weight while reducing the weight on the unimportant words.

A commonly-used weighting scheme is "**tf-idf**". There are two elements to tf-idf. The first one is that the tf-idf value increases with the number of times a word appears in a document. This is basically the same idea as before in Section 14.2 where we were counting words using bag-of-words. The second one is that the tf-idf value decreases if the word appears very frequently in the corpus (i.e. the collection of all text documents). The basic idea is that if a word (for example "and") appears all the time throughout all documents, it probably is not that important and should receive a lower weight. We cover several code examples in **Chapter 16 starting on page 185**.

Although the main point of tf-idf is to weigh the words in a more informative way, you can also use it to **remove unimportant words**. One way to do this is to get rid of words with a very low tf-idf score.

## 14.4 One-hot Encoding (Word-level)

The basic idea behind one-hot encoding is to represent each word in a sentence or document by a binary vector, where the vector has as many elements as there are unique words in the vocabulary. In the one-hot encoded vector, the element corresponding to the given word is set to 1, while all other elements are set to 0. This results in a sparse matrix where each row represents the one-hot encoded vector for a word in the sentence or document, and each column corresponds to a unique word from the vocabulary. One-hot encoding is a simple and efficient way to encode categorical data, such as words in text, and it serves as the foundation for more advanced NLP techniques, like word embeddings and deep learning models.

While one-hot encoding can represent the same information as a binary vector (i.e. the presence or absence of a word in a text document), it can go beyond that. Specifically, one-hot encoding can also be used to represent the sequence of words, allowing the model to capture the order in which the words appear, even if some words are repeated.

```
# Example sentence:
sentence = \
    'Pandas munch on bamboo while koalas snack on eucalyptus leaves.'
# Tokenize the sentence into words.
words = sentence.lower().split()
# Get unique words.
unique_words = sorted(set(words))
# Create a dictionary to map unique words to their indices.
word_to_index = {word: index for index, word in enumerate(unique_words)}
# Perform one-hot encoding.
one_hot_encoded_words = []
for word in words:
    encoding = [0] * len(unique_words)
    encoding[word_to_index[word]] = 1
    one_hot_encoded_words.append(encoding)
# Print the one-hot encoded vectors.
for word, encoding in zip(words, one_hot_encoded_words):
    print(f'{word}: {encoding}')
```

The output is as follows, representing the sentence as a sequence of one-hot encoded vectors (or as a matrix if you concatenate the vectors on top of each other). You can observe that one-hot encoding can capture the order in which the words appear.

```
pandas:     [0, 0, 0, 0, 0, 0, 1, 0, 0]
munch:      [0, 0, 0, 0, 1, 0, 0, 0, 0]
on:         [0, 0, 0, 0, 0, 1, 0, 0, 0]
bamboo:     [1, 0, 0, 0, 0, 0, 0, 0, 0]
while:      [0, 0, 0, 0, 0, 0, 0, 0, 1]
koalas:     [0, 0, 1, 0, 0, 0, 0, 0, 0]
snack:      [0, 0, 0, 0, 0, 0, 0, 1, 0]
on:         [0, 0, 0, 0, 0, 1, 0, 0, 0]
eucalyptus: [0, 1, 0, 0, 0, 0, 0, 0, 0]
leaves.:    [0, 0, 0, 1, 0, 0, 0, 0, 0]
```

In production you could use the `DictVectorizer` class from the `sklearn.feature_extraction` module to perform the one-hot encoding in a computationally more efficient way.

## 14.5 Word Embeddings (Word-level)

Another way to transform text to vectors are so-called **word embeddings**. The idea is to use a mathematical embedding from a space with one dimension per word (i.e. a very high-dimensional space such as one-hot encoding) to a continuous vector space with much lower dimensionality. The goal is to have a relatively dense representation to avoid issues with sparsity and to have a higher information denseness. **Words sharing common contexts should be located in close proximity to another in that space.**

A popular example is **word2vec**, which is an unsupervised model that transforms words based on their co-occurrence into a higher-dimensional vector space. For example, the vector for "king" minus the vector for "man" plus the vector for "women" should be close to the vector for "queen." Other popular models are **GloVe** and **FastText** which are both similar to word2vec. We discuss word embeddings in more detail in **Chapter 20 starting on page 205**.

## 14.6 Document Embeddings (Document-level)

Document embeddings are a powerful natural language processing (NLP) technique that involves representing entire documents, such as sentences, paragraphs, or even longer texts, as fixed-size vectors in a continuous vector space. Building upon the concept of word embeddings, which capture semantic and syntactic relationships between individual words in a low-dimensional space, document embeddings aim to **represent the overall meaning and context of a document in a compact and efficient manner**.

While word embeddings focus on single words, document embeddings capture the global context and complex interactions between words in a given text. This enables more effective analysis and comparison of documents, as well as improved performance in various NLP tasks, such as text classification, semantic similarity, clustering, and information retrieval. Several techniques have been proposed for generating document embeddings, including approaches based on BERT, GloVe + Aggregation, and the Universal Sentence Encoder (USE), among others.

**BERT**, a deep learning model based on the transformer architecture, has become a popular choice for generating document embeddings due to its powerful bidirectional and contextualized representations. **GloVe**, which captures global co-occurrence information, can be combined with **aggregation** methods like averaging or summation to create document embeddings. This approach is particularly useful for simpler or resource-constrained tasks because it is computationally less expensive and can be easily implemented compared to more complex models like BERT. The **Universal Sentence Encoder**, on the other hand, provides an efficient and versatile option for generating embeddings for various text lengths, making it particularly suitable for tasks that require quick embeddings generation, such as real-time semantic search, and tasks with variable input lengths, ranging from single sentences to full paragraphs.

These models utilize different approaches to learn and represent the structure and meaning of documents, offering valuable tools for processing and understanding textual data in a wide range of use cases, such as sentiment analysis, document summarization, question-answering systems, and recommendation engines.

# Chapter 15

# Bag-of-Words (BoW)

The bag-of-words (BoW) model is probably the simplest way of converting text into numbers, i.e. into something the computer can more easily understand. We have already introduced this idea in Section 14.2 starting on page 175. The basic idea is to simply count the words in a given text document. If some words appear more frequently than others it might indicate that these words bear a particular significance for the meaning of this piece of text.

## 15.1 list.count and BoW

A simple word counter can be constructed using `list.count`, which is similar to `str.count` discussed in Section 12.1 starting on page 153. Here we assume that we already have converted a text document into a sequence of words (or "tokens") stored in a list.

```
>>> tokens = ['python', 'hello', 'python']
>>> {tk: tokens.count(tk) for tk in set(tokens)}
{'hello': 1, 'python': 2}
```

## 15.2 Dict and BoW

In this example we use a Python dict to store the word counts. We cycle through all words (or "tokens") and update the dictionary manually by increasing the word count.

```
tokens = ['python', 'hello', 'python']
wc = {}                         # Initialize word count dict.
for tk in tokens:
    wc[tk] = wc.get(tk, 0) + 1
```

This code results in the content of `wc` being

```
{'python': 2, 'hello': 1}
```

## 15.3 Defaultdict and BoW

This BoW example is similar to the one using a dict, but this time we use a defaultdict and initialize it with zero (i.e. a call to the `int()` function). We have mentioned the defaultdict class before on page 76.

```
from collections import defaultdict
tokens = ['python', 'hello', 'python']
wc = defaultdict(int)    # Initialize the word count.
for tk in tokens:
    wc[tk] += 1
```

This code results in the content of `wc` being

```
defaultdict(<class 'int'>, {'python': 2, 'hello': 1})
```

## 15.4 "Counter" Class and BoW

A simple way of calculating token frequencies is by using the `Counter` class from the `collections` module (see also the discussion on page 75). Recall that a `Counter` object has a similar structure as a dictionary. The code below can be found in the file `code-nltk-collections-bag-of-words.py` on the course website.

```
# This file shows simple examples how to calculate bag-of-words.
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from collections import Counter
text = '''The cat is in the box.
          The cat likes the box.
          The box is over the cat.'''
# Simple example without any preprocessing.
c = Counter(word_tokenize(text))
c                                   # View the counts.
c.values()                          # Only the word count numbers.
c.most_common(2)                    # The two most common words.
list(c.elements())                  # All the words.
# Convert to lowercase and only keep alphabetic (remove punctuation etc.)
tokens = [w for w in word_tokenize(text.lower()) if w.isalpha()]
# Keep words that are not stopwords (i.e. remove stopwords).
no_stops = [t for t in tokens if t not in stopwords.words('english')]
Counter(no_stops).most_common(2)
# Now we lemmatize the words (similar to stemming).
wnl = WordNetLemmatizer()
lemmatized = [wnl.lemmatize(t) for t in no_stops]
Counter(lemmatized).most_common(2)
```

```
# An alternative that is native to NLTK but actually an extension of
# the `Counter` class. Another advantage is that it can plot easily
# using the matplotlib package.
from nltk.probability import FreqDist
fd = FreqDist(lemmatized)
fd                            # Looks like `Counter`.
fd.plot(10)                   # Specify how many words to plot at most.
```

## 15.5 NLTK and BoW

The NLTK library provides the `FreqDist` class, which is an extension of the `Counter` class (See Section 15.4 starting on page 180). It can be used to compute the count of each unique word. Compared to the `Counter` class, `FreqDist` provides extra methods tailored for natural language processing tasks, such as `hapaxes()` (returns a list of elements that occur only once), `most_common()` (returns a list of most common elements and their counts), `plot()` (visualizes the frequency distribution), and `freq()` (returns the normalized frequency of a given element).

```
from nltk import word_tokenize, FreqDist
document = 'I love this movie. The movie is excellent.'
tokens = word_tokenize(document)
FreqDist(tokens)
```

The output looks as follows:

```
FreqDist({'movie': 2, '.': 2, 'I': 1, 'love': 1, 'this': 1,
          'The': 1, 'is': 1, 'excellent': 1})
```

## 15.6 Scikit-learn Package and BoW

The scikit-learn (sklearn) package is the standard for machine learning in Python. However, it also has functionality for dealing with texts. The code can be found in the file `code-sklearn-bag-of-words.py` on the course website.

```
# This code uses scikit-learn to calculate bag-of-words
# (BOW). `CountVectorizer` implements both tokenization and occurrence
# counting in a single class.
from sklearn.feature_extraction.text import CountVectorizer
# Instantiate a class that converts text documents to numeric vectors.
v = CountVectorizer(stop_words='english')
# Minimal corpus for illustration.
cp = [
    'The␣sun␣filled␣the␣sky␣with␣a␣deep,␣deep␣red␣flame.',
    'The␣waves␣rolled␣along␣the␣shore␣in␣a␣graceful,␣gentle␣rhythm',
    'The␣winter␣sun␣touched␣the␣painting',
    'The␣painting␣was␣a␣field␣of␣flowers']
```

```
# Learn the vocabulary dictionary and return the document-term
# matrix. Tokenize and count word occurrences.
bow = v.fit_transform(cp)
# Each term found by the analyzer during the fit is assigned a unique
# integer index corresponding to a column in the resulting
# matrix. This interpretation of the columns can be retrieved as
# follows.
bow.toarray()                  # Print the document-term matrix.
bow.A                          # Using the `A` property.
v.get_feature_names_out()      # Which term is in which column?
# Inverse mapping from feature name to column index.
v.vocabulary_.get('painting')
# Mapping of documents to BOW. Words that were not seen in the
# training corpus are ignored.
v.transform(['The dry painting.']).toarray()
```

## 15.7 Gensim Package and BoW

Gensim is a very powerful NLP package that can, among many other things, also do bag-of-words. In the example below we use an in-memory corpus, i.e. the whole corpus is held in computer memory. On the other hand, to have a more memory-friendly implementation, we will discuss an alternative way to compute BoW in gensim in Section 20.5.1 starting on page 208.

The following code can be found in the file code-gensim-bag-of-words.py on the course website.

```
# Bag-of-words using gensim.
from gensim.corpora.dictionary import Dictionary
from nltk.tokenize import word_tokenize
# This is an example corpus consisting of movie reviews. You can think
# of each movie review as a separate text document.
cp = \
    ['The movie was about a spaceship and aliens. The movie is wonderful!',
     'I really liked the movie. More people should go see it.',
     'Awesome action scenes, but boring characters.']
# Create tokenized corpus. Very basic preprocessing. Usually you would
# do more work here.
cp = [word_tokenize(doc.lower()) for doc in cp]
cp = [[token for token in doc if token.isalnum() and len(token) > 1]
      for doc in cp]

# Pass to gensim `Dictionary` class. This assigns to each token
# (e.g. word) a unique integer ID. Later on we will just work with
# those IDs instead of the tokens directly because it is
# computationally easier to handle (there is a one-to-one mapping
# between both, so we are not losing any information). The reason why
# we use a dictionary is that it gives us a list of words we are
# interested in examining further. If a word is not in the dictionary
# but occurs in a document, it will be ignored by gensim.
d = Dictionary(cp)
d.token2id  # Like dict(d); show mapping between tokens and their IDs.
d.token2id.get('awesome')        # What's the ID for 'awesome'?
d.get(0)                         # Which token has ID=0?
d.dfs # In how many documents does each token appear? (Document frequency).

# For a single document, we can now calculate the token frequencies
# using the dictionary we just created. "Calculating token
# frequencies" means we're counting words.
d.doc2bow(cp[2])

# Next, using the dictionary we just created, we build a gensim
# corpus, which is just a bag-of-words representation of the original
# corpus. This is a nested list (a list of lists), where each list
# corresponds to a document. Inside each list we have tuples in the
# form
#
# (token_ID, token_frequency).
#
# So all we are really doing here is counting words.
cp = [d.doc2bow(doc) for doc in cp]
# This gensim corpus can now be saved, updated, and reused using tools
# from gensim. The dictionary can also be saved and updaed as well,
# e.g. if we need to add more words later on.

# Print the first three token IDs and their frequency counts from the
# first document.
cp[0][:3]
# For the first document, sort the tokens according to their
# frequency, with the most frequent tokens coming first.
sorted(
    cp[0],                  # First document.
    key=lambda x: x[1],  # Sort by token_frequency (second element).
    reverse=True)         # Most frequent first.
```

# Chapter 16

# tf-idf Weighting

Note that we have already introduced tf-idf from a conceptual point of view in Section 14.3 starting on page 175. In contrast, in this chapter we go into greater detail and demonstrate how to actually implement tf-idf using Python programming.

Let's suppose you're working in a bank and the term "loan" comes up all the time in all your text documents. In this case, "loan" might not be very meaningful since you're *always* dealing with loans. So you might want to down-weight these kinds of words. tf-idf can help in this situation. It stands for "term-frequency—inverse document frequency" and helps to put more weight on words that are seemingly more important (and less weight on words that are not, e.g. "loan" in the example above.) If a word appears very frequently in many of your documents (such as "loan" in the example above), it will receive a lower weighting in tf-idf.

Mathematically the idea of tf-idf translates to

$$\text{tfidf}(t,d,D) = \text{tf}(t,d) \times \text{idf}(t,D),$$

where $\text{tf}(t,d) = f_{t,d}$ is typically the number of times the term $t$ occurs in document $d$ (same as in bag-of-words) and

$$\text{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

is the logarithmically scaled inverse fraction of the documents that contain the word, where $N$ is the total number of documents in the corpus $D$ (so $N = |D|$) and $|\{d \in D : t \in d\}|$ is the number of documents where the term $t$ appears in.

Taken together, you can see that if in the example above "loan" appears in many documents (so $|\{d \in D : t \in d\}|$ would be relatively close to $N$), it means that idf is closer to zero and thus tfidf is closer to zero as well. In total, "loan" would receive a lower weighting.

There are several variants in the way tf-idf can be calculated, although the version presented above is the most important baseline case. Another popular way $\text{tf}(t,d)$ is computed is by $\text{tf}(t,d) = f_{t,d} / \sum_{\hat{t} \in d} f_{\hat{t},d}$, which is the relative frequency of term $t$ within document $d$.

Let us consider a simple numerical example. Suppose that $tf = 8$ and $idf = 0.3$. To compute the tf-idf value, you multiply both and obtain $8 \times 0.3 = 2.4$.

## 16.1 Scikit-learn and tf-idf

Here is another example showing the usage of SciKit-Learn (or sklearn for short) to compute tf-idf. Although sklearn is most often used for machine learning, it can also calculate tf-idf (or BoW as we have seen in Section 15.6). The code can be found in the file `code-sklearn-tf-idf.py` on the course website.

```
# This code uses scikit-learn to calculate tf-idf. The code is very
# similar to the one used with `CountVectorizer` (which give BoW).
from sklearn.feature_extraction.text import TfidfVectorizer
# Here we instantiate the `TfidfVectorizer` class, which allows us to
# convert text documents to numeric vectors.
#
# `max_df` tells sklearn to ignore terms that have a document
# frequency higher than this threshold when building the
# vocabulary.
#
# `norm=None` avoids normalizing each document vector to a norm of
# one; it means we are not adjusting for document length (i.e. number
# of words in the document) in the example below. This is done for
# better illustration of how tf-idf is computed. For production, it is
# usually better to enable normalization, e.g. `norm='l2'`.
v = \
    TfidfVectorizer(
        stop_words='english',
        max_df=0.9,
        norm=None)
cp = [                          # Minimal corpus for illustration.
    'The sun filled the sky with a deep, deep red flame.',
    'The waves rolled along the shore in a graceful, gentle rhythm',
    'The winter sun touched the painting',
    'The painting was a field of flowers']
# Learn the vocabulary dictionary and return the document-term
# matrix. Tokenize and count word occurrences.
tfidf = v.fit_transform(cp)
# Each term found by the analyzer during the fit is assigned a unique
# integer index corresponding to a column in the resulting
# matrix. This interpretation of the columns can be retrieved as
# follows.
tfidf.toarray()                 # Print the document-term matrix.
tfidf.A                         # Using the `A` property.
v.get_feature_names_out()       # Which term is in which column?
# Inverse mapping from feature name to column index.
v.vocabulary_.get('painting')
# Mapping of documents to their tf-idf scores. Words that were not
# seen in the training corpus are ignored.
v.transform(['The dry painting.']).A
```

## 16.2 Gensim and tf-idf

Here we have a basic example that shows how you can calculate tf-idf using the gensim package. This code uses an in-memory corpus, meaning the whole corpus is held in computer memory. An alternative implementation using streaming data can be obtained by adapting the concepts presented in Section 20.5 starting on page 208. Streaming data allows for larger corpora to be processed, as the entire corpus is not held in computer memory.

The following code can be found in the file `code-gensim-tf-idf.py` on the course website. Please keep in mind that there are different variants by which tf-idf values can be calculated. So the following code example may yield different tf-idf values than the previous example did using SciKit-Learn.

```
# This script illustrates how to use tf-idf with gensim.
from nltk.tokenize import word_tokenize
from gensim.corpora.dictionary import Dictionary
from gensim.models.tfidfmodel import TfidfModel
corpus = \
    ['The movie was about a spaceship and aliens. It is wonderful!',
     'I really liked the movie. More people should go see it.',
     'Awesome action scenes, but boring characters.']
tokenized_corpus = [word_tokenize(doc.lower()) for doc in corpus]
d = Dictionary(tokenized_corpus)
bowcorpus = [d.doc2bow(doc) for doc in tokenized_corpus]
# Instantiate a `TfidfModel` from BoW corpus. Specifically, here the
# idf values are computed and stored in the `tfidf` object.
tfidf = TfidfModel(bowcorpus)
# Compute the tf-idf values of the first document. This is a shortcut
# for calling the `tfidf.__getitem__(bowcorpus[0])` magic method.
tfidf_weights = tfidf[bowcorpus[0]]
tfidf_weights[:5]                    # First five weights (unordered).
 # Print top five weighted words.
sorted_tfidf_weights = \
    sorted(
        tfidf_weights,
        key=lambda x: x[1],
        reverse=True)
for term_id, weight in sorted_tfidf_weights[:5]:
    print(d.get(term_id), weight)
```

In the example above, gensim makes use of the `__getitem__()` magic method to enable indexing of an object of class `TfidfModel`. Magic methods are discussed in Section 6.6.6 starting on page 63.

# Chapter 17

# Storing and Analyzing the Text Corpus In Numerical Format

## 17.1 Representing Text as Numbers

Building on the numerical representation of text (Chapter 14), we can now explore some applications, i.e. what to do with those numbers in the context of NLP. For the purpose of this chapter, let's focus on document-level vectors, i.e. each vector represents an entire text document. Often each element of the vector corresponds to a specific word, e.g. in BoW or tf-idf. On the other hand, for document embeddings such as doc2vec, a one-to-one relation between vector elements and corresponding words does no longer exist.

If we now look at the set consisting of all documents, we have the **text corpus**. Given the document-level vectors (in numerical format), we will explore in this chapter how we can represent the corpus in numerical format as well. In addition, we will discuss a few ways we can analyze the numerical representation of the text corpus, e.g. using machine learning.

## 17.2 TDM, DTM, and Tidy Text

A text corpus can be represented in several ways, three of which we will discuss in this section. The terminology used in this section (e.g. "term-document matrix") assumes that there is a direct mapping between words ("terms") and vector elements, e.g. as in BoW or tf-idf. However, we can use all representations for document embeddings as well (e.g. doc2vec), even though in general document embedding models are not designed to have have a direct mapping where each vector element represents only one specific word.

Depending on whether we stack the document-level vectors next to each other or on top of each other (after transposing them), we will obtain a matrix of different form. If we stack the vectors next to each other, we obtain a so-called **term-document matrix (TDM)** where the rows correspond to terms and the columns correspond to documents. On the other hand, if we transpose the vectors and stack them on top of each other, we have a so-called **document-term matrix (DTM)** where the rows correspond to documents and the columns correspond to terms.

Both matrices are mathematically equivalent to each other. For example, if you transpose a term-document matrix, you will obtain a document-term matrix (and vice versa).

Another, maybe more convenient way of representing a corpus is the **tidy text** format which represents the document as a table. The basic idea is that for each document-word pair, you write down the word score (e.g. word count, tf-idf, etc.) next to it, like so:

```
Document | Word | Score
---------|------|----------
doc1     | this | 5
doc1     | is   | 8
doc1     | it   | 3
doc2     | this | 7
doc2     | it   | 1
```

The tidy representation of text data can be very practical because it avoids sparsity. Document-term matrices are often sparse (especially if you are using BoW), which means there are a lot of entries with zeros. Either you use special programming techniques (which can be cumbersome and sometimes slow) to deal with these sparse matrices, or you use a regular matrix representation and run out of computer memory very quickly (which is bad in general). The tidy text format representation is much more memory-friendly. Notice how **the word "is" does not occur in document doc2 and thus does not use up any space in the tidy text format**. Furthermore, it does not require any special programming techniques; after all, it's just a dense (i.e. non-sparse) table.

## 17.3 Analyzing the Numerical Text Representation of the Text Corpus

Having represented the text corpus in TDM, DTM, tidy, or another numerical format, we can now analyze the corpus using some of the techniques discussed in earlier chapters, e.g. document similarity or machine learning. A few example applications are as follows:

1. You can compare the **similarity** of two documents (or two words) using the cosine similarity measure. For further explanations about cosine similarity see Chapter 18 starting on page 193 and Section 19.1.4 starting on page 201.

2. You can find out which documents (or which words) form similar **clusters**. For example, if you are looking at newspaper articles, you might find three clusters (e.g. using $k$-means clustering), and upon closer inspection you might discover that these clusters roughly correspond to politics, business, and health. The point here is that a priori you don't know the clusters and learn about them from the data. For further information about clustering and unsupervised learning in general please see Chapter 11 starting on page 115.

3. You can **classify** documents into different categories. For example, you could classify tweets from Twitter into "positive" or "negative" sentiment about the stock

market. This approach assumes you have a way to obtain positive/negative labels for some tweets, so you can use these labels to train your classifier. For further information about classification and related techniques from supervised learning, please see Chapter 11 starting on page 115.

# Chapter 18

# Document Similarity

Often you would like to find out how similar two documents are to each other. To this end, the cosine similarity allows you compute a number between $-1$ and 1, which indicates how similar two text documents are to each other. A value of $-1$ means the documents are the exact opposite, a value of 0 indicates “orthogonality” or “decorrelation” (meaning that there is no connection between the documents), and a value of 1 means the documents are exactly the same.

However, the cosine similarity measure does not directly work on text. Instead, it works on real-valued vectors that represent the text documents. For example, two text documents could be represented by two BoW vectors. The cosine similarity can be applied to any two vectors, so it will work with BoW, tf-idf, or any other method that turns text documents into vectors.

The basic idea is to compare the similarity of two documents by (1) converting them to two vectors and (2) comparing the angles between those two vectors. However, since it is computationally easier to calculate the cosine of the angle between two vectors, people tend to use the so-called “cosine similarity” instead of the angle.

Mathematically, the following identity holds:

$$a \cdot b = \|a\| \|b\| \cos\theta,$$

where $a$ and $b$ are real-valued vectors, $a \cdot b$ is the dot product, $\|\ldots\|$ is the Euclidean norm, and $\theta$ is the angle between $a$ and $b$. The **cosine similarity** is defined as $\cos\theta$, and is therefore equal to

$$\frac{a \cdot b}{\|a\| \|b\|}.$$

In other words, if $a = (a_1, \ldots, a_d)^t$ and $b = (b_1, \ldots, b_d)^t$, then the cosine similarity is equal to

$$\frac{\sum_{i=1}^{d} a_i b_i}{\sqrt{\sum_{i=1}^{d} a_i^2}\sqrt{\sum_{i=1}^{d} b_i^2}}.$$

The code listed below shows several ways how you can compute the cosine similarity. The code can be found in the file `code-cosine-similarity.py` on the course website.

```
# Cosine similarity.
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Let's assume we have two documents that have been converted to the
# following vectors, e.g. by bag-of-words (BoW), tf-idf, or other
# means (potentially followed by dimensionality reduction such as
# SVD).
docA = np.array([9, 4, 8, 2, 7, 5, 0])
docB = np.array([0, 1, 2, 7, 0, 4, 4])

# We can now "manually" compute the cosine similarity. In the first
# example we compute the norm "by hand" whereas in the second example
# we use the `norm` function from numpy.
np.dot(docA, docB) / (np.sqrt(np.sum(docA ** 2)) * np.sqrt(np.sum(docB ** 2)))
np.dot(docA, docB) / (np.linalg.norm(docA) * np.linalg.norm(docB))


# Alternatively we can use the function `cosine_similarity` from the
# scikit-learn (sklearn) package. It has the advantage that it can be
# applied to two or more vectors at once and can compute the cosine
# similarity between all vector pairs.
cosine_similarity(np.vstack((docA, docB)))
```

A drawback of the cosine similarity measure is that similar documents may show up far away from each other when represented as vectors. This can happen for example with bag-of-words: Words with similar meaning may not be near each other, for example “hello” could be at index 3 while “hi” could be at index 3,000 of the BoW vector. Therefore the cosine similarity will indicate that documents containing these words may be dissimilar. This problem is not limited to BoW, it applies to all vector representations that do not take into account the context of words.

A solution to this problem is the so-called **soft cosine measure (SCM)** that represents an extension of the cosine similarity. The basic idea of SCM is to transform words to word embeddings (such as word2vec or GloVe), and afterwards compute the similarity between the transformed words. Word embeddings represent text by vectors that take into account the context of words. This allows for words that have similar meanings to receive a high similarity score.

The code listed below for computing the soft cosine measure with Gensim can be found in the file `code-gensim-soft-cosine-similarity.py` on the course website.

```
from nltk.corpus import stopwords
from gensim.corpora import Dictionary
from gensim.models import TfidfModel
import gensim.downloader as api
from gensim.similarities import \
    SparseTermSimilarityMatrix, WordEmbeddingSimilarityIndex

# Define a text corpus containing example data. Here each element is a
# separate text document. In this specific example, each text document
# has only one sentence, although the code also works for documents
# containing several sentences.
#
# The text documents are specifically chosen to NOT have any words in
# common. So a cosine similarity measure would have problems detecting
# meaningful similarity. In contrast, the soft cosine measure (SCM)
# will still be able to pick up whether or not two documents are
# indeed similar. Specifically, the first and second document should
# be similar according to SCM, although they do not share the same
# words.
cp = ['Obama␣speaks␣to␣the␣media␣in␣Illinois',
      'The␣President␣greets␣the␣press␣in␣Chicago',
      'Oranges␣are␣a␣tasty␣fruit']

def preprocess(doc):      # Tokenize a document and remove stop words.
    return [w for w in doc.lower().split() \
            if w not in stopwords.words('english')]

cp = [preprocess(doc) for doc in cp]

# Compute bag-of-words (BoW).
d = Dictionary(cp)
cp = [d.doc2bow(doc) for doc in cp]

# Compute tf-idf.
tfidf = TfidfModel(cp)
cp = [tfidf[doc] for doc in cp]

# Load GloVe vectors. These are "word embeddings," i.e. representation
# of words by real-valued vectors. The GloVe model will be stored in
# the folder `gensim-data` in your home folder. Here we use the
# `glove-twitter-25` model for demonstration purposes because it is
# relatively small (around 100MB). For production you could load a
# larger model if you like. See
# https://github.com/RaRe-Technologies/gensim-data for details.
model = api.load('glove-twitter-25')
# Compute the sparse term similarity matrix.
termsim_matrix = \
    SparseTermSimilarityMatrix(
        WordEmbeddingSimilarityIndex(model), # Cosine similarities between word emb
        d,                                   # Dictionary from BoW.
        tfidf)                               # tf-idf model.
# Define a function to compute the soft cosine measure (SCM).
def SCM(docA, docB):
    return termsim_matrix.inner_product(docA, docB, normalized=(True, True))
```

```
# Compute the SCM for several document pairs.
SCM(cp[0], cp[1]) # Should be similar, even though no words in common.
SCM(cp[0], cp[2]) # Should be dissimilar.
```

# Chapter 19

# Dimensionality Reduction

Dimensionality reduction plays a crucial role in natural language processing (NLP) by transforming high-dimensional text data into a lower-dimensional representation that captures the essential semantic information. This process reduces computational complexity, mitigates the curse of dimensionality, and enhances the performance of machine learning algorithms on text data. Some of the most commonly used dimensionality reduction techniques in NLP include latent semantic analysis (LSA) based on singular value decomposition (SVD), latent Dirichlet allocation (LDA), and t-distributed stochastic neighbor embedding (t-SNE). These techniques uncover latent structures and relationships in the text data, allowing for more efficient and effective text analysis, visualization, and modeling.

## 19.1 Latent Semantic Analysis (LSA)

### 19.1.1 What It's All About

One of the most important methods for dimensionality reduction is latent semantic analysis (LSA) based on **singular-value decomposition (SVD)**. With SVD, you look into linear combinations of words that are arranged in such a way as to lose a minimum amount of information. SVD is closely related to principal component analysis (PCA), as SVD can be used to compute the PCA. (Alternatively, PCA can also be done using an eigenvalue decomposition.)

For the following discussion, it is useful to keep the following definition in mind: The matrix $B$ is **orthogonal** if and only if $B^tB = BB^t = I$, where $I$ is the identity matrix. This definition is equivalent to $B^t = B^{-1}$, i.e. a matrix is orthogonal if and only if its transpose is equal to its inverse. In the context of matrix notation, superscript $t$ indicates the **transpose** of a matrix, i.e. $B^t$ is the transpose of $B$.

Suppose your **term-document matrix** is given by the $m \times n$ matrix $M$, which means we have $m$ terms and $n$ documents. So the rows of $M$ correspond to different terms while the columns of $M$ correspond to different text documents. Using a mathematical result called "singular value decomposition" (SVD), it is possible to factorize $M$ into

$$M = U\Sigma V^t,$$

where:

- $U$ is a $m \times m$ orthogonal matrix. Each **row** of $U$ corresponds to a different **term**, and each **column** of $U$ corresponds to a different **concept/topic**.
- $\Sigma$ is a diagonal $m \times n$ matrix with non-negative numbers on the diagonal. Each **diagonal element** can be interpreted as corresponding to a different **concept/topic** in the term-document matrix.
- $V^t$ is the transpose of an orthogonal $n \times n$ matrix $V$. Each **column** of $V^t$ corresponds to a different **document**, and each **row** of $V^t$ corresponds to a different **concept/topic**.

The diagonal entries of $\Sigma$ are called the **singular values** of $M$. Without loss of generality, the singular values are commonly listed in descending order (in which case $\Sigma$ is unique, even though $U$ and $V$ may not always be uniquely determined). The interpretation is that $U$ **captures information about the terms** (spanning a term vector space), $V^t$ **captures information about the documents** (spanning a document vector space), and **both are linked to each other using the concepts/topics encoded in** $\Sigma$.

In practice, you would not have to compute the SVD by hand. Instead, the computer program can create the SVD for you by computing the matrices $U$, $\Sigma$, and $V$. For an example application with programming code see **Section 20.6 starting on page 212**.

For **dimensionality reduction** (or "dimension reduction," as it is sometimes called), you set the smallest singular values to zero as follows:

- You create a modified $k \times k$ matrix $\hat{\Sigma}$ where you keep only the first $k$ diagonal elements and discard the remaining elements.
- **Values of $k$ are usually chosen between 100 and 300 in text analytics applications.** Keep in mind that this is a reasonable number as (i) the number of topics in a corpus is probably not much larger than 100 to 300, and (ii) in typical applications the dimensions of $M$ are larger than 100 to 300.
- You also create the modified matrices $\hat{U} = (U_1, \ldots, U_k)$ and $\hat{V} = (V_1, \ldots, V_k)$, where $U_1, \ldots, U_k$ and $V_1, \ldots, V_k$ correspond to the first $k$ columns of $U$ and $V$, respectively.
- Then you compute the $m \times n$ matrix

$$\hat{M} := \hat{U}\hat{\Sigma}\hat{V}^t.$$

- Note that the same $\hat{M}$ can equivalently be computed by setting the diagonal entries below $k$ in $\Sigma$ to zero, calling this matrix $\hat{\Sigma}_0$, and computing $U\hat{\Sigma}_0 V^t$, i.e. using just the original $U$ and $V$. However, for computational reasons it is usually easier to use $\hat{U}\hat{\Sigma}\hat{V}^t$.

The factorization $\hat{U}\hat{\Sigma}\hat{V}^t$ is called the **truncated SVD**. The matrix $\hat{M}$ still has the same dimensions $m \times n$ as $M$, but $\hat{M}$ now has rank $k$. So $\hat{M}$ is of lower rank than $M$ (assuming that $M$ has rank $> k$, which corresponds to more than $k$ elements on the diagonal of $\Sigma$

being non-zero). Remember that the **rank of a matrix** is the dimension of the vector space spanned by its columns. In this sense, the dimension $k$ of the vector space spanned by the columns of $\hat{M}$ is smaller than the one spanned by $M$. That's why we call the calculations "dimensionality reduction."

### 19.1.2 Numerical Examples

Consider the following numerical examples. Suppose a $3 \times 3$ matrix $M$ is given whose SVD is specified as follows by $U$, $\Sigma$,and $V$. Note how the singular values of $\Sigma$ are shown in descending order on its diagonal.

$$U = \begin{pmatrix} -0.32 & 0.65 & -0.69 \\ -0.75 & -0.62 & -0.24 \\ -0.58 & 0.43 & 0.69 \end{pmatrix}$$

$$\Sigma = \begin{pmatrix} 15.75 & 0 & 0 \\ 0 & 5.11 & 0 \\ 0 & 0 & 3.27 \end{pmatrix}$$

$$V = \begin{pmatrix} -0.42 & 0.91 & 0.07 \\ -0.57 & -0.21 & -0.79 \\ -0.71 & -0.37 & 0.60 \end{pmatrix}$$

Given this SVD, we know that $M$ satisfies

$$M = U\Sigma V^t = \begin{pmatrix} 4.98 & 3.96 & 1.00 \\ 2.02 & 8.02 & 9.09 \\ 5.99 & 2.96 & 7.03 \end{pmatrix}$$

For $k = 2$, you calculate the truncated SVD as follows. Keep in mind that in the last part you need to transpose $\hat{V}$ in order to calculate $\hat{M}$. Observe that $\hat{M}$ is still kind of similar to the original $M$, although we have already removed some information.

$$\hat{U} = \begin{pmatrix} -0.32 & 0.65 \\ -0.75 & -0.62 \\ -0.58 & 0.43 \end{pmatrix}$$

$$\hat{\Sigma} = \begin{pmatrix} 15.75 & 0 \\ 0 & 5.11 \end{pmatrix}$$

$$\hat{V} = \begin{pmatrix} -0.42 & 0.91 \\ -0.57 & -0.21 \\ -0.71 & -0.37 \end{pmatrix}$$

$$\hat{M} = \hat{U}\hat{\Sigma}\hat{V}^t = \begin{pmatrix} 5.14 & 2.18 & 2.35 \\ 2.08 & 7.40 & 9.56 \\ 5.84 & 4.75 & 5.67 \end{pmatrix}$$

Or for $k = 1$ the truncated SVD is as follows. Observe that $\hat{M}$ now becomes more dissimilar to the original $M$ as we have removed more information, although some resemblance to $M$ still remains.

$$\hat{U} = \begin{pmatrix} -0.32 \\ -0.75 \\ -0.58 \end{pmatrix}$$

$$\hat{\Sigma} = \begin{pmatrix} 15.75 \end{pmatrix}$$

$$\hat{V} = \begin{pmatrix} -0.42 \\ -0.57 \\ -0.71 \end{pmatrix}$$

$$\hat{M} = \hat{U}\hat{\Sigma}\hat{V}^t = \begin{pmatrix} 2.12 & 2.87 & 3.58 \\ 4.96 & 6.73 & 8.39 \\ 3.84 & 5.21 & 6.49 \end{pmatrix}$$

### 19.1.3 Interpretation

You can interpret this new $\hat{M}$ matrix as a new term-document matrix that now represents a latent semantic space containing the most important information about the documents in the corpus. Mathematically, you can think of SVD as an **initial rotation coupled with scaling**, or you could also think of it as a **projection into a lower-dimensional space**.

### 19.1.4 Application in Text Analytics

There are a few things to note about SVD. First of all, you can apply the SVD for example on a term-document matrix (TDM) containing raw word counts (i.e. **BoW**) or weighted word counts (e.g. **tf-idf**). Moreover, SVD can also be applied to **co-occurrence matrices**, and even dense word embeddings like **word2vec** or **GloVe**. Second, it is important to realize that SVD is **unsupervised**, i.e. no human input is necessary for SVD to reduce the dimensionality of the text documents.

To illustrate SVD, let's assume for example that you are looking at customer reviews of cars. You might see that reviews using the term "gas-mileage" also use "economy" a lot. Or reviews using "reliability" also use the term "defects" a lot. So there are two dimensions, the first one being "gas-mileage/economy" and the second one being "reliability/defects." SVD would ideally pick up these two dimensions and realize that words within each dimension are referring to the same thing. It is kind of like using synonyms, but the difference is that this structure is picked up from the whole collection of text documents and no human input is necessary.

The basic idea of SVD is to pick up **latent semantic structure**, which means that there is some structure about the meaning of words hidden in the word counts that you crystallize with SVD. For this reason, any analysis that you perform after applying the SVD (the SVD is used to extract the latent semantics) is also called **latent semantic analysis (LSA)** or **latent semantic indexing (LSI)**. In addition to the following paragraphs, we discuss LSA in more detail with **code examples in Section 20.6 starting on page 212**.

For example, for LSA, you could look at **cosine similarity** (Chapter 18) measures to determine how similar certain documents are to each other (**document similarity**). Let's say you have identified a few newspaper articles that were published on days before a big drop in the stock market occurred. If all the sudden you find many other articles being published that are very similar according to the cosine similarity measure, you should get worried that another drop in stock prices might be around the corner.

Another application is to look at **term similarity**. It is similar to document similarity, but instead of looking at the rows of a document-term matrix you look at its columns (e.g. using cosine similarity). It can tell you how similar different terms are to each other. For example, you could use it to find terms with similar meaning.

Finally, you can also use LSA for **document retrieval**, similar to what Google does. You enter a query, which is viewed as a short document and projected into the latent semantic space generated by SVD. Then you can find the documents that are most similar to that query using document similarity, e.g. cosine similarity.

Keep in mind that **in order to work with document similarity measures such as the cosine similarity, you actually do not need to use SVD**. Instead, in principle you can apply the cosine similarity measure directly to a BoW or tf-idf term-document matrix (or document-term matrix). However, in that case it would not be called LSA or LSI because you are ignoring the latent semantic structure of the documents.

## 19.2 Topic Models

Another approach for dimensionality reduction is **topic modeling**. The basic idea is to look at co-occurrence patterns of terms corresponding to semantic topics in a collection of documents. Each document is thought of as a mixture of a number of topics (e.g. a “cats” topic and a “dogs” topic), and each word can be attributed to one of these topics with some probability. For example, “milk,” “meow,” and “kitten” would be cat-related with high probability, while “puppy,” “bark,” and “bone” would be dog-related with a high probability. It is also possible that a word is related to several topics, e.g. the word “salt” might be from a cooking-related topic (with, say 90% probability) or from a mining-related topic (with, say, 10% probability).

The most popular topic modeling approach is **latent Dirichlet allocation (LDA)**. Here we discuss the conceptual idea of LDA, while we show examples of programming code in **Section 20.7 starting on page 214**. (By the way, you should not confuse it with linear discriminant analysis, which is also abbreviated as “LDA.”) LDA is an unsupervised machine learning model. LDA is a generalization of **probabilistic latent semantic analysis (PLSA)** (also known as **probabilistic latent semantic indexing (PLSI)**), which fulfills a similar purpose. Other relatively popular topic models are the **hierarchical Dirichlet process (HDP)** and **random projections/indexing**.

LDA (or topic modeling in general) is often associated with dimensionality reduction. The goal is similar to that of latent semantic analysis (LSA), which is to take BoW or tf-idf (which is typically high-dimensional) and convert it into another representation that largely retains the information content but has a much lower dimension. The key benefit is that you try to extract and compress the relevant information and get rid of potential noise.

While both LDA and LSA can be used for dimensionality reduction, they have very different approaches. While LSA uses ideas from linear algebra (i.e. a matrix factorization), LDA on the other hand uses a probabilistic approach.

The basic of LDA idea is to model the conditional probability that the author of the document in question is using a given term, provided that he is writing about a specific topic. Because of its probabilistic nature, it is possible that a given word comes from several potential topics. For example, the word “milk” might be from a cow-related topic or from a cat-related topic.

**The dimensionality reduction then is based on the identified topics.** You could represent a document not by its terms, but directly by its topic distribution. For example, you can represent document $d$ by its topic distribution

$$p(z_i|d), \qquad i = 1,\dots,k,$$

where the $z_i$’s represent the $k$ topics that are identified in the corpus. So you would convert each document $d$ into a vector $\hat{d}$ of length $k$:

$$\hat{d} := \begin{pmatrix} p(z_1|d) \\ \vdots \\ p(z_k|d) \end{pmatrix}$$

The vector $\hat{d}$ summarizes the probabilities that text document $d$ covers the topics $(z_1,\ldots,z_k)$. The sum of all elements in the vector equals 1.

The model learns the topic distribution for each document and the word distribution for each topic by analyzing the **co-occurrence patterns of words across documents**. Through this process, LDA can uncover the underlying thematic structure in the text data and provide a compact representation of documents

The length of the vector representing a document in LDA depends on the number of topics chosen for the model. As we have seen, the vector length is equal to the number of topics, and each element in the vector corresponds to the probability that the document covers a specific topic.

There is no universally optimal number of topics, as the ideal length of the vector depends on the specific dataset and use case. Generally, the number of topics is chosen based on domain knowledge, prior assumptions, or through experimentation, such as by evaluating the coherence or interpretability of the topics generated by the model.

For instance, if you choose to model your text data with 10 topics, then the length of the vector representing each document will be 10. The number of topics typically ranges from a few to several hundred, depending on the size and complexity of the text corpus. In practice, it is common to test multiple models with different numbers of topics and choose the one that provides the most meaningful and coherent topic representation for the given problem.

## 19.3 t-SNE

Finally, even if you end up with only a few topics (say, 20), it is still difficult to understand and visualize them. One way to further reduce dimensionality is the $t$-distributed stochastic neighbor embedding (**t-SNE**), which allows you to reduce the dimensions to, say, two, and then plot them for easier visualization and understanding.

For illustration, the following example is based on the Sklearn documentation. It takes four data points with three dimensions as input, and returns four data points transformed to two dimensions.

```
>>> import numpy as np
>>> from sklearn.manifold import TSNE
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
>>> X
array([[0, 0, 0],
       [0, 1, 1],
       [1, 0, 1],
       [1, 1, 1]])
>>> TSNE(n_components=2).fit_transform(X)
array([[ 415.623     ,   -0.99826646],
       [ 193.7694    ,   96.35017   ],
       [ 318.27267   , -222.85101   ],
       [  96.41912   , -125.502556  ]], dtype=float32)
```

# Chapter 20

# LSA, Topic Models, and Word Embeddings in Gensim

## 20.1 Overview of Gensim

- The Gensim package started in 2008 and its name is short for "generate similar," i.e. one of the initial design goals was to generate a list of the most similar documents for a given document.
- Gensim is very versatile, e.g. we have used it before for **bag-of-words** in Section 15.7 starting on page 182 or for **tf-idf** in Section 16.2 starting on page 187. **Please take a look at these earlier examples to refresh your understanding of how gensim represents tokens (e.g. words) by integer IDs.**
- Gensim can do much more than BoW and tf-idf. It is a very popular Python library for dealing with LSA, topic models, and word embeddings.
- In Gensim, a document is usually represented by the features extracted from it, not by its "surface" string form. This means that it is up to you how you get the features. This is intentional, since there are a lot of ways to process documents and the creator of Gensim decided not to constrain it in this regard.

## 20.2 In-Memory Corpus

An easy way to store a text corpus is a list of strings, with each list entry representing a different text document. For example:

```
documents = \
    ["Human machine interface for lab abc computer applications",
     "A survey of user opinion of computer system response time",
     "The EPS user interface management system",
     "System and human system engineering testing of EPS",
     "Relation of user perceived response time to error measurement",
     "The generation of random binary unordered trees",
     "The intersection graph of paths in trees",
     "Graph minors IV Widths of trees and well quasi ordering",
     "Graph minors A survey"]
```

The advantage is that this corpus is conceptually easy to understand. The disadvantage is that the whole corpus is held in computer memory, which can be a problem if the corpus grows big. We have discussed how to compute BoW using an in-memory corpus in Section 15.7 starting on page 182 or tf-idf in Section 16.2 starting on page 187. On the other hand, we discuss a more memory-efficient alternative using streaming data in Section 20.5 starting on page 208.

## 20.3 Saving and Loading a Corpus

Often you have created a corpus and would like to save it to disk. Or you would like to load it from disk. The following code illustrates how this can be done in gensim. Here we assume the corpus is already available in vector form, i.e. there is a token ID (representing the word/token) and a corresponding numerical value, e.g. from BoW or tf-idf. The code below can be found in the file `code-gensim-010-save-corpus-to-file.py` on the course website.

```
from gensim import corpora

# Create a very simple corpus (just a list) consisting of two
# documents. The first document has one word with token ID of 1 and
# value of 0.5 while the second document is empty.
corpus = [[(1, 0.5)], []]

# Save corpus in Matrix Market format. This is a relatively popular
# format.
corpora.MmCorpus.serialize('data-gensim-corpus.mm', corpus)

# Save corpus in SVMlight format.
corpora.SvmLightCorpus.serialize('data-gensim-corpus.svmlight', corpus)

# Save corpus in Blei's LDA-c format.
corpora.BleiCorpus.serialize('data-gensim-corpus.lda-c', corpus)

# Save corpus in GibbsLDA++ format
corpora.LowCorpus.serialize('data-gensim-corpus.low', corpus)

# You can also read the corpus back into Python. Here you load a
# corpus iterable from a Matrix Market file. We then print it in a
# couple of different ways. The first way just shows some meta
# information about the corpus (after all, it's a stream). The second
# way loads it entirely into memory and prints it. The third way
# prints it one document at a time in a memory-friendly way.
corpus = corpora.MmCorpus('data-gensim-corpus.mm') # An iterable.
print(corpus)
print(list(corpus))
for doc in corpus:
    print(doc)
```

## 20.4 Conversion of Corpora to/from NumPy/SciPy

Sometimes a corpus might be saved using a matrix from NumPy or SciPy (see also Section 6.9 starting on page 78). It is possible to convert back and forth to/from the gensim corpus format. Here we again assume each document is already represented by token ID and value pairs. The code below can be found in the file `code-gensim-020-numpy-scipy.py` on the course website.

```
# This script illustrates the conversion between numpy/scipy matrices
# and gensim corpora.

import gensim
import numpy as np

# Use a random matrix as an example. This is in fact a dense matrix
# from numpy, which is not as efficient as a sparse matrix (which we
# will discuss further below). The basic idea during the conversion is
# that the numpy matrix has stored the word counts (or whatever vector
# space we're talking about here) such that a text document
# corresponds to a given COLUMN. On the other hand, when you print the
# gensim corpus, each text document corresponds to a given ROW (at
# least when you're printing it in the IPython command shell).
m_np = np.random.randint(10, size=[5, 2])
corpus = gensim.matutils.Dense2Corpus(m_np)
list(corpus)
for doc in corpus:
    print(doc)

# Here we convert the the corpus back into a dense numpy 2D array.
length = max(len(doc) for doc in corpus) # Generator epression.
gensim.matutils.corpus2dense(corpus, num_terms=length)

# Here we use a sparse matrix from SciPy. This can be much more memory
# efficient than a dense matrix, especially when many matrix elements
# are zero (as is often the case in text mining).
import scipy.sparse
m_sp = scipy.sparse.random(5, 2, density=0.5)
m_sp.todense()  # Take a look.
m_sp.A          # Take a look (same as above).
```

```
corpus = gensim.matutils.Sparse2Corpus(m_sp)
list(corpus)
for doc in corpus:
    print(doc)

# Convert corpus back to ‘scipy.sparse.csc‘ matrix.
gensim.matutils.corpus2csc(corpus)
```

## 20.5 Streaming Corpus

To avoid the problem of holding the whole corpus in memory, you can stream the corpus. This means that instead of holding it in memory, you only hold one document at a time in memory and process the corpus step by step. This approach uses **iterators**, which are **discussed in Section 6.12** starting on page 83.

In the following example code we convert each document to lowercase and return a list of words in that document. Here we assume that `data-gensim-mycorpus.txt` is **a file holding one document per line** (see the course website for a copy of that file).

```
class MyCorpus(object):   # Class that instantiates an iterable.
    def __iter__(self):   # Define a generator function.
        for line in open('data-gensim-mycorpus.txt'):
            yield line.lower().split()
```

Now you can create a corpus object **without** loading the whole corpus into memory:

```
mycp = MyCorpus()   # Create iterable by instantiating the ‘MyCorpus‘ class.
```

If you want to take a look at the corpus, you can iterate over it and print each document, one at a time:

```
for doc in mycp:
    print(doc)
```

### 20.5.1 BoW Using Streaming Corpus

A more complete example can be found in the following script. The code assumes that **all documents are contained in a single text file, with each line containing a separate text document**. For simplicity, we only convert to lowercase and use a very simple tokenizer that splits on whitespace. (In production, you would use a for example NLTK to tokenize as discussed in Section 13.1 starting on page 161 or use gensim's tokenizer as illustrated in Section 20.5.2 starting on page 210.) Finally, we calculate BoW. All of this is done using iterators, so we can stream the whole corpus through memory without having to load the whole corpus all at once. The code below can be found in the file `code-gensim-030-streaming.py` on the course website.

```
# This script shows how to stream text documents using iterators. The
# key advantage is that we do not need to hold all the documents
# (i.e. the whole corpus) in memory. Instead, we process it chunk by
# chunk using iterators.
#
# In the first part, we show how to generate and modify a gensim
# `Dictionary` using iterators.
#
# In the second part, we show how to stream a corpus and calculate bag
# of words.
from gensim import corpora

# Construct dictionary WITHOUT loading all texts in memory. Instead,
# we chunk over the documents step by step using a generator
# expression. Once we're done with a document, the memory is freed.
dictionary = \
    corpora.Dictionary(
        line.lower().split() for line in open('data-gensim-mycorpus.txt'))

# Get the IDs in gensim that correspond to stopwords.
stoplist = set('for a of the and to in'.split())
stop_ids = \
    [dictionary.token2id[stopword] for stopword in stoplist
     if stopword in dictionary.token2id]

# Obtain the token IDs for words that appear only ONCE in the text
# corpus (these words will be removed later). `iteritems` from the six
# library is used here to iterate over the keys and values of the
# dictionary `dictionary.dfs` in a way that works both with Python 2
# and 3. (In Python 3 you would use `dictionary.dfs.items()`.)
# `dictionary.dfs` is a dict with the token ID and its document
# frequency (i.e. the frequency with which it occurs in the text
# corpus as a whole).
from six import iteritems
once_ids = \
    [tokenid for tokenid, docfreq in iteritems(dictionary.dfs)
     if docfreq == 1]

# Remove tokens that are stopwords or words that appear only once (or
# both).
dictionary.filter_tokens(stop_ids + once_ids)

# Remove gaps in ID sequence after words that were removed.
dictionary.compactify()

# Show how many words ("tokens") the dictionary contains, show their
# mappings to integer IDs, and show their document frequency (i.e. how
# often they appear in the text corpus).
print(dictionary)
print(dictionary.token2id)
print(dictionary.dfs)

# Using the 'dictionary' created above, we can also stream a corpus
# into gensim and calculate BoW. Here we define the '__iter__()'
# method as a generator function. As a consequence, an instance of
# 'MyCorpus' is an iterable. We iterate through a file under the
# assumption that each line holds one document. If this assumption is
# not satisfied (e.g. your file has a different format to hold the
# text documents), you can modify the definition of the '__iter__'
# method to fit your input format.
class MyCorpus(object):
    def __iter__(self):          # Define generator function.
        for line in open('data-gensim-mycorpus.txt'):
            yield dictionary.doc2bow(line.lower().split())

# Printing the corpus just shows the address of the object in
# memory. To see all the documents (represented as vectors using
# 'doc2bow' above), we iterate over the corpus. The important thing is
# that only a SINGLE vector resides in memory at a time!
mycp = MyCorpus()       # Create iterable (by instantiating the class).
print(mycp)
for vector in mycp:
    print(vector)

# Another way to print a corpus is to load it entirely into memory.
print(list(mycp))
```

### 20.5.2 BoW and SVD Using Streaming Corpus

Here is another example using iterators. In this example, the assumption on the data is that **each document is stored in a separate text file** ending with `.txt`. We also use gensim's tokenizer `gensim.utils.tokenize` to split up each document into separate words (tokens). We then calculate BoW and finally conclude with an example application of singular value decomposition (SVD). The point of writing this script is to show how all of these steps (including SVD) can be done by steaming data without overloading the memory. The code below can be found in the file `code-gensim-040-streaming-with-iterators.py` on the course website.

```
# This script shows how you can construct a streaming corpus and then
# pass it on to functionality in gensim. The basic idea is that the
# corpus is not held in memory all at once, but instead processed
# record-by-record. For the purpose of this script, we assume that one
# document corresponds to one file on disk. At the end of the script
# we show an illustration using singular value decomposition (SVD).
import gensim, os

def iter_documents(top_directory):
    """This is a generator function: Iterate over all documents, yielding
    one tokenized document (=list of utf8 tokens) at a time. Each
    document corresponds to one file. The generator function finds all
    `.txt` files, no matter how deep under `top_directory`.

    """
    for root, dirs, files in os.walk(top_directory):
        for fname in filter(lambda fname: fname.endswith('.txt'), files):
            # Read each `.txt` document as one big string.
            document = open(os.path.join(root, fname)).read()
            # Break document into utf8 tokens.
            yield gensim.utils.tokenize(document, lower = True, errors = 'ignore')

# Here we print all tokens for all documents. `iter_documents()`
# creates a generator iterator that yields one text document at a
# time. Keep in mind that `gensim.utils.tokenize` itself returns a
# generator iterator for each document, so we have to iterate again
# through `doc_tokens`.
for doc_tokens in iter_documents('.'): # '.' refers to the working directory.
    print('\nNext␣document:\n')
    for token in doc_tokens:
        print(token)

class TxtSubdirsCorpus(object):
    """This class instantiates an iterable: On each iteration, return
    bag-of-words vectors, one vector for each document. Process one
    document at a time using generators, never load the entire corpus
    into RAM.

    """
    def __init__(self, top_dir): # Constructor method.
        self.top_dir = top_dir   # Save the top-level directory name.
        # Create a dictionary, which is a mapping for documents to
        # sparse vectors.
        self.dictionary = gensim.corpora.Dictionary(iter_documents(top_dir))

    def __iter__(self):          # Define a generator function.
        for doc_tokens in iter_documents(self.top_dir):
            # Transform tokens (strings) into a sparse bag-of-words
            # vector, one document at a time.
            yield self.dictionary.doc2bow(doc_tokens)

# Create the streamed corpus of sparse document vectors. `corpus` is
# an iterable.
corpus = TxtSubdirsCorpus('.')

# Print the corpus vectors. Each vector is sparse and corresponds to
# the contents of one text document.
for vector in corpus:
    print(vector)

# Run truncated Singular Value Decomposition (SVD) on the streamed
# corpus. In this case we give a hint to the SVD algorithm to process
# the input stream in groups of 5000 vectors. This is called
# "chunking" or "mini batching." The return values are as follows: `U`
# contains the left-singular vectors (encoded as a matrix where the
# columns correspond to documents), while `Sigma` contains the
# singular values of the corpus (encoded as a vector). The V^t matrix
# is not stored explicitly because it may not fit into memory.
from gensim.models.lsimodel import stochastic_svd
U, Sigma = \
    stochastic_svd(
        corpus,
        rank=200,              # Truncate the SVD.
        num_terms=len(corpus.dictionary),
        chunksize=5000)
```

## 20.6 Latent Semantic Analysis (LSA)

We have discussed LSA (and how it works together with SVD) before in Section 19.1 starting on page 197. Here we go into more details and show how it works in Gensim. Gensim internally uses the BLAS libraries for calculating the singular value decomposition (SVD), so if you run this code on a multi-core machine, BLAS takes care of parallelizing the computations to speed up the code.

A big advantage is the "online" feature (vs. "batch" feature) of LSA. Suppose the nature of the input stream of documents changes and there is **topic drift**. For example, you've been following the financial press and people were originally talking about financial companies (e.g. banks) but now prefer to talk about tech companies. In that case, LSA re-orients itself to reflect these changes in a relatively small amount of updates. This makes LSA more suitable for text analytics where topics can change dynamically. In contrast, LDA (see Section 20.7) can be much slower to react to these changes and therefore it might make sense to run LDA in batch mode if there is topic drift (i.e. run it on the whole corpus and there are not changes to the corpus afterwards).

The data is based on a random download of parts of the English Wikipedia. Please see the instructions in Chapter 3 in the "Wikipedia" bullet point on how you can download the data yourself from Wikipedia. You still need to run the following command (from the Linux command line) on it to convert the articles to plain text and store the results as sparse BoW or tf-idf vectors:

```
$ python3 -m gensim.scripts.make_wiki
```

As always, when working with data, you need to monitor your data quality and be careful not to fall into the garbage in, garbage out trap. Apparently there are lots of templates listed in the Wikipedia download. Furthermore, some websites seem to be influenced by bots that import databases of cities, countries, etc. So if you would get some real work done with this Wikipedia data, **further preprocessing and cleaning would be required**.

In the code below, `lsi[mm]` (after converting it to a dense matrix with `corpus2dense`) corresponds to $U^{-1}M$ from the SVD given by $M = U\Sigma V^t$. And `mm` in the code below corresponds to the matrix $M$. (For a primer on SVD, see Section 19.1 starting on page 197.) This implies that $V^t = \Sigma^{-1}U^{-1}M$, or $V = (U^{-1}M)^t\Sigma^{-1} = \text{lsi[mm]}^t/\Sigma$, where we have used the fact that $\Sigma$ is a diagonal matrix. This is the formula we use in the code below to compute $V$.

The code listed below for computing LSA with Gensim can be found in the file `code-gensim-050-LSA.py` on the course website.

```python
# This script illustrates how to use latent semantic analysis (LSA)
# using the Gensim package. The data is a random extract from the
# English Wikipedia.

import logging, gensim
logging.basicConfig(
    format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)

# Load the ID to word mapping, i.e. the dictionary. This has been done
# in a previous step, when preparing the corpus with
# `gensim.scripts.make_wiki`.
id2word = \
    gensim.corpora.Dictionary.load_from_text(
        '../data-wiki-en/wiki_en_wordids.txt.bz2')
# Load the corpus iterable.
mm = gensim.corpora.MmCorpus('../data-wiki-en/wiki_en_tfidf.mm')
# Take a look at basic information about the corpus.
print(mm)
# Compute LSA (or LSI as it is sometimes called) of the English
# Wikipedia.
lsi = \
    gensim.models.lsimodel.LsiModel(
        corpus=mm,
        id2word=id2word,
        num_topics=400)
# Print the words that contribute most (both positively and
# negatively) for each of the first ten topics.
lsi.print_topics(10)
# Recover the matrices of the SVD. See
# https://github.com/RaRe-Technologies/gensim/wiki/recipes-&-faq for
# details. Keep in mind that the matrix V is not stored explicitly by
# gensim because it may not fit into memory, as its shape is 'num_docs
# * num_topics'. However, you can manually compute it as shown below
# to get it as a 2-dimensional numpy array (i.e. a matrix).
U = lsi.projection.u
Sigma = lsi.projection.s
V = \
    gensim.matutils.corpus2dense(
        corpus=lsi[mm],    # Use gensim's streaming 'lsi[corpus]' API.
        num_terms=len(lsi.projection.s)).T / \
    lsi.projection.s
```

## 20.7 Latent Dirichlet Allocation (LDA)

We have discussed LDA in Section 19.2, starting on page 202. The idea is that you can think of words in a given document coming from a set of topics. For example, if you see the word "meow," it is likely to come from a topic related to cats, while "bark" comes from a dogs-topic. The word "milk" might be from a cow-related topic (with, say 80% probability) or from a cat-related topic (with, say, 20% probability).

In this chapter we focus on gensim, but keep in mind that you can do LDA with other packages as well, such as lda or sklearn, specifically `sklearn.decomposition.LatentDirichletAllocation`.

You can run LDA in online or in batch mode. **Online mode** means that you have an input stream of training documents and update LDA sequentially based on the new documents observed. **Batch mode** on the other hand means that you go over the entire corpus when creading the LDA model and there are no additional documents encountered afterwards. In contrast to LSA (see Section 20.6), where online mode usually works fine, you need to be **careful when using LDA in online mode**. The problem is that if the properties of the input stream change (e.g. the documents were about financial companies, now they are about tech companies), the impact of later updates on the LDA model gradually diminishes. This means that the quality of the LDA's model fit to the data decreases and LDA effectively gets confused and increasingly unsuitable to work with the updated data. In that case, you might want to **run LDA in batch mode** where the entire training corpus is known beforehand and thus does not exhibit topic drift by definition. However, running LDA in batch mode might take a longer time than running it in online mode.

In the following example, the data is the same as in Section 20.6, please take a look there how to obtain the data and the caveats mentioned there about the data quality. The code below can be found in the file `code-gensim-060-LDA.py` on the course website.

```
# This script illustrates how to use latent Dirichlet allocation (LDA)
# using the Gensim package. The data is a random extract from the
# English Wikipedia.
```

```python
import logging, gensim
logging.basicConfig(
    format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)

# Load the ID to word mapping, i.e. the dictionary. This has been done
# in a previous step, when preparing the corpus with
# `gensim.scripts.make_wiki`.
id2word = \
    gensim.corpora.Dictionary.load_from_text(
        '../data-wiki-en/wiki_en_wordids.txt.bz2')
# Load the corpus iterable.
mm = gensim.corpora.MmCorpus('../data-wiki-en/wiki_en_tfidf.mm')
# Take a look basic information about the corpus.
print(mm)

# Extract 100 LDA topics, using 1 pass and updating once every 1 chunk
# (10,000 documents).
lda = \
    gensim.models.ldamodel.LdaModel(
        corpus=mm,
        id2word=id2word,
        num_topics=100,
        update_every=1,
        chunksize=10000,
        passes=1)
# Print the most contributing words for 10 randomly selected topics.
lda.print_topics(10)

# Extract 100 LDA topics, using 20 full passes, no online updates.
lda = \
    gensim.models.ldamodel.LdaModel(
        corpus=mm,
        id2word=id2word,
        num_topics=100,
        update_every=0,
        passes=20)

# A trained model can used be to transform new, unseen documents
# (plain bag-of-words or tf-idf count vectors) into LDA topic
# distributions. Here `doc_bow` would be a set of new documents.
doc_lda = lda[doc_bow]
```

## 20.8 Word2vec

Word2vec is a very popular way to **convert words into vectors** in a meaningful way. We have already discussed it briefly in Section 14.5 starting on page 177. The analysis is based on the **co-occurrence of words** in each text document. The algorithm learns automatically some inherent relations between the words. Each word is transferred into a vector in a larger vector space using unsupervised learning (so no human intervention is necessary, besides maybe tuning some parameters). The basic idea is that ideally you can do calculations in this vector space as follows, where "queen," "king," "man," and "woman" are vectors representing each of these words:

$$\text{queen} \approx \text{king} - \text{man} + \text{woman}.$$

However, one has to keep in mind that doc2vec really works on the word level, not the document level. So it is not directly comparable to BoW or tf-idf because those methods yield one vector per document, while word2vec yields one vector per word. There are also alternatives to word2vec, such GloVe (for a comparison see here), but they don't seem to perform that much better.

So how does word2vec actually work? The basic idea is that it chains two or three layers in a neural network together. There is an input layer, a hidden layer, and an output layer. (Word2vec is therefore not deep learning, see Section 11.5 starting on page 122.) There are two alternative ways this works, the first one being called **continuous bag of words (CBOW)**, which tries to use the word context to predict a so-called target word. For example, if you give it "I love" then it would (hopefully) predict that the next word is "NLP." The second way is called **skip-gram** and tries to predict a target context. For example, if you give it the word "love," it will (hopefully) predict that the word before is "I" and the following word is "NLP."

So you see that word2vec really looks at the co-occurrence of words. Specifically, it **takes word order into account**. It is similar to performing BoW or tf-idf on n-grams, but it may perform better (depending on the application) because it does not suffer from the high degree of sparsity and the high dimensionality that BoW on n-grams is plagued with.

The code below can be found in the file `code-gensim-070-word2vec.py` on the course website.

```python
# This script contains a basic example on how to use word2vec in
# gensim.

# Import modules and set up logging.
import gensim, os, logging
logging.basicConfig(
    format='%(asctime)s␣:␣%(levelname)s␣:␣%(message)s',
    level=logging.INFO)

# word2vec expects a sequence of sentences as its input. In this
# example, each sentence is a list of words. This is fine, but
# everything is in RAM so if you have lots of sentences this could be
# a problem. Iterables to the rescue! In fact, all gensim requires is
# to get the sentences sequentially, e.g. using an iterable (see
# further below).
sentences = [['1st', 'sentence'], ['2nd', 'sentence']]
# Train word2vec on the two sentences.
model = gensim.models.Word2Vec(sentences, min_count=1)

# Here we assume the input is on several files on disk, one sentence
# per line. Using an iterable, gensim can process each input file
# line-by-line. It yields one sentence after another.
class MySentences(object):      # This class instantiates an iterable.
    def __init__(self, dirname): # Constructor method.
        self.dirname = dirname

    def __iter__(self):          # Here we define a generator function.
        for fname in filter(lambda fname: fname.endswith('.txt'),
                            os.listdir(self.dirname)):
            for line in open(os.path.join(self.dirname, fname)):
                yield line.split() # Yield one sentence, split into words.

# Here we get a memory-friendly iterable and hand it over to gensim's
# word2vec. '.' refers to the current directory. Gensim will run
# several passes over the iterable. The first one is to collect words
# and frequencies to build an internal dictionary structure. The later
# ones are to train the neural model.
sentences = MySentences('.')    # Instantiate the class. Create iterable.
model = gensim.models.Word2Vec(sentences)

# If for some reason your input stream is non-repeatable, you can
# perform the steps mentioned above manually. This code is just for
# illustration, it will NOT work in this script as we have not defined
# the 1-pass generators here.
model = gensim.models.Word2Vec(iter=1) # Empty model, no training yet.
model.build_vocab(some_sentences) # Can be a non-repeatable, 1-pass generator.
model.train(other_sentences) # Can be a non-repeatable, 1-pass generator.

# If you want to fine-tune the training, you can change some
# parameters.
model = \
    gensim.models.Word2Vec(
        sentences,
        min_count=10,      # Prune infrequent words. Default is 10.
        size=200,          # Size of neural net layers. Default is 100.
        workers=4)         # Workers for parallelization. Default is 1.

# You can save and load the model to/from file.
model.save('mymodel')
```

```
new_model = gensim.models.Word2Vec.load('mymodel')
new_model.train(more_sentences) # You can continue to train it with more sentences.

# Originally, word2vec was released by Google and was written in
# C. You can read from the format used by that implementation as well.
model = Word2Vec.load_word2vec_format('/tmp/vectors.txt', binary=False)
# Using gzipped/bz2 input works too, no need to unzip.
model = Word2Vec.load_word2vec_format('/tmp/vectors.bin.gz', binary=True)

# You can find the most similar words. Here the output might be
# something like "[('queen', 0.50882536)]".
model.most_similar(
    positive=['woman', 'king'],
    negative=['man'],
    topn=1)
# You can check which word does not match. Here the output might be
# 'cereal'.
model.doesnt_match("breakfast␣cereal␣dinner␣lunch";.split())
# You can also check the similarity between different words. Here the
# output might be a number such as 0.74.
model.similarity('woman', 'man')

# If you need the raw output vectors.
model['computer']                # Raw NumPy vector of a word.
# If you want them en-masse as a two-dimensional NumPy matrix.
model.syn()
```

## 20.9 Doc2vec

Doc2vec is an extension of word2vec. We have already discussed it briefly in Section 14.5 staring on page 177. The basic idea is to extend word2vec (which works on the word level) to work on larger blocks of texts, such as sentences, paragraphs, or entire documents. The goal is that similarly to BoW and tf-idf, you want to **generate one vector per document**. Similar to word2vec, doc2vec takes word order into account.

As the name suggests, doc2vec is related to word2vec. One could extend word2vec in various ways. One simple way would be to simply take the average over all word vectors that occur in a document. However, this might be a bit crude, so what doc2vec does is to **add additional features (variables) to the word2vec neural network representing document IDs**.

The script below shows basic usage. The code below can be found in the file `code-gensim-080-doc2vec.py` on the course website.

```
from gensim.test.utils import common_texts
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
common_texts  # Take a look at the common texts. Output is as follows:
# [['human', 'interface', 'computer'],
#  ['survey', 'user', 'computer', 'system', 'response', 'time'],
#  ['eps', 'user', 'interface', 'system'],
#  ['system', 'human', 'system', 'eps'],
#  ['user', 'response', 'time'],
#  ['trees'],
#  ['graph', 'trees'],
#  ['graph', 'minors', 'trees'],
#  ['graph', 'minors', 'survey']]
model = \
    Doc2Vec(
        [TaggedDocument(doc, [i]) \
            for i, doc in enumerate(common_texts)],
        vector_size=5,          # Dimensionality of feature vectors.
        window=2,      # Max distance between current & predicted word.
        min_count=1,  # Ignore words with lower frequency.
        workers=3)    # Number of worker threads to train the model.
# Show the vector for a new document.
model.infer_vector(['system', 'response'])
# Output is as follows. Note that the length of the output vector is
# as specified above in `vector_size`.
# array([-0.04900227, 0.00258817, 0.06024337, 0.0207104 , -0.01241806],
#        dtype=float32)
```

The script below shows more advanced usage. The code below can be found in the file `code-gensim-090-doc2vec.py` on the course website.

```
# This script illustrates doc2vec in gensim. Actually this is more of
# a collection of code snippets, not really a runnable script. In this
# example, we assume that we're interested in sentence-level data (not
# document-level data; in any case, the code would stay very similar
# if you update it to work on whole documents.)

# The input to Doc2Vec is an iterator of `LabeledSentence`
# objects. Each such object represents a single sentence, and consists
# of two simple lists -- list of words and a list of labels. WHY DO WE
# NEED LABELS? Because labels in doc2vec act the same way as words in
# word2vec.
sentence = \
    LabeledSentence(            # Or use `TaggedDocument`.
        words=['some', 'words', 'here'],
        labels=['SENT_1'])

# Here is an example to read text from a file with one sentence per
# line, using the following class as training data. In principle, one
# could have SEVERAL labels per sentence, but the most common
# application is probably to have ONE label per sentence, as shown
# here.
class LabeledLineSentence(object):
    def __init__(self, filename): # Constructor method.
        self.filename = filename

    def __iter__(self):          # __iter__() method implemented as generator functic
        for uid, line in enumerate(open(filename)):
            yield LabeledSentence(words=line.split(), labels=['SENT_%s' % uid])

# You can manually control the learning rate as follows, which might
# in some cases yield better results. Or you could randomize the order
# of the input sentences.
model = Doc2Vec(alpha=0.025, min_alpha=0.025) # Use fixed learning rate.
model.build_vocab(sentences)
for epoch in range(10):
    model.train(sentences)
    model.alpha -= 0.002            # Decrease the learning rate.
    model.min_alpha = model.alpha # Fix the learning rate, no decay.

# You can save and load Doc2Vec instances the usual way.
model = Doc2Vec(sentences)
# Store the model to mmap-able files (can map files into memory).
model.save('my_model.doc2vec')
# Load the model back.
model_loaded = Doc2Vec.load('my_model.doc2vec')

# Since labels in doc2vec act in the same way as words in word2vec. So
# to get the most similar words/sentences to the first sentence
# (i.e. label 'SENT_0'), you could type:
print(model.most_similar('SENT_0'))
print(model['SENT_0'])
```

# Appendix A

# Operating System

Python is available on all of the major operating systems such as Windows, macOS, and Linux. Therefore **it does not really matter which operating system you use to run Python**.

## A.1 CLI

When talking about computer programming in the context of an operating system, sometimes the terms "**command line interface (CLI)**," "**command prompt**," "**shell**," or "**terminal**" are mentioned. These terms are largely synonymous with each other, and refer to a method of interacting with the operating system through text commands. They date back to the computer terminals of the 1960s, which typically consisted of a keyboard for input and a screen or printer for output, allowing users to communicate with the computer system. The basic idea is that instead of using a mouse (not available at that time) and clicking your way through the operating system, you type commands on a keyboard and look at the text output generated by your computer. Of course, CLIs have evolved and improved considerably since the 1960s.

The command line can be a very effective tool because you can combine various commands in ways that are difficult to replicate using a mouse and a point-and-click interface. For example, using the command line you can rename files in batches, use version control, search for files, etc. You can also program most shells by writing scripts (i.e. small computer programs) that automate tasks. Various command line interfaces are available for Windows (e.g. Windows Command Prompt ("CMD," "cmd.exe"), PowerShell, or Windows Terminal), macOS (e.g. Zsh), and Linux (e.g. Bash).

## A.2 Unix Terminology

In the context of installing software or doing system administration on macOS (a Unix operating system) or Linux (a Unix-like operating system), you will sooner or later come across the following terms:

- **"root"** is the name of the **superuser**. This is a special user account used for system administration. It is very powerful as there are almost no restrictions to what root can do on a computer. Use it with care.

- **“sudo”** is a program that allows you to run another program with the security privileges of another user, typically the superuser.

## A.3 Linux

Linux comes in hundreds of different “flavors,” so-called Linux distributions. The popularity of various Linux distributions can change quite a bit as time goes by. Distrowatch has a popular ranking of Linux distributions based on page hits. Some of the most popular Linux distributions these days are MX Linux, EndeavourOS, and Mint. If you would like to use Linux and are not sure which Linux distribution to use, you can give one of these a try.

- Before installing Linux, **make a backup** of all your data.
- You can install Linux inside a **virtual machine** or as **dual boot**. The advantage of a virtual machine over dual boot is that it might be easier to set up and that you don’t mess up your existing system if something fails in the Linux installation. But the disadvantage is that you’ll have less memory available to run your code (this might or might not be an issue depending on the memory requirements of your code and the amount of memory on your computer).
- Use a 64-bit Linux. Assuming that your computer hardware is from around 2004 or later, you do not need to use 32-bit Linux.

# Appendix B

# Python Installation

## B.1 Quick Summary

- For data science and machine learning, Anaconda is the easiest way to get started with Python on all major operating systems. **Anaconda** is a popular open-source distribution of Python that includes the language itself and some of the most popular add-on packages. You can find detailed installation instructions for Windows, macOS, and Linux here:
  docs.anaconda.com/anaconda/install

- Once you have installed Anaconda, you can type commands into the Anaconda prompt (if you're on Windows) or in the command line (if you're on Linux or macOS). The following bullet points explain how you can type commands to install an IDE and various Python packages. You do not need to be the root user and instead you can type the following commands as normal user.

- On Windows, you can skip this step, but if you're on Linux or macOS, you may need to activate Anaconda before you start using it:

  ```
  source ~/anaconda3/bin/activate
  ```

  Replace `~/anaconda3` with the actual installation path of Anaconda on your system in case it is different.

- To write Python code, use the Spyder integrated development environment (IDE). You can install Spyder with

  ```
  conda install anaconda::spyder
  ```

- You can install Python packages using `conda`, e.g. typing the following commands will install some of the fundamental Python packages required for this book:

```
conda install anaconda::numpy
conda install anaconda::scipy
conda install anaconda::pandas
conda install anaconda::pandasql
conda install anaconda::scikit-learn
conda install anaconda::tensorflow
conda install anaconda::keras
conda install anaconda::nltk
conda install anaconda::spacy
conda install anaconda::gensim
conda install anaconda::requests
conda install anaconda::bs4
conda install anaconda::selenium
conda install anaconda::seaborn
conda install anaconda::bokeh
conda install anaconda::dash
conda install anaconda::markdown
conda install conda-forge::pelican
conda install conda-forge::wordcloud
conda install conda-forge::rpy2
```

- On Linux and macOS, after you're done using Python, you can deactivate it again. This step is optional for Linux and macOS. (And on Windows you can skip this step.)

  ```
  conda deactivate
  ```

## B.2 Python Package Installation

As mentioned, I recommend using the **Anaconda** Python distribution. It was created specifically with the needs of a data scientist in mind. It includes Python itself and hundreds of packages. It is easy to set up and uses conda to manage the packages.

There are two main ways to installa Python packages: conda and pip. **If you use Anaconda, you should use conda to install Python packages, and you should not use pip**. The only reason you should use pip to install packages is if the package is not available through conda. The usual way to install a package is by typing the following command in the Anaconda prompt (if you're on Windows) or in the command line (if you're on Linux or macOS):

```
conda install package-name
```

You should replace `package-name` with the name of the package you would like to install, e.g. `scipy`.

You might already have a Python installation that was set up independently of Anaconda. For instance, many Linux distributions come with a system-wide Python installation by default. In such cases, you can install packages directly using tools like pip or conda, as both can operate without requiring Anaconda.

- **Pip** is the historically older installer. Often people use pip together with venv (or virtualenv which is very similar). Unless you need to manage non-Python dependencies, it is usually **best to go with pip+venv**. See Section B.3 starting on page 225 for details.

- **Conda** is the younger relative of pip. It is also open-source. In contrast to pip, it can also handle non-Python library dependencies. It is the package manager for the Anaconda Python distribution, but can also be used in isolation, independently of Anaconda. It is possible to install (some) conda packages within a venv, but it is better to use conda's own environment manager. Conda's environment manager works seamlessly with both conda and pip.

## B.3 Installing Packages With pip And venv

In this section, we explore the use of pip and venv on Linux and macOS. This method is only recommended for installing packages if you are **not** using Anaconda. Pip and venv are essential tools in the Python ecosystem that streamline package management and environment isolation.

- **pip**: Python's package installer, `pip` allows developers to easily install and manage libraries from the Python Package Index (PyPI).

- **venv**: This tool enables the creation of virtual environments, ensuring that each project has its own dependencies without conflicts.

Together, pip and venv help maintain organized and reproducible development setups, enhancing both productivity and project stability. Before diving in, it's important to understand why venv is beneficial. There are two primary reasons:

1. **Managing Python Packages for Different Projects:** Suppose you have two projects.

    - Project A requires an older version of the gensim package for backward compatibility.
    - Project B uses the latest version of gensim.

    With venv, you can create separate virtual environments for each project, keeping the different versions of gensim isolated. This ensures that each project uses the appropriate version without any conflicts.

2. **Installing Packages as a Normal User:** Using venv allows you to install packages without needing superuser privileges. This approach prevents the installation of Python packages globally on your system, which could potentially interfere with system tools or other projects. By keeping installations within a virtual environment, you maintain system integrity and reduce the risk of breaking other applications.

By leveraging pip and venv, you can effectively manage your Python projects, ensuring that each has the necessary dependencies while keeping your system clean and organized.

When you create a new project, you usually put it into a new directory. For example, let's assume that you would like to put your project into the `myproject` directory. The following code lets you create that directory, change into that directory, and then create a virtual environment there.

```
mkdir myproject            # Create new project directory.
cd myproject               # Change into directory.
python -m venv myenv       # Create virtual environment.
```

This code will set up a directory called `myenv` inside your project directory that contains the virtual environment. (If you're using Git, you should exclude `myenv` from your version control system using `.gitignore`.)

In any case, whenever you want to work on your project, you follow these steps:

1. Activate the venv. You change into your project directory (e.g. `cd myproject`) and then activate the venv.

   ```
   source myenv/bin/activate
   ```

2. You write code for your project or install packages you need for your project. For example, if you want to install or upgrade the requests package, you can now type at the command line:

   ```
   pip install --upgrade requests
   ```

3. When you're done (e.g. want to switch to another project or leave your venv for some reason), you can type

   ```
   deactivate
   ```

# Appendix C

# IDEs and Text Editors

We can create, edit, and save our Python programming code using an integrated development environment (IDE) or a text editor. An IDE allows you to edit your source code, run your programs, an debug your programs. It is a complete set of tools to develop your software and bring the different aspects of computer programming together in a single application, the IDE. In contrast, a text editor has a more narrow focus, which is to edit your source code. Nonetheless, there can be considerable overlap between IDEs and text editors, because many text editors used for programming have added functionality that make them similar to a full-blown IDE.

## C.1 Python IDEs

There are many IDEs for Python. **If you use Anaconda, I recommend using Spyder, as Spyder is already available in Anaconda.**

- Spyder is also built with a focus on data science and has an interface similar to RStudio, just like Rodeo. If you are using Anaconda, I recommend using Spyder as it is available in Anaconda.
- Rodeo was built specifically for data analysis. It is also relatively similar to RStudio, so switching between both is easy.
- PyCharm is made by JetBrains, which is well-known for a famous Java IDE called IntelliJ IDEA. If you have used other JetBrains products, PyCharm might be more familiar to you. However, PyCharm is not all open-source.
- AWS Cloud9 is an alternative from Amazon. It is cheap, but not free. It lets you code directly in your browser and you don't need to configure your local machine. It also enables collaboration between different people in real-time.

Honorable mention: Atom was a very popular code editor developed by GitHub. It was retired in December 2022. There are efforts by the community to continue Atom under the name "Pulsar" and the founder of Atom is writing a successor called "Zed" written in Rust.

## C.2 Text Editors

- For beginners I generally recommend using one of the Python IDEs mentioned in Section C.1 starting on page 227.
- Vim and Emacs are two alternative code editors for the brave.
- Instead of going with a full-fledged IDE, you can alternatively use Vim or Emacs, which are relatively ancient: vi and Emacs were both initially released in 1976. Nonetheless, they are **very powerful text editors** that still very widely used today.
- In fact, Vim started as a clone of the original vi editor and has added more features. Nowadays Vim is more popular than the original vi. On most Linux distributions, if you type `vi` on the command line, you will be redirected to Vim.
- There is a long history of mostly friendly rivalry between Vim and Emacs, just google "Editor Wars." In terms of popularity, Vim is by now the clear winner over Emacs, although that doesn't necessarily make Vim better than Emacs. I personally tend to favor Emacs slightly due to its internal use of a Lisp dialect to provide a deep extension capability. (Lisp is a very powerful family of programming languages with a long and distinctive history originating from MIT.) On the other hand, Vim has its own extension language as well, Vimscript.
- Whether you decide on using Vim or Emacs is really up to you and based on your personal preferences. You will make a good choice in any case.
- The basic idea about Vim and Emacs is that you can be more productive with them because you avoid taking your hands off the keyboard all the time to do something with your mouse. It is possible to control Vim and Emacs from your keyboard alone, without having to click around with your mouse. Ideally you get into a **flow-like state of mind**, where your ideas flow naturally from your thoughts through the keyboard onto the screen.
- An advantage of Vim and Emacs is that you can use both with all major programming and markup languages. This means the if you work e.g. with both Python and R, you don't have to switch between different IDEs all the time. For example, this book is written in Emacs (using the document preparation system LaTeX) and the Python programming code for this course is written in Emacs as well.
- Both Vim and Emacs are extremely extensible and you can customize them ad infinitum. **You can even play Tetris on Emacs** if you wish! There is the joke by Vim advocates that Emacs is a great operating system, lacking only a decent editor.
- There are several so-called "**Emacs starter kits**" out there, the most popular one being Spacemacs and Doom Emacs. You can combine it nicely with Magit, which is an interface to the Git version control system, see Appendix D starting on page 229.

# Appendix D

# Version Control

A version control system lets you track changes in computer files, i.e. mostly source code files, but also other files such as documentation. Moreover, it allows you to coordinate the work performed on those files among several people. And even if you just code by yourself, it can be immensely helpful to put your code under version control. For example, it allows you to track past changes you made in your code. And of course if you are working in teams on a software project, then version control is a must. It allows you to keep your codebase in a consistent state, even if several people work at it at the same time.

Please keep in mind that version control is not related to the version number of your software, e.g. Version 1.0 or 2.0. A version number is simply a number commonly assigned by the developers to a specific software release. However, the internal development of the software proceeds in many small steps (where version control is used to keep track of the changes made in these small steps), and these smaller steps usually do not have a version number.

## D.1 Git

There are many version control systems out there, the most popular one for open source being Git. You can put your code in a so-called **repository**, which contains your code as well as the metadata that keeps track of the changes you have made. Different **revisions** act like snapshots of your codebase. Git is often operated from the command line, although there are also graphical clients available. There are a number of useful online resources available:

- Setting up Git
- Using Git
- Git cheat sheets
- Git and GitHub learning resources

## D.2 GitHub

- GitHub is the leading hosting service for repositories. As the name indicates, it is mainly targeted at Git, but it also supports other version control systems. In any case, **Git plus GitHub is the gold standard nowadays** and you are encouraged to use this combination.

- **Why use GitHub** to host your repositories? For three reasons:

  1. It increases your visibility and helps you with your career and job market. You can reference your work displayed on GitHub directly on your CV and in your application package to impress potential employers.
  2. It is best industry practice to use version control.
  3. GitHub fosters collaboration and an open culture.

- For instructions see the official documentation, or google for something like “GitHub tutorial” or “GitHub command line tutorial” (without the quotes). I **recommend using GitHub from the command line and setting up SSH keys**. It just makes your life much simpler compared to HTTPS, where you always have to re-enter your password.

- Your **data** can go on GitHub as well. If it is not too big you can just include it in your normal code repo. However, a better practice might be to use Git Large File Storage or Dropbox.

- If you need to share some **code snippets** (i.e. small regions of re-usable source code) or notes, put them on GitHub Gist. This can come in handy for blog posts, see also Section 2.2 starting on page 13.

- Keep in mind that GitHub can render GitHub Flavored Markdown (GFM), and Jupyter Notebooks, so you can make a lot of documentation etc. available online directly in your GitHub repo (or reference it in your blog, see Section 2.2).

- Keep in mind that **everything you put on GitHub is available for the whole world to see** (unless you use a private repo, which you should not do for this course). This is part of the reason why open-source software often is of high quality. People cannot hide crappy code in their drawer any more, so they have incentives to polish it. Another reason is that coding is much more collaborative, so your learning curve increases because you can learn from other coders much more easily; furthermore, it is easier to identify star-coders whom you can follow and you can read and learn from their code.

# Appendix E

# Python Package Guide

Python is a language that really shines when it comes to its ecosystem, i.e. the packages available. In fact, one reason why Python is so popular is that for many tasks you do not have to write a program from scratch, but instead can use an existing package written by someone else that provides the required functionality.

This chapter therefore provides a list of selected Python packages that are useful in the context of text analytics and NLP. The list is not set in stone, but it will give you a good overview of the most important Python packages for data science and NLP. I have ordered them roughly according to their popularity, in particular by the amount of stars on GitHub.

## E.1 NLP

The following list includes some of the most popular Python packages for text analytics and NLP:

- Hugging Face has several well-known libraries, the most popular being the Transformers library which makes use of deep learning
- SpaCy
- Rasa allows you to automate text- and voice-based conversations, it can be used to create chatbots and voice assistants
- Gensim
- NLTK
- TextBlob is built on top of NLTK and Pattern, it can perform sentiment analysis, part-of-speech tagging, noun phrase extraction, and translation.
- Stanza is Stanford NLP Group's official Python NLP library, which provides access to the CoreNLP library (which is written in Java)
- Spark NLP is useful if you like to make use of deep learning models such as transformers; it is built on top of Apache Spark

## E.2 Web Scraping

Many of the most interesting textual datasets are available from the internet. To obtain this data, one has to use a technique called "web scraping," which means to "scrape" the data from the web. This is a huge topic and the list below is an opinionated summary of the most important Python packages. See also **Appendix F starting on page 239** for more details on how to use these packages.

The following lists provide an overview of various web scraping packages:

- Obtaining data:
  - Requests
  - urllib2 used to be popular, but nowadays most people use Requests.
  - Selenium can be used if there is a dynamic website that uses JavaScript/AJAX, or if you need to click through some forms to get to the data you need.
  - RoboBrowser is similar to Selenium but is built on Requests and Beautiful Soup. For some of the older folks out there, RoboBrowser is similar to Mechanize (which has been around since 2004). It has browser history and cookies, is able to fill in forms and click links. However, if you need to deal with JavaScript/Ajax, Selenium is more suitable.
  - requests_html is similar to the requests+BeautifulSoup combination, but it can deal with JavaScript. Effectively it is an alternative to Selenium.
- Extracting/parseing data from previously downloaded HTML or XML documents:
  - Beautiful Soup (bs4)
  - lxml: Some people are of the opinion that lxml is better than Beautiful Soup. For those people using lxml instead of Beautiful Soup, most use lxml.html, with lxml.html.html5parser if needed. Some people also use lxml.etree with `HTMLParser` class if needed. For CSS selectors, lxml.cssselect is popular.
  - requests_html, see above.
  - python-readability can extract the main body text from a HTML document and clean it up.
- Scrapy is a complete web framework, but for simple web scraping tasks it is overkill. It can be useful if you want to have a spider that crawls through entire websites in a systematic way. You can swap individual modules with other Python web scraping libraries, e.g. if you need to scrape dynamic websites (i.e. websites making heavy use of JavaScript/Ajax) you can use Selenium as a backend for Scrapy.
- PySpider is another popular web crawler similar to Scrapy (but less popular).
- Specific web services:

- Get **newspaper** data: Newspaper3k, Goose3 (a re-rwite of the original python-goose library), or news-please. If you look at the source code of news-corpus-builder you can see some example code using Goose (and Feedparser).
- Newspaper3k and Goose can also be used for extracting the text of an article as well as meta information such as the title or author.
- See also Pattern above for **Google, Twitter, and Wikipedia**.
- For accessing **Google News** you can use GoogleNews.
- There are many Python packages for accessing **Twitter**, see also Section F.4 starting on page 248 for code examples. Broadly speaking, packages fall into two categories depending on whether they use the official Twitter API or reverse-engineer Twitter's JavaScript front-end (which is an undocumented, unofficial API). The official Twitter API is convenient to access, but it has several limitations, e.g. the amount of data you can download. In contrast, the unofficial JavaScript API often imposes fewer restrictions and typically does not require authentication (but one would need to check the legal terms for downloading). The following list is a partial record of Python packages for scraping Twitter, sorted according to popularity as measured by GitHub stars:
  1. Twint does not require authentication, does not use Twitter's official API, and does not have rate limitations.
  2. Tweepy, see also Section F.4 starting on page 248. Tweepy conceptually is the "opposite" to Twint in the sense that Tweepy accesses the official Twitter API.
  3. Pattern, as mentioned above.
  4. twitter-scraper, installed and imported as `twitter_scraper`, for accessing the unofficial API.
  5. python-twitter for accessing the official Twitter API.
  6. snscrape for accessing the unofficial API. Besides **Twitter**, snscrape can scrape from various social networks such as **Facebook, Instagram, Reddit, Telegram, VKontakte, and Weibo**.
  7. twitter for accessing the official Twitter API.
  8. twitterscraper for accessing the unofficial API.
  9. twython for accessing the official Twitter API.
  10. Scweet if everything else fails. It only uses Selenium, so it "pretends" to be a normal user, and it doesn't use the official or unofficial API.
  11. GetOldTweets3 if you need to retrieve older tweets. For accessing the unofficial API.
- PRAW and PSAW let you access **Reddit**. PRAW is more suitable for real-time access, while PSAW (which is a connection method to the pushshift.io API) allows you to get historical data. You can also use PRAW and PSAW in combination.
- facebook-page-post-scraper for scraping **Facebook** pages.
- Feedparser for parsing **Atom and RSS feeds**.

## E.3 Extracting Text from PDF Files

- Textract is a very popular package to extract text from any document. It is a wrapper for Poppler/Xpdf's `pdftotext`.

- Tika is an alternative that might be easier to install if you're running Windows.

```
from tika import parser
raw = parser.from_file('sample.pdf')
print(raw['content'])
```

- PyPDF2 is sometimes mentioned, but it seems to have some problems as it does not always recognize the text correctly.

- If nothing works, you can try to call the `pdftotext` binary (from Xpdf) from Python directly. The code could look similar to the following one, where you might have to adapt the path to `pdftotext` depending on where it is installed in your system:

```
import os, subprocess
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
args = ['/usr/local/bin/pdftotext',
        '-enc',
        'UTF-8',
        '{}/my-pdf.pdf'.format(SCRIPT_DIR),
        '-']
res = subprocess.run(args, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
output = res.stdout.decode('utf-8')
```

## E.4 Core Data Libraries

These are the packages most essential for data wrangling in Python. If your data has different data types (e.g. one column contains dates while another contains numbers), you should use Pandas. Pandas is explained in detail in Appendix G starting on page 255. On the other hand, for vectors, matrices, or arrays (i.e. data containing only one data type) you can use NumPy. SciPy provides sparse matrices and it has basic numerical tools such as integration, differentiation, and optimization.

- Pandas

- NumPy

- SciPy

## E.5 Databases

As your data become larger, it often makes sense to store it in a database. Broadly speaking, there are two kinds of databases: SQL and NoSQL.

SQL databases are used when you can fit your data nicely in tables. Even if your data originally is not in a tabular format, it is often possible to convert it to tabular format and save it in an SQL database. If you need an SQL database, my recommendation is to use PostgreSQL or SQLite.

On the other hand, broadly speaking, NoSQL is any database that is not SQL. As you can imagine, there is a great variety of NoSQL databases, e.g. document-based, key-value pairs, graph databases, or wide-column stores. The most popular NoSQL database is MongoDB.

In Python, you can use the following popular packages to connect to a database:

- SQLAlchemy for SQL
- PyMongo for MongoDB

## E.6 Big Data

- Dask is very similar to Pandas (in fact it builds on top of Pandas) and it can deal with datasets that are larger than your computer's memory. It can also parallelize some operations and thus speed up your calculations.
- Spark if your dataset is really huge. Often Spark is not necessary as Dask is sufficient.

## E.7 Finance

- Zipline for backtesting; other backtesting solutions include backtrader, pyalgotrade (or its successor basana), vectorbt, qstrader, pysystemtrade, and bt (in decreasing order of popularity as measured by GitHub stars). Some of these libraries are more actively maintained than others. If you are looking for a backtesting library that is actively maintained, you can check their recent commit history, e.g. go to the library's GitHub page, click "Insights" and then "Contributors."
- PyFolio for portfolio and risk analysis
- TA-lib for technical analysis

## E.8 Visualization and Plotting

See also Section H.2 starting on page 278 as well as Section G.7 starting on page 270.

- Bokeh is immensely popular and can visualize your data interactively in the browser. You could also combine Bokeh with Chartify, which is built on Bokeh.

  If you require interactivity, you should give Bokeh a try. On the other hand, for static plots it might be easier to get started with Matplotlib or Seaborn, for example.

- Matplotlib is another very popular plotting library in Python. One of the oldest plotting libraries in Python, it has been around since 2003 and therefore has received a lot of active development over the years. Matplotlib is good for basic plotting such as bars, pies, lines, and scatter plots.

- Plotly can create interactive graphs in the browser, similar in spirit to Bokeh. Plotly is based on the JavaScript library Plotly.js.

- Seaborn is for statistical data visualizations. It is based on Matplotlib. **If you would like to create statistical graphics, Seaborn is a very good starting point.**

- Altair is a declarative statistical visualization library.

- ggplot is a Python port of the wildly popular ggplot2 package from R.

- Holoviews for data visualization.

## E.9 Interactive Web Apps

Sometimes you would like to create a web app that showcases the resuls of your analysis.

- Superset or Redash are very popular, but might be less suitable for smaller apps.

- For simple web apps, you can also use Bokeh or Plotly, as mentioned in Section E.8 starting on page 235. Bokeh and Plotly are very good for interactive data visualizations.

- **Dash** is based on Plotly and can be used to build interactive web-based dashboards. If you're not sure which package to use, you won't make a mistake using Dash.

- Pyxley tries to be for Python what Shiny is for R

- Bokeh can also create interactive visualizations, although its main focus is more on plotting, not on creating web apps

- Jupyter notebook, maybe even with iPyWidgets for interactive widgets can be useful showing and visualizing what you're doing. It is more like a report that shows what you have done and it's less interactive, but if that's what you need, it's a great choice. Note that GitHub renders Jupyter notebooks, which is great if you put your code on GitHub anyway. You can also render a notebook on your computer (of course) but also at nbviewer.jupyter.org.

## E.10 Miscellaneous

- Statsmodels for statistics
- Delorean for date and time

## E.11 AI and Machine Learning

Artificial intelligence (AI) and machine learning are discussed in detail in Chapter 11 starting on page 115. Some of the useful Python packages include:

- Lime: If you need to explain to someone what your machine learning classifier is doing.
- SciKit-Learn: Great for ML, but for deep learning use another library.
- LightGBM and XGBoost: For distributed gradient boosting.
- Vowpal Wabbit (VW) is very popular, written mainly in C++ (with Python bindings), is pretty fast, and can deal with datasets that are larger-than memory.
- Annoy for approximate nearest neighbors.
- H2O.ai: Great for ML including deep learning. Written mostly in Java, has bindings for Python, R, Scala.
- MLlib on Spark if your dataset is truly huge

## E.12 Deep Learning

- See also Section 11.5 starting on page 122 for further discussions on deep learning.
- Note that deep learning can be though of as being a subset of machine learning, although in the Python ecosystem, deep learning is often spread out to separate packages.
- The Transformers library by Hugging Face provides thousands of pretrained models to perform tasks on different modalities such as text, vision, and audio.
- On the other hand, there are several libraries available if you would like to train your own deep learning model. We discuss them next.
- **My recommendation is to use Keras** if you're just getting started. The reason is that Keras is usable from a higher level than TensorFlow or PyTorch. This means you don't have to worry about so many details because the most common use cases are relatively easy to implement. On the other hand, if you have a very specific application that requires a customized approach, you might be better off using TensorFlow or PyTorch.

- TensorFlow by Google
- PyTorch by Meta AI (formerly Facebook AI)
- Keras has several backends (TensorFlow, PyTorch, and JAX), focusing on being user-friendly
- JAX by Google, it's supposed to fix some of TensorFlow's biggest pain points
- H2O.ai as mentioned above
- Chainer

There are also some "old" projects that deserve an honorable mention, although they are no longer actively developed and should not be used for new projects:

- MXNet by the Apache Software Foundation
- Theano: In maintenance mode since end of 2017; the main developer MILA stopped developing Theano

# Appendix F

# Web Scraping

This chapter covers the basics of web scraping. Web scraping deals with obtaining data from various pages on the internet. We cover this part relatively early on to enable you to start collecting the data you need at an early stage. If you would like to have an overview about which web scraping library to use, please see our previous discussion in Section E.2 starting on page 232.

## F.1 Primer on Web Technologies

Before we go into web scraping, we first need to understand the basics of the hyper text markup language (**HTML**). HTML is a markup language that tells your browser the structure of the webpage, e.g. it says that there should be a link to a given website or that some text should be displayed as a title or that a given JPEG file should be displayed as a picture. There are also cascading style sheets (**CSS**), which determine the visual layout of the webpage. Here we are mostly going to focus on HTML because for our purposes (i.e. NLP), this is where we find the information (i.e. text!) we're looking for. However, CSS (and maybe even XPath) might be important if you need to extract data from very specific parts of the webpage. We discuss this further below. For now we focus on HTML.

### F.1.1 HTML

Here is an example of a very simple HTML webpage. The code below can be found in the file `code-HTML-example.html` on the course website. You can even open this file right in your browser and **take a look how this webpage is rendered (i.e. displayed) in your browser**. It won't look fancy, but you get the point of how HTML works. The following HTML code can be found in the file `code-HTML-example.html` on the course website.

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Text Mining</title>
  </head>
  <body>
    <h1>Overview of Text Mining</h1>
    <p>
      Text mining, also referred to as text data mining, roughly
      equivalent to text analytics, is the process of deriving
      high-quality information from text. You can find more
      information on
      <a href="https://en.wikipedia.org/wiki/Text_mining">Wikipedia</a>
    </p>
  </body>
</html>
```

You can see that all HTML **tags** such as "`title`," "`h1`," or "`p`" are opened using "`<...>`", then something happens in between, and then the tags are closed again using "`</...>`". For example, <p> opens a "paragraph," then there is some text inside that paragraph, and then the tag gets closed again with </p> (note the forward slash "/"). Another thing that is important is that some HTML tags have **attributes**. For example, the "a" tag (which is used for creating links) has the "`href`" attribute. Some websites use another attribute, the "`id`" attribute, for some tags to give them a unique ID. If that is the case, you can use this ID to extract specific elements from the webpage (otherwise you need to use CSS selectors or XPath selectors, see Section F.1.4 on page 242).

### F.1.2 XML

Some of you might have heard about XML and that HTML has something to do with it. There is a long history about XML and HTML (and we're not going to go into it). The current state of affairs is that nowadays HTML5 (which is the current version of HTML) and XML are very similar. You can write a HTML document either in "HTML syntax" or in "XML syntax." The example above is written in HTML syntax, but could relatively easily be transformed into XML syntax with only minor modifications (mainly at the top of the file, the rest of the file does not require any changes). The difference between the two syntaxes is that the former is roughly 99% valid XML while the latter is 100% valid XML.

Although their files look very similar, XML and HTML have very different conceptual origins. XML is used to save data in a file that is both human-readable and machine-readable. **You can think of an XML document as a single-file mini-database.** On the other hand HTML is used to create webpages. Specifically, **HTML gives the browser the data and instructions needed to render a webpage.** It just so happens that HTML gives this data in a format that is very similar to XML. That's why HTML and XML are so similar, even though they have very different purposes. Now why are they so similar? Because both of them originate historically from the same root, which is another markup language called **SGML**.

Has it always been the case that HTML and XML are so similar? Yes and no. They have always been somewhat similar, but in the past HTML had diverged from XML quite a bit. The reason is that web browsers (e.g. Microsoft Edge, Mozilla Firefox, or Google Chrome) are relatively lenient when encountering malformed HTML. This has led to poor industry practices where some web designers for example wrote webpages where some HTML tags were not properly closed. In fact, you still might find websites in the wild with HTML that is strictly speaking invalid but still "works" because web browsers "understand" what the web developer is trying to do and still render the webpage, even if it contains invalid markup. This is, by the way, one reason why you need packages such as Beautiful Soup or lxml (discussed in Section F.3 starting on page 245) is because they fix invalid HTML. In any case, there is an active push towards cleaning up this mess, especially starting from HTML5, which requires the markup language to be very close (or identical) to valid XML.

### F.1.3 Document Object Model (DOM)

The Document Object Model, or **DOM** for short, is a way to represent HTML (or XML) in a tree structure. If you look at our HTML example from Section F.1.1 starting on page 239, you can see that for example the parent of the "`p`" tag is the "`body`" tag. And the "`body`" tag is the child of the "`html`" tag. So you could actually build a **tree structure** containing parents and children that represents the whole HTML document. In other words, **the DOM has the same information content as the HTML document, but this information is represented in a different way**, i.e. in a tree structure.

Now why do we need the DOM tree? There are at least two reasons for that:

1. First, it is useful to identify specific elements of the webpage. We will discuss this further in Section F.1.4 starting on page 242. Basically it can help you to **grab specific information you need from the webpage**. What happens is that the browser parses the HTML and internally represents the webpage as a DOM tree. So **the browser does actually not work with the HTML after it is parsed, but it actually works with the DOM internally**. Even though you might not directly refer to the DOM tree when grabbing data from the webpage (e.g. you are referring to an "`id`" attribute of a HTML tag or use a CSS selector or XPath selector), you are effectively using the DOM because that's how the webpage is represented internally.

2. Second, it is possible to change and update the DOM programmatically. Section F.1.6 starting on page 242 discusses this in more detail, but basically **you can write a JavaScript program that gets executed inside the browser and manipulates the DOM**. For example, using JavaScript, you can remove a specific HTML element such as a link. If that update of the DOM occurs, the browser will change how the webpage is displayed accordingly (e.g. the link will disappear from the displayed webpage). As you can imagine, this is a very powerful way of updating a webpage since you can change *any* part of the webpage according to your liking. So the key point is that **if you want to change the webpage, instead of loading a completely new HTML document (and thus reloading the whole webpage from the web server), you run a program inside the**

**browser that changes a specific element of the webpage (in the DOM) and leaves the rest of the webpage unchanged**. This can be faster to do and requires less new data to download.

### F.1.4 CSS and XPath Selectors

So given all that, how do we extract the information we need from a webpage? The simplest way is to simple get all the text from a website after stripping the HTML tags. On the other hand, if you are looking to extract very specific parts of a website, there are several approaches, but in my opinion the following usually works best:

1. First, you focus on **IDs** in the HTML markup.

2. If that doesn't work, you try **CSS selectors**. They are often more readable, brief, and concise than XPath.

3. Only if that doesn't work you use **XPath**. For example, walking *up* the DOM tree (i.e. going from a child to a parent node, see Section F.1.3 starting on page 241) is not possible with CSS. For XPath you need to use lxml because Beautiful Soup does not support it.

How do you know specifically which elements to choose? You can either look at the raw HTML, which might work for relatively small and simple webpages. If the webpage is more complicated, you can open it in your browser and inspect it there. You can press **F12 in Google Chrome** or you can use the **Firebug extension in Firefox**.

### F.1.5 HTTP

HTTP is the protocol through which your browser exchanges data with the web server. For example, if you point your browser to wikipedia.org, your browser uses HTTP to say: "Hello Wikipedia, please give me the files (mostly HTML files but potentially also image files etc.) for your title page." Wikipedia then sends the file(s) to the browser, again through the HTTP protocol.

### F.1.6 JavaScript and Ajax

Some webpages are difficult to scrape, especially if they use JavaScript and possibly AJAX. **JavaScript** is a programming language that your browser can execute. It is a whole programming language, so it is very different than HTML (which is a markup language). The difference between JavaScript and HTML is that JavaScript is dynamic, you can have for-loops, if/else statements, and so on, while with HTML it is much more static in the sense that it tells the browser the content of the website *once*, and then that's it.

So the basic idea with JavaScript is that you can make a the website more interactive by changing it on the fly **without having to reload the whole website** (i.e. without loading another HTML file from scratch from the server). For example, Facebook makes heavy use of JavaScript and has written famous JavaScript libraries such as React. And while JavaScript is already updating the website in the browser on the fly, it sometimes needs to exchange data with the web server (e.g. one of the computers run by Facebook). This is done through **Ajax**, which is a technique to use JavaScript to asynchronously **retrieve data in the background** without interfering with the display or behavior of the existing webpage. A common Ajax technique is to use the **XMLHttpRequest** object (or **XHR** in short) to transfer data between the browser and a web server. It is important to understand that **Ajax is not a new technology and not a new programming language. Ajax is simply a different way to use JavaScript in the web browser to communicate with a web server.**

As you can imagine, scraping webpages that use JavaScript and/or Ajax can seem difficult if you do not have any means to execute that JavaScript code. (For example, you cannot straightforwardly use the Requests library.) All you would get is some HTML that looks very differently than the HTML you would see rendered in your browser. The reason is that **the JavaScript running in your browser updates and changes the HTML** (strictly speaking, it changes the document object model, or **DOM** in short, see also Section F.1.3 starting on page 241). So without running JavaScript (and packages such as Requests cannot run JavaScript), it seems you are out of luck.

Luckily there are (at least) three solutions to this problem:

1. Don't just grab the HTML. Take a remote-controlled browser, point it towards the webpage, let it run the JavaScript (inside the browser), and after the webpage is fully loaded, collect the resulting HTML from the browser. This is the main idea of **Selenium** and a few other similar projects such as Watir (for Ruby) or CasperJS (for JavaScript). This approach can also be useful if you need to **fill out some forms or press some buttons** on the webpage to get to your data (no matter whether the webpage uses JavaScript or not).

2. Open your browser such as Chrome or Firefox (no Selenium needed here) and take a look at the **XHR communication** between your browser and the web server (**F12 in Chrome** or the **Firebug extension in Firefox**). Most likely you will see some JSON files (or XML files or other file formats in some cases) being sent back and forth. Most likely the XHR communication will be over HTTP, but it could also be another protocol. **The great thing is that often you will find that the Ajax interface is in fact really an undocumented API.** An API is a so-called "application programming interface," in this case an interface to **programmatically access the database behind the web server**. So instead of going through a lot of painful HTML parsing, you can simply **use this API to directly grab all the data you need**. In fact, due to this undocumented API, **scraping data from Ajax may actually make your life much easier** in some cases. You can use the **Requests** library (maybe together with Beautiful Soup to parse the HTTP response and with the json package to deal with the data in JSON format) to accomplish this by sending data (usually JSON data, sometimes XML or other data) back and forth via XHR. This sounds great, but here is a word of caution: It really depends on the website whether this approach works well. Often times it does, but some websites interleave and nest their Ajax so much that it's difficult to understand what's going

on. In these cases it might be easier to just use Selenium.

3. The website has an official API. For example, Twitter uses JavaScript and Ajax, but you don't have to worry about that because Twitter provides a separate API where you can download the tweets directly. See Section F.4 starting on page 248 for more details.

## F.2 Package Recommendations

1. If you need to scrape data from a specific web service such as Twitter, Facebook, Atom/RSS, etc., check some of the **specialized packages** listed in Section E.2 starting on page 232.

2. If you need to deal with JavaScript/Ajax-heavy sites and there is no specialized Python package available, use **Selenium**. If you like Requests, you can also combine Selenium with Requestium or Selenium-Requests (use Selenium to load the site, log in, fill out forms, etc., then Selenium-Requests moves all the cookies from Selenium into a Requests session where you can continue to use Requests as usual). Alternatively you could use requests_htlm, which can also deal with JavaScript.

3. If the website does not use a lot of problematic JavaScript/Ajax and if there is no specialized Python package available, check the following sequence whether it works for you:

   - If you just have to download and parse some webpages, use **Requests and Beautiful Soup** (or alternatively Requests and lxml if you prefer). Keep in mind that the `requests.Session` method (from the Requests package) allows you to persist parameters and cookies across requests made, which might come in handy if you need that sort of thing (although RoboBrowser might be easier in that case, see next bullet point).
   - If you need to "do" something with the webpage before getting your data (e.g. fill out forms, click buttons, logging in, etc.), use **RoboBrowser**.
   - If you need to crawl through the web, use **Scrapy**, maybe together with Scrapyz which tries to make Scrapy easier for simple spiders (or alternatively use PySpider or Pattern if you prefer).

4. Morph.io is great if you need to host your scraper in the cloud for free.

5. Important: No matter what you do, **do not overload the website you are trying to scrape**. If you hit it too often with your requests for data, it might thing you are an attacker and will shut down your IP address. In this case, you probably will have to wait for some time to access the website again.

## F.3 Web Scraping Examples

### F.3.1 Requests

We begin by downloading a webpage from Wikipedia and saving it to a file. This webpage does not use a lot of JavaScript, therefore it is fine to use the Requests package. The code is really straightforward, thanks to the simplicity of the Requests package. The code below can be found in the file `code-requests-scrape-wikipedia.py` on the course website.

```
# This script shows how to use the requests package to scrape a page
# from Wikipedia.
import requests
# Get the response. It is always a good idea to set the `timeout`
# argument in production code. Otherwise your program may hang
# indefinitely.
r = \
    requests.get(
        'https://en.wikipedia.org/wiki/Natural_language_processing',
        timeout=3)
# The following code block is strictly speaking not necessary, but it
# helps you to better understand the response you got.
r.raise_for_status()            # Ensure we notice bad responses.
r.status_code
r.headers['content-type']
r.encoding
print(r.text)                   # `print` gives nicer output for HTML.
r.text.encode('utf-8')          # Use specific encoding.
r.json()            # Doesn't work in this example since no JSON data.
# Finally we write the response content to file.
with open('data-wikipedia-NLP.html', mode='wb') as fd:
    fd.write(r.content)
```

### F.3.2 Beautiful Soup

The next thing we do is use the Beautiful Soup (bs4) package to extract certain elements from the webpage downloaded before. In this example, the webpage is downloaded by the Requests package, but you can use bs4 equally well on a webpage downloaded by other packages, e.g. Selenium (see Section F.3.3 starting on page 247).

bs4 does two things. First it parses the HTML of the website. This means it analyzes the tree structure inherent in the HTML and stores this analysis for later processing. This step is done "automatically" in bs4 when using the `BeautifulSoup` class. Second, bs4 can extract certain elements from the webpage. The code below can be found in the file `code-bs4-read-wikipedia.py` on the course website.

```
# This script shows how to download and parse a Wikipedia page using
# Requests and Beautiful Soup. We also use the `re` module for regular
# expressions.
import re
import requests
from bs4 import BeautifulSoup
r = \
    requests.get(
        'https://en.wikipedia.org/wiki/Natural_language_processing',
        timeout=3)
s = BeautifulSoup(r.text, 'lxml') # Use `lxml` to parse the webpage.

# Take a look at the parsed webpage.
print(s.prettify())

# Extract all the text from the webpage. THIS IS WHAT YOU NEED MOST
# OFTEN FOR NLP AND TEXT ANALYTICS, unless you need to extract only
# part of the webpage.
print(s.get_text())

# Ways to navigate the data structure by HTML tag. This will find the
# FIRST tag of that name (not ALL the tags of that name) in the HTML
# document.
s.title
s.title.name
s.title.string
s.title.parent.name
s.p                # First 'p' tag.
s.p.get_text()
s.p['class']       # In our example, there's no 'class' HTML attribute.
s.header['class']
s.a                # First 'a' tag.

# With the `find` and `find_all` methods you can dive deeper into the
# HTML and have more control over what you extract. The difference
# between `find` and the `find_all` is that the former only finds the
# FIRST child of this tag matching the given criterial, while the
# latter gets ALL of them.
#
# We start with a demonstration of `find`. HTML tags may have
# attributes, e.g. an 'a' tag usually has a 'href' attribute as in `<a
# href="...">...</a>`. In the following line of code we find the first
# tag that has an 'id' attribute with value 'footer'. This is probably
# the easiest way to extract part of a webpage (assuming the part of
# the webpage you would like to extract has a corresponding
# attribute).
s.find(id='footer')
# Alternatively you can also use a CSS selector using the `select`
# method. This works no matter whether the part you're trying to
# extract has tag with a specific attribute or not.
s.select('#footer')
# Keep in mind that XPath is not directly supported by BeautifulSoup.

# Extract all 'a' tags.
atags = s.find_all('a')          # Find all `<a ...>...</a>` tags.
atags[3]
atags[3].name
atags[3].get('href') # Get the actual `href` attribute (i.e. the URL).

# If we want to get all `href` attributes, we loop over all `a` tags.
{tag.get('href') for tag in s.find_all('a')}

# Find all tags whose names start with the letter 'b' (in this case
# 'body', 'b', and 'br').
{tag.name for tag in s.find_all(re.compile('^b'))}

# Find all tags whose name contains the letter 't'.
{tag.name for tag in s.find_all(re.compile('t'))}

# We can also pass a list to the `find_all` method, in which case bs4
# allows a string match against any item in that list.
{tag.name for tag in s.find_all(['a', 'body'])}

# Find all the tags in the document, but none of the text
# strings. `True` matches anything it can.
{tag.name for tag in s.find_all(True)}
```

### F.3.3 Selenium

The previous example using the Requests package works well on webpages that don't use a lot of JavaScript. However, when a webpage includes a significant amount of JavaScript, the Requests package might no longer work so well because it cannot easily execute the JavaScript to produce the final HTML version of the website. To solve this problem, we can use Selenium WebDriver (using the `selenium-python` package) instead of Requests to obtain the data we need. After we have downloaded the desired HTML file using Selenium WebDriver, we can then extract further data from the HTML file using Beautiful Soup, similar to the previous example. Before running the following code, you need to download the `chromedriver` file from chromedriver.chromium.org. Make sure that the version of your `chromedriver` file matches the version of your Chrome browser, otherwise you will receive an error message. (Selenium WebDriver also works with browsers other then Chrome. We just use Chrome here because it is very widely used.) The code below can be found in the file `code-selenium-download-google.py` on the course website.

```
# This script shows how to navigate and download a website
# (i.e. Google in the code below) that potentially includes a lot of
# JavaScript. We use Selenium WebDriver for web browser
# automation. This code is only intended for academic purposes.
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.by import By
import os

home = os.path.expanduser('~')  # Home directory/folder.
# You might have to update the following line, depending on where you
# have saved the `chromedriver` file during installation.
driver = webdriver.Chrome(service=Service(home + '/bin/chromedriver'))

# Go to the Google website.
driver.get('https://www.google.com')

# Enter a search string. In case the following command does not work,
# you need to check the XPath of the Google search box and update it
# in the code below. Google might sometimes change its website
# slightly, which can result in a different XPath that needs to be
# updated in your code.
e = \
    driver.find_element(
        By.XPATH,
        '/html/body/div[1]/div[3]/form/div[1]/div[1]/div[1]/div/div[2]/input')
e.send_keys('NLP trends')        # Enter search string into Google.
e.send_keys(Keys.ENTER)          # Hit the "Enter" key (in browser).

# Go back to Google homepage by clicking on "Google" logo.
driver.find_element(By.XPATH, '//*[@id="logo"]/img').click()

# Download the HTML of the webpage for further analysis. Keep in mind
# that this only downloads the HTML of the website, not any CSS or
# JavaScript. After you have downloaded the HTML, you can extract data
# from it, e.g. using Beautiful Soup (shown elsewhere in this book).
with open('data-google-page.html', 'w') as f:
    f.write(driver.page_source)
```

## F.4 Mining Twitter

In the previous sections we have seen how to manually scrape content from a website. Twitter also has a website, so we could use these techniques on Twitter as well. However, it might be cumbersome due to the heavy usage of JavaScript and Ajax on Twitter.

Luckily we do not have to worry about this problem because Twitter provides an official API (the “Twitter API”) we can use to programmatically download Twitter content.

Furthermore, there are several high-quality packages available that allow us to interface to this Twitter API with relative ease (see also Section E.2 starting on page 232).

Among these packages, Tweepy is one of the most popular. Tweepy uses the Twitter API, which is an interface provided by Twitter. You can use this interface to access Twitter using a program, e.g. a Python script written by you. To obtain access to the Twitter API, you first need to sign up for a (regular) Twitter account, in case you haven't already got one. Second, you need to sign up for a developer account.

There are different access levels to the Twitter API. When signing up for a developer account, you will by default get Essential access, which only allows you to connect to Twitter API v2, but not to Twitter API v1.1. To be able to connect to Twitter API v1.1, you need to apply for Elevated, Elevated+, or Academic Research access. You can apply for this access from within the developer portal after you have signed up for a developer account. In what follows we will focus on the Twitter API v1.1.

After you have set up your Twitter developer account, you need to save the following information given to you by Twitter when you create a "Project" and an "App" on the Developer Portal:

- API Key and API Key Secret: These are the user name and password representing your App
- Access Token and Access Token Secret: These specify the Twitter account the request is made on behalf of

For easy reference, you can save them in a file that you can read into Python whenever you need to. In this example, I save them in a file called `code-API-key-tweepy.py`:

```
consumer_key = '...'           # Here you enter the API Key.
consumer_secret = '...'        # Here you enter the API Key Secret.
access_token = '...'           # Here you enter the Access Token.
access_token_secret = '...'    # Here you enter the Access Token Secret.
```

You can later read this file into Python by running

```
exec(open('code-API-key-tweepy.py').read())
```

If you have saved this file in a different directory, you need to add the location of the file before the file name. In the following example, the file is saved in the parent directory (represented by "..."), so we prepend "..." to the filename:

```
exec(open('../code-API-key-tweepy.py').read())
```

If you would like to obtain a better overview of the data provided by Twitter, you can take a look at Twitter's data dictionary. Specifically, the documentation about the user object contains information about the various fields such as `user.screen_name` or `user.followers_count` that we'll use in some of the Tweepy examples in the section.

In order to connect to Twitter, you supply all of your authentication material to Tweepy, which then handles the connection to Twitter for you (see the file `code-tweepy-010-setting-u` on the course website):

```
# This script shows how you can set up your Twitter API login
# information.
import tweepy

# Load Twitter API authentication data.
exec(open('../code-API-key-tweepy.py').read())

auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)

# Print the screen name / user name of your account.
print(api.verify_credentials().screen_name)
# Print the tweets in your stream.
public_tweets = api.home_timeline()
for tweet in public_tweets:
    print(tweet.text)
```

If you want to listen to what is happening on Twitter for a given keyword or hashtag, you can use the following code (see the file `code-tweepy-020-streaming-api.py` on the course website):

```
# This script listens on the Twitter stream for the keyword 'python'
# and prints select fields from each tweet, such as name, follower
# count, date, and the tweet itself.
import tweepy

# Load Twitter API authentication data.
exec(open('../code-API-key-tweepy.py').read())

# Define subclass of `tweepy.Stream` to add logic to `on_status`. See
# http://docs.tweepy.org/en/latest/extended_tweets.html.
class listener(tweepy.Stream):
    def on_status(self, status):
        print(
            status.user.screen_name, ':',
            status.user.followers_count, ':',
            status.created_at, ':')            # UTC time.
        if hasattr(status, 'retweeted_status'): # Check if it's a retweet.
            try:
                print(status.retweeted_status.extended_tweet['full_text'])
            except AttributeError:
                print(status.retweeted_status.text)
        else:
            try:
                print(status.extended_tweet['full_text'])
            except AttributeError:
                print(status.text)
```

```
# Create a Stream.
mystream = \
    listener(
        consumer_key,
        consumer_secret,
        access_token,
        access_token_secret)
# Start a Stream.
mystream.filter(track=['python'])
```

Sometimes you might just want to save the ongoing Twitter stream to a file. This is also easy to do (see the file code-tweepy-030-stream-to-file.py on the course website):

```
# This script listens to a keyword (e.g. 'python' in the example
# below) and writes the whole Twitter stream to a file.
import tweepy

# Load Twitter API authentication data.
exec(open('../code-API-key-tweepy.py').read())

class listener(tweepy.Stream):
    def on_status(self, status):
        with open('data-streaming-tweets.txt', 'a') as f:
            f.write(str(status) + '\n\n')

    def on_request_error(self, status_code):
        print(status_code)

mystream = \
    listener(
        consumer_key,
        consumer_secret,
        access_token,
        access_token_secret)
mystream.filter(track=['python'])
```

Alternatively, you can also stream only a selection of attributes of the tweet. The difference to the previous example is that here we are using on_status, which is usually what you need, unless you want to stream as much data as possible (in which you can consider using on_data). Another difference is that we are looking for mentions of stock index codes (e.g. $SPX for the S&P 500) and related codes. This ensures that we are picking up the relevant finance Twitter stream. The following code can be found in the file code-tweepy-040-stream-select-fields-to-file.py on the course website.

```
# This script shows how to stream only selected fields to a file. The
# fields are separated by ' : ' (i.e. space, colon, space) for easier
# parsing later on.
import tweepy

# Load Twitter API authentication data.
exec(open('../code-API-key-tweepy.py').read())

# Define custom listener class that is a derived class from the
# `tweepy.Stream` base class. It writes specific data fields from the
# Twitter stream to a file.
class listener(tweepy.Stream):
    def on_status(self, status):
        with open('data-streaming-tweets.txt', 'a') as f:
            if hasattr(status, 'retweeted_status'): # Check if it's a retweet.
                try:
                    mytweet = status.retweeted_status.extended_tweet['full_text']
                except AttributeError:
                    mytweet = status.retweeted_status.text
            else:
                try:
                    mytweet = status.extended_tweet['full_text']
                except AttributeError:
                    mytweet = status.text
            f.write(            # Write the data to file. UTC time.
                status.user.screen_name + ' : ' + \
                str(status.user.followers_count) + ' : ' + \
                str(status.created_at) + ' : ' + \
                ' '.join(mytweet.split()) + '\n')

    def on_request_error(self, status_code):
        print(status_code)

# Instantiate an object of class `listener` (which is a subclass of
# `tweepy.Stream`).
mystream = \
    listener(
        consumer_key,
        consumer_secret,
        access_token,
        access_token_secret)
# Filter the stream for certain keywords. Here we filter tweets that
# mention a select stock index/ETF, fixed income index/ETF, or a
# commodities index/ETF. More tickers and/or keywords could be added
# here.
mystream.filter(
    track=[
        '$SPX', '$SPY', '$ES',
        '$DJI', '$DJIA', '$INDU', '$YM',
        '$NQ', '$NASDAQ', '$QQQ',
        '$TLT',
        '$GC', '$GLD',
        '$NG', '$WTI'])
```

Sometimes you just want to download all the tweets from a given user. You can do this with relative ease and then save the result to a CSV file that you can later process separately (see the file `code-tweepy-050-download-all-tweets-from-user.py` on the course website). If you are looking for interesting people to follow, you can google "top finance twitter accounts."

```
# Twitter only allows access to a users most recent 3240 tweets
# with this method.
import tweepy
import csv

# Twitter user you want to download.
screen_name = 'StockCats'

# Load Twitter API authentication data.
exec(open('../code-API-key-tweepy.py').read())

# Authorize the Twitter API.
auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
auth.set_access_token(access_token, access_token_secret)
api = tweepy.API(auth)

# Initialize a list to hold all the tweets.
alltweets = []
# Make initial request for most recent tweets (200 is the maximum
# allowed count).
new_tweets = \
    api.user_timeline(
        screen_name=screen_name,
        count=200)
# Save most recent tweets to the list.
alltweets.extend(new_tweets)
# Save the id of the oldest tweet less one.
oldest = alltweets[-1].id - 1
# Keep grabbing tweets until there are no tweets left to grab.
while len(new_tweets) > 0:
    print("getting␣tweets␣before␣%s" % oldest)
    # All subsequent requests use the max_id param to prevent
    # duplicates.
    new_tweets = \
        api.user_timeline(
            screen_name=screen_name,
            count=200,
            max_id=oldest)
    # Save most recent tweets.
    alltweets.extend(new_tweets)
    # Update the id of the oldest tweet less one.
    oldest = alltweets[-1].id - 1
    print("...%s tweets downloaded so far" % len(alltweets))

# Use list comprehension to transform the tweets into a 2D array
# (list of row objects) that will populate the CSV file.
outtweets = \
    [[tweet.id_str, tweet.created_at, tweet.text.encode("utf-8")]
     for tweet in alltweets]
# Write to CSV file.
with open('data-%s-tweets.csv' % screen_name, 'w') as f:
    writer = csv.writer(f)
    writer.writerow(["id", "created_at", "text"]) # Column names.
    writer.writerows(outtweets)
```

# Appendix G

# Data Munging with Pandas

## G.1 Introduction to Pandas

In this section we take a closer look at the Pandas Python package. When Pandas was created, the key idea was to enable Python users to perform similar tasks as were already common in R. The most important element was to carry the data.frame concept from R (i.e. rectangular data where columns can be of different data types) over to Python. It is fair to say that Pandas is probably the cornerstone of data science in Python and the main reason why data science has caught on in Python so well in the past. The creator of Pandas is Wes McKinney, an MIT graduate who wrote the first Pandas versions while working at AQR, a famous quantitative hedge fund.

Pandas can be useful in at least three ways.

1. You have analyzed your textual data and have obtained some sort of scores or values based on the textual analysis, e.g. sentiment scores. You can then analyze them in Pandas.

2. Your textual data is saved as tidy text in a Pandas DataFrame and you would like to perform further analysis on this data. See Section 17.2 starting on page 189 for details about tidy text.

3. If you have financial data, e.g. stock market data from AlphaVantage (see also Section G.8 starting on page 271) or accounting data from Morningstar, you can merge them together and analyze them with your textual data.

Pandas is built on top of NumPy. There are **two main differences between NumPy and Pandas**. The first is that Pandas' data structures have labels, e.g. the DataFrame has row and column labels, while a two-dimensional NumPy array does not. Second, Pandas data structures can contain different data types, e.g. the first column in a Pandas data frame might contain numeric data while the second column contains character strings. On the other hand a NumPy array usually only contain a single data type, although it is also possible to have several data types in a NumPy array, e.g. with the `object` dtype or with structured arrays.

Pandas has a **Series**, which is a one-dimensional labeled array, a **DataFrame**, which is a two-dimensional table with row and column labels. Pandas also has a **Panel**, which is a labeled three-dimensional data structure. Think for example of a dataset containing time series of sales and net income for several stocks such as IBM and Apple.

## G.2 Introductory Pandas Example

Here we illustrate basic pandas usage. The code below can be found in the file `code-pandas-010-intro.py` on the course website.

```
# This script gives a brief introduction on pandas in Python.

import pandas as pd
import numpy as np

# Very simple DataFrame with only one column.
pd.DataFrame(
    data=[3, 5, 6, 7],          # Data input is a list.
    columns=['MyColumnName'])

# Use `columns=...` to rearrange the order of the columns.
pd.DataFrame(
    {'first_name': ['Jason', 'Molly', 'Tina', 'Jake', 'Amy'],
     'last_name': ['Miller', 'Jacobson', ".", 'Milner', 'Cooze']},
    columns=['last_name', 'first_name'])

# The columns of a DataFrame can hold data of different data
# types. Here we are using a dictionary as an input. The keys of the
# dict will become the column names of the DataFrame, while the values
# of the dict will become the column entries of the DataFrame.
df = \
    pd.DataFrame(
        { 'A': 1.,          # Will be repeated to fill the whole column.
          'B': pd.Timestamp('2019-12-20'), # Will be repeated.
          'C': pd.Series(1, index=list(range(5)), dtype='float32'),
          'D': np.array([3] * 5, dtype='int32'),
          'E': ['Jason', 'Molly', 'Tina', 'Jake', 'Jeff'],
          'F': pd.Categorical(['test', 'train', 'test', 'train', 'train'])})
df.dtypes                       # Show the data types of the columns.
```

## G.3 Writing/Loading a DataFrame To/From a File

- As usual we import the pandas package at the beginning:

```
import pandas as pd
```

- You can save and load a DataFrame to a file. The simplest format is comma-separate values (**CSV**), which also lets you easily interchange the data with other programs.

  ```
  df.to_csv('example-data.csv')         # Save data to a CSV file.
  df = pd.read_csv('example-data.csv')  # Load from CSV file.
  ```

- For an example of how a CSV file looks like, see Section G.4 starting on page 258.
- One problem with CSV is that it does not retain information on **metadata**. For example, just by looking at the CSV file you can only guess what the original data types of the columns are. So CSV is good if you want to exchange your data with other programs, but if you just want to **save and reload your data in Python**, you can **pickle** it (which is a way to serialize the object and save it to disk, see also Section 6.10 starting on page 80). If you use the following code, you only need to import pandas and you do not need to separately import pickle because it is already loaded by Pandas as a dependency.

  ```
  df.to_pickle('example-data.pkl')         # Save DataFrame ‘df‘ to file.
  df = pd.read_pickle('example-data.pkl')  # Load DataFrame from file.
  ```

- If you need to exchange DataFrame data with R (or even if you use it only in Python and want a different way to serialize the data), you can save it in the **feather format**:

  ```
  df.to_feather('example-data.feather')
  df = pd.read_feather('example-data.feather')
  ```

  In R, you can write and read the data as follows:

  ```
  library('feather')
  write_feather(df, 'example-data.feather')
  df = read_feather('example-data.feather')
  ```

- If your Pandas data is very large, you can also store it to the **HDF5 format** via the **PyTables** package. This format is good if you have data exceeding the size of your computer memory. In this case, HDF5 is a way to store data efficiently and retrieve only the parts of the data that you need to work on. It is similar in spirit to a database.

  ```
  # Writing to HDF5 with ‘mode='a'‘ to append the data.
  # If you want to overwrite, use ‘mode='w'‘.
  my_df.to_hdf('example.h5', key='df', mode='a', format='table')
  # Reading from HDF5 and retrieve a subset with ‘query‘.
  df_read = pd.read_hdf('example.h5', key='df', where=['B > 50'])
  ```

## G.4 Basic Pandas Usage

Below we show how to slice and dice data saved as a CSV file. It is an excerpt from the Center for Research in Security Prices (CRSP) from the University of Chicago. The data file `data-CRSP-extract.csv` can be found on the course website. The variables included are:

1. TICKER: The ticker symbol of the stock.
2. date: The date in which the stock traded.
3. PRC: The stock price.
4. RET: The stock return.
5. VOL: Number of shares traded in millions.
6. SHROUT: Number of shares outstanding.

The beginning of the CSV file looks like this, so you can see that different columns are separated by commas:

```
TICKER,date,PRC,RET,VOL,SHROUT
AAPL,2014-01-02,553.13,-0.014064,8338094,892.447
AAPL,2014-01-03,540.97998,-0.021966,13992006,892.447
AAPL,2014-01-06,543.92999,0.005453,14820614,892.447
AAPL,2014-01-07,540.03748,-0.007156,11381939,892.447
```

After importing into Pandas (see the following code example, in particular the `read_csv` function), the beginning of the dataset looks like this:

```
  TICKER        date        PRC       RET       VOL   SHROUT
0   AAPL 2014-01-02  553.13000 -0.014064   8338094  892.447
1   AAPL 2014-01-03  540.97998 -0.021966  13992006  892.447
2   AAPL 2014-01-06  543.92999  0.005453  14820614  892.447
3   AAPL 2014-01-07  540.03748 -0.007156  11381939  892.447
```

The following code contains in-line comments that explain what's going on. The code below can be found in the file `code-pandas-020-longer-intro.py` on the course website and the data file we use is also on our course website.

```
import pandas as pd
import numpy as np

df = pd.read_csv('data-CRSP-extract.csv')
df.dtypes

# Selecting column(s).
df['TICKER']          # Select one column, resulting in a Pandas Series.
df.TICKER             # Same as df['TICKER'], returns a Series.
```

```
df[['TICKER']]       # Returns DataFrame. Inner brackets are a list.
df[['TICKER', 'PRC']]          # DataFrame with two columns.

# Convert date to a different data type, i.e. datetime.
df['date'] = pd.to_datetime(df['date'])

# Now you can do arithmetics with the date, e.g. add one day. This is
# not possible if the data type is `object` (e.g. a string) as it was
# before.
df['date'] + np.timedelta64(1, 'D')

# Basic information about the DataFrame.
type(df)
df.columns                  # Take a look at the column names.
list(df)                    # Column names, similar to `df.columns`.
df.shape                    # Dimensions.
len(df.index)               # Number of rows, same as `len(df)`.
len(df.columns)             # Number of columns.
df.describe()               # Compute some summary statistics.

# Slicing, i.e. selecting rows.
df.head()                    # First five rows by default.
df.head(3)                   # First three rows.
df[:3]                       # First three rows.
df.iloc[:3]                  # First three rows.

df.tail()                    # Last five rows by default.
df.tail(3)                   # Last three rows.
df[-3:]                      # Last three rows.
df.iloc[-3:]                 # Last three rows.

# Query the data, i.e. we select rows that satisfy certain conditions.
df[df.TICKER == 'FB']                      # Extract Facebook.
df.query("TICKER == 'FB'")                 # Extract Facebook.
df[(df.TICKER == 'FB') & (df.PRC > 120)] # Facebook with price>120.
df.query("TICKER == 'FB' & PRC > 120")   # Facebook with price>120.
df[(df.TICKER == 'FB') | (df.PRC < 120)] # Facebook or price<120 (or both).
df.query("TICKER == 'FB' | PRC < 120") # Facebook or price<120 (or both).
df[df.TICKER.isin(['AAPL', 'FB'])]      # Apple and Facebook.
df.TICKER.unique()                        # All tickers.
df[['TICKER', 'SHROUT']].drop_duplicates() # Two columns w/ unique entries.

# Modify the data. The code below creates a new DataFrame that is
# changed according to which DataFrame method is called,
# e.g. `rename()`, `drop()`, or `sort_values()`. By default, however,
# these methods do not change `df` itsefl. Instead, if you would like
# to modify `df` directly, you can use the `inplace=True` parameter.
```

```
df.rename(columns={'VOL': 'Volume', 'PRC': 'Price'}) # Two columns renamed.
df.drop('TICKER', axis=1)        # New DataFrame without this column.
df.sort_values(by='PRC')         # Sort increasing by PRC.
df.sort_values(by='PRC', ascending=False) # Sort decreasing by PRC.
df.sort_values(by=['SHROUT', 'PRC'], ascending=[False, True])

# The `assign()` method creates a new DataFrame containing the
# changes. For example, we create a new column. The existing `df`
# DataFrame is not modified. `assign()` can be useful if we would like
# to create a new column that subsequently should be used for grouping
# the data (see the `groupby()` example below).
df.assign(HIGH_PRC = (df.PRC > 120))    # New column created.
df.assign(abc = df.PRC + 8 * df.SHROUT) # New column.
# `assign` does not have the `inplace=True` parameter. If you would
# like to modify `df`, you could assign the values to the new column
# directly, for example:
# df['abc'] = df.PRC + 8 * df.SHROUT

# Apply a function to each column of the DataFrame. (If you would like
# to apply a function to each row of a DataFrame, use the `axis=1`
# parameter.) As the functions below expect numeric types, we only
# apply them to the price (PRC) and stock return (RET) columns for
# illustration.
#
# In the first example, we use the `amax()` function (called as
# `np.amax()`) from NumPy, as it is faster than the built-in `max()`
# function of Python. Note that you will often see `np.max()` in code
# of other people, this is simply an alias for `np.amax()` and behaves
# exactly the same. Note that below we pass for example `np.amax` and
# not `np.amax()` to `apply()` because we want to pass the function
# itself to `apply()`, i.e. we do not want invoke the function
# directly (the function would be invoked later by `apply()`).
df[['PRC', 'RET']].apply(np.amax) # Maximum of each column.
df[['PRC', 'RET']].apply(lambda x: x.max() - x.min()) # Range.
df[['PRC', 'RET']].apply(lambda x: x + 3) # Add three to each column.

# Aggregating (summarizing) the data. Here we use the `max()` and
# `min()` methods of the Pandas Series.
df.PRC.max() # Maximum price.
df.PRC.min() # Minimum price.

# Aggregating using the `agg()` method. The difference to `apply()` is
# that `agg()` is often used to apply multiple aggregation functions
# at once. Below we use the string representations of the aggregating
# functions, e.g. `'max'` or `'min'`, which refer to the built-in
# Pandas methods.
df.agg('max')                    # `np.max()` applied to all columns.
df.agg(['max', 'min'])          # Max and min applied to all columns.
df.agg(                         # Different aggregations per column.
    {'RET': ['max', 'min'],
     'VOL': ['min', 'mean', 'sum']})

# Group by ticker and calculate the maximum and minimum stock returns
# for each group.
df.\
    groupby('TICKER').\
    agg({'RET': ['max', 'min']})
# Compute average stock return and stock price for each stock and
# month. Using the `reset_index()` at the end is optional, it puts the
# `TICKER` and `date` into a separate column instead of using them as
# the index (i.e. the row names).
df.\
    groupby(['TICKER', pd.Grouper(key='date', freq='ME')])\
    [['RET', 'PRC']].\
    agg('mean').\
    reset_index()

# Count how many observations have a high price (i.e. PRC>120).
df.\
    assign(HIGH_PRC = (df.PRC > 120)).\
    groupby('HIGH_PRC').\
    agg({'HIGH_PRC': 'count'}).\
    rename(columns={'HIGH_PRC': 'Count'})

# Total trading volume for each ticker, assigned to a new
# column. `transform()` is similar to `agg()` in that it boils down
# the input to one single number. However, unlike `agg()`,
# `transform()` will repeat this number to fill the column.
df['TVOL'] = \
    df['VOL'].\
    groupby(df['TICKER']).\
    transform('sum')

# Removing columns.
del df['TVOL']                  # Delete column.
df.head()                       # Column is gone.
# df.drop('TVOL', axis=1, inplace=True) # Alternative way to delete column.
# df.drop(df.columns[[0, 1, 3]], axis=1, inplace=True) # Delete by column number.
```

## G.5 Reshaping the Data

Sometimes we would like to change the way the DataFrame looks like by reshaping it. For example, suppose your data has the time series of stock returns for different stocks "stacked" on top of each other:

```
     TICKER        date       RET
0      AAPL 2014-01-02 -0.014064
1      AAPL 2014-01-03 -0.021966
2      AAPL 2014-01-06  0.005453
3      AAPL 2014-01-07 -0.007156
4      AAPL 2014-01-08  0.006338
...     ...         ...       ...
2263     FB 2016-12-23 -0.001107
2264     FB 2016-12-27  0.006310
2265     FB 2016-12-28 -0.009237
2266     FB 2016-12-29 -0.004875
2267     FB 2016-12-30 -0.011173
```

Now let's say you would like to "unstack" your data. Specifically, each stock ticker should show the stock returns in a separate column. The "unstacked" data would look as follows; you can see that the actual data entries have been preserved, although they now take on a different shape:

```
             date      AAPL        FB      MSFT
0      2014-01-02 -0.014064  0.001116 -0.006683
1      2014-01-03 -0.021966 -0.002797 -0.006728
2      2014-01-06  0.005453  0.048445 -0.021132
3      2014-01-07 -0.007156  0.012587  0.007750
4      2014-01-08  0.006338  0.005352 -0.017852
..            ...       ...       ...       ...
751    2016-12-23  0.001978 -0.001107 -0.004878
752    2016-12-27  0.006351  0.006310  0.000632
753    2016-12-28 -0.004264 -0.009237 -0.004583
754    2016-12-29 -0.000257 -0.004875 -0.001429
755    2016-12-30 -0.007796 -0.011173 -0.012083
```

To achieve these or similar reshaping of your data, you can use the `melt()` and `pivot()` DataFrame methods of Pandas. Alternatively, you could also use the `stack()` and `unstack()` DataFrame methods, which work in similar ways. The main difference between melt/pivot and stack/unstack is that the former reshape the data based on column values, while the latter reshape the data based on index labels. The code below can be found in the file `code-pandas-030-reshaping-the-data.py` on the course website.

```python
# To reshape the way your data looks like, you can use the melt/pivot
# methods as well as the similar stack/unstack methods. We will focus
# on melt/pivot in the examples below.

import pandas as pd
import numpy as np
df = pd.read_csv('data-CRSP-extract.csv')
```

```
df['date'] = pd.to_datetime(df['date'])

# In the first example, we pivot (unstack) `df` to put the returns of
# each stock ticker in a separate column. Afterwards we partially
# revert the previous pivot by melting (stacking) the stock returns
# into their previous shape.
pivot_df = \
    df.\
    pivot(
        index='date',
        columns='TICKER',
        values='RET').\
    reset_index()
print(pivot_df)
pivot_df.melt(id_vars=['date'])

# In the second example we first melt (stack) and then pivot (unstack)
# the DataFrame.
melted_df = df.melt(id_vars=['TICKER', 'date'])
print(melted_df)
# Reverse the previous `melt()`.
melted_df.\
    pivot(
        index=['TICKER', 'date'],
        columns='variable',
        values='value').\
    reset_index()

# For each stock, compute the means of each variable.
melted_df.\
    groupby(['TICKER', 'variable']).\
    mean()                        # Alternatively: `agg('mean')`
# Of course this could have been done directly with `df` as well, for
# example:
df.groupby('TICKER').mean()
# We just wanted to show an example computation you could do on the
# melted DataFrame `melted_df`.
```

## G.6 Merging Two Datasets

We can also merge two DataFrames. Merging (or "joining" as it is sometimes called) means bringing two Pandas DataFrames together. This can be useful not only if you have two DataFrames, but also when you are aggregating (i.e. summarizing) data.

### G.6.1 Using merge() for Data Aggregation

In the following example we have daily CRSP data and would like to add a new column that shows the monthly trading volume. We thus **aggregate** the data in the sense that several data points (e.g. daily trading volume) are summarized into fewer data points (e.g. monthly trading volume). Afterwards we merge the monthly trading volume back to the original daily dataset.

Please take a look at the comments in the code that explain what's going on. The code below can be found in the file `code-pandas-040-monthly-trading-volume.py` on the course website and the data files we use are also on our course website.

```
# This script shows how to calculate monthly trading volume and merge
# it back to daily data.

import pandas as pd
import numpy as np

df = pd.read_csv('data-CRSP-extract.csv') # Import from CSV file.
df['date'] = pd.to_datetime(df['date'])   # Convert to date (and time).
df = df[['TICKER', 'date', 'PRC', 'VOL']] # Just need a few columns.

# We could accomplish the computations in this script in one line,
# using `groupby()` and `transform()`, as shown below. But that's
# beside the point. The point of this script is to demonstrate how to
# merge two datasets.
newdf = df.copy()
newdf['MVOL'] = \
    newdf.\
    groupby(['TICKER', pd.Grouper(key='date', freq='ME')])\
    ['VOL'].\
    transform('sum')
print(newdf)

# Calculate monthly trading volume. We first extract the columns we
# need from the original `df` DataFrame, group it by ticker and month,
# and sum up the (monthly) volume.
mvol = df[['TICKER', 'date', 'VOL']]
mvol = mvol.groupby(['TICKER', pd.Grouper(key='date', freq='ME')]).sum().reset_inde
mvol.rename(columns={'VOL': 'MVOL'}, inplace=True) # Rename column.

# Take a look at the data. You can see that here we have ONE
# observation per ticker and per month.
mvol.tail()

# Here we create a column in each DataFrame that has the day removed,
# so we only have year and month. This is necessary for merging both
# DataFrames (`df` and `mvol`) together in a later step where we use
# the year-month information to link up both DataFrames.
```

```
mvol['mdate'] = mvol.date.dt.to_period('M') # Convert to monthly dates.
del mvol['date']                             # Not needed any more.
df['mdate'] = df.date.dt.to_period('M') # Convert to monthly dates.

# Convert year-month to string. Although not strictly necessary, it
# makes some things easier to handle, e.g. querying a DataFrame in the
# subsequent step.
df['mdate'] = df.mdate.astype(str)
mvol['mdate'] = mvol.mdate.astype(str)

# Here we select some subsets of both DataFrames. In production, you
# would SKIP this step. Here we use it only to better illustrate the
# following merge to have a small result we can actually look at and
# inspect.
df = df.query("TICKER == 'AAPL' & date >= '2014-12-20' & date <= '2015-01-15'")
mvol = mvol[(mvol.TICKER == 'AAPL') & (mvol.mdate >= '2014-11') & (mvol.mdate <= '2

# Next we merge both DataFrames together using the ticker symbol
# (`TICKER`) and a column representing the year-month (`mdate`). Here
# we are using a so-called "left join," which is for our purposes the
# most important kind of join. It means that we are adding data to the
# "left" DataFrame `df` (instead of the "right" DataFrame `mvol`). The
# data added comes from the "right" DataFrame `mvol`. We use "left"
# and "right" because in the call to `pd.merge`, the `df` DataFrame
# comes first, so is on the "left" side, while `mvol` is on the
# "right" side.
df_mvol = pd.merge(df, mvol, how='left', on=['TICKER', 'mdate'])

# Take a look at the two original DataFrames `df` and `mvol` and the
# merged DataFrame `df_mvol`. It will hopefully become clear what has
# happened during the merge, i.e. information about monthly trading
# volume from `mvol` has been added to `df`. Observe that if there is
# a row in `mvol` with no match in `df`, then this row from `mvol`
# will NOT be included in the merged DataFrame `df_mvol`. For example,
# the rows from November and February 2014 are in `mvol`, but they are
# NOT added to `df_mvol` because there is no matching row in `df`.
df
mvol[['TICKER', 'mdate', 'MVOL']]
df_mvol

# Finally we can delete the `mdate` column as we don't need it any
# more.
del df_mvol['mdate']

# Here is our final result, with monthly trading volume added to each
# observation (i.e. to each row).
df_mvol
```

### G.6.2 Merging Stock Market and Accounting Data

In the next example we have stock market data from CRSP and accounting data from Compustat. The variables in Compustat are too numerous to list here, but we can give an explanation of the most important ones used in the example below:

1. tic: The ticker symbol of the company's stock.

2. datadate: The reporting date. In the example below we have quarterly data. Keep in mind that the data might not actually be released on the reporting date. For example, a balance sheet dated December 31 might be released in January or February due to the time required to prepare adjusting entries, write footnotes, and perhaps be audited or reviewed by a CPA firm. To be on the safe side, we wait for three months before using these numbers and merge them to the stock market data.

3. atq: A company's total assets.

The goal of the following code is to add the latest accounting data (released each quarter) to each trading day in CRSP. We illustrate three ways to do so. The first two ways use the PandaSQL package, while the third way uses the `merge_asof` function from Pandas. The PandaSQL package allows us to join different datasets based on date ranges. It works by converting the Pandas DataFrame to SQLite, running your SQL queries, and converting back to DataFrame. The code below can be found in the file `code-pandas-050-merging-data.py` on the course website and the data files we use are also on our course website.

```
# This Python script illustrates how to merge two datasets based on
# the nearest date, without requiring an exact match of the date. We
# show three ways to achieve this. The first two ways use the
# `pandasql` library to interface to SQL from Pandas, while the third
# way uses the `merge_asof` function from Pandas.
import pandas as pd
import numpy as np
from pandasql import sqldf
pysqldf = lambda q: sqldf(q, globals()) # Shortcut function.

# Import CRSP and convert the date strings to `datetime64`.
crsp = pd.read_csv('data-CRSP-extract.csv')
crsp['date'] = pd.to_datetime(crsp['date'])

# Same for Compustat.
cs = pd.read_csv('data-Compustat-extract.csv')
cs['datadate'] = pd.to_datetime(cs['datadate'])

# Next we define the time window during which the accounting
# information from Compustat is valid. First, we wait for three months
# before using any Compustat data. The reason is that in real life,
# the data is often released with a lag. For example, if a company's
# balance sheet is as of December 31, 2014, it is often released in
# January or February 2015. To be on the safe side, we therefore add
# three months to 'datadate'. This date corresponds to the BEGINNING
# of the time window we will use when merging.
cs['datadate'] = cs['datadate'] + np.timedelta64(121, 'D')
# Second, we let the accounting data expire after about one year. This
# date corresponds to the END of the time window we will use when
# merging.
cs['datadate2'] = cs['datadate'] + np.timedelta64(380, 'D')

# Extract a subset of the data for better illustration of the
# following merge. 'PRC' is the stock price and 'atq' is total assets.
crsp = crsp[['TICKER', 'date', 'PRC']]
cs = cs[['tic', 'datadate', 'datadate2', 'atq']]

# Inspect the data. Our GOAL in the remaining code is to add Compustat
# data to CRSP such that 'date' (from CRSP) lies in the time window
# given by 'datadate' and 'datadate2' (from Compustat).
crsp.tail()
cs.tail()

# Here we use SQL syntax to merge the two DataFrames using a left join
# (i.e. we add data to CRSP from Compustat because the 'crsp'
# DataFrame is mentioned to the "left" side of the 'cs' DataFrame in
# the 'FROM ...' part of the SQL query). The following merge could be
# done in a more memory-efficient way completely in SQL. But for
# simplicity we show this version first. 'SELECT *' means to select
# all columns from the resulting merged data. In the 'ON...' part, in
# order to tell SQL the DataFrame and column it should use, we add the
# DataFrame name in front of the column, separated by a dot. For
# example, 'crsp.TICKER' means that we are referring to the 'TICKER'
# column from the 'crsp' DataFrame.
df1 = \
    pysqldf(
        'SELECT * ' + \
        'FROM crsp LEFT JOIN cs ' + \
        'ON crsp.TICKER=cs.tic AND ' + \
        'crsp.date BETWEEN cs.datadate AND cs.datadate2')
# Clean up the result from the merge. PandaSQL converts 'datetime64'
# to 'str'. So we need to convert it back to 'datetime64' here.
df1['date'] = pd.to_datetime(df1['date'])
df1['datadate'] = pd.to_datetime(df1['datadate'])
df1['datadate2'] = pd.to_datetime(df1['datadate2'])
# Let's take a look at the resulting merge. The first thing you should
# notice is that 'date' is always between 'datadate' and
# 'datadate2'. So we were successful in adding data to CRSP based on
# the time window specified in Compustat through 'datadate' (the
# beginning of the time window) and `datadate2` (the end of the
# window).
df1.tail()
# There's still one problem we need to solve. The merge has added ALL
# rows from Compustat that fit the merging criteria, i.e. whose time
# window contains a `date` from CRSP. So for each observation
# originally in CRSP, we now have around FOUR observations. One for
# each quarter from Compustat whose one-year time window between
# `datadate` and `datadate2` matches a given `date` from CRSP. If you
# look closely at the `date` column, you will see that many dates are
# repeated.
df1.tail()
# You can look at these multiple matches from a different angle by
# counting how many matches from Compustat we have on average for each
# Ticker-date combination. For illustration, we look at the `size()`
# of each ticker-date group and save it in the column name
# `counts`. This gives us the number of observations we have for each
# ticker and each date. Most of these values turn out to be four as we
# typically have four observations in Compustat per year (i.e. four
# quarters). To further summarize this, we then calculate the mean of
# the `count` column for each ticker.
df1.\
    groupby(['TICKER', 'date']).\
    size().\
    to_frame(name='counts').\
    reset_index().\
    groupby('TICKER').\
    agg({'counts': 'mean'})
# To finally solve this problem of the multiple matches, we only keep
# the most recent match from Compustat and discard all the others. To
# do that, we first need to sort by `datadate` to ensure we are
# picking the latest information from Compustat. (Keep in mind that
# the following code works because `groupby()` preserves the order.)
# Then we grab the latest observation in each ticker-date group using
# `tail()` and thus discard all the earlier observations.
df1 = df1.sort_values('datadate').groupby(['TICKER', 'date']).tail(1)
df1 = df1.sort_values(by=['TICKER', 'date']).reset_index(drop=True) # Cosmetics.
df1.tail()  # Inspect the data. Now there is only ONE date per ticker.
del df1['datadate2']

# The approach above works, but it is a bit clumsy. First, it might
# require a lot of memory if your data size gets larger because the
# SQL query keeps a lot of data that we don't need in the end,
# e.g. all four quarters instead of the most recent quarter. Second,
# it would be more elegant if we can perform the whole merge directly
# in SQL without having to remove some data using Pandas at the
# end. We will show how to solve these two problems next.
# The following code gives the same result, but the whole merging is
# done in SQL and it is more memory-friendly in case your dataset is
# large. The reason is that SQL has some internal optimizations it can
# perform when executing the query. (In addition, you could also
# define `datadate2` "on the fly" in SQL, although we omit this part
# here for clarity.)
#
# Here it is important to keep in mind the order of execution of the
# SQL statements:
#
# 1) FROM ... LEFT JOIN ... ON ... (specifies which DataFrames to use
# and the kind of join to perform),
#
# 2) GROUP BY (specifies the grouping),
#
# 3) SELECT (specifies which columns to select; `*` means all columns
# and `MAX(cs.datadate)` takes the maximum value of `datadate` from
# the `cs` DataFrame in each group and discards all other observations
# in the group).
df2 = \
    pysqldf(
        'SELECT␣*,␣MAX(cs.datadate)␣' + \
        'FROM␣crsp␣LEFT␣JOIN␣cs␣' + \
        'ON␣crsp.TICKER=cs.tic␣AND␣' + \
        'crsp.date␣BETWEEN␣cs.datadate␣AND␣cs.datadate2␣' + \
        'GROUP␣BY␣crsp.TICKER,␣crsp.date')
df2['date'] = pd.to_datetime(df2['date'])
df2['datadate'] = pd.to_datetime(df2['datadate'])
df2.drop(['MAX(cs.datadate)', 'datadate2'], axis=1, inplace=True) # Remove columns.
df2 = df2.sort_values(by=['TICKER', 'date']).reset_index(drop=True) # Cosmetics.

# Inspect the data. It should be same as with `df1`.
df2.tail()

# Check if both DataFrames are the same. They should be the same as
# both ways of merging should yield the same results.
df2.equals(df1)

# Last but not least, we can also merge both datasets directly in
# Pandas using the `merge_asof()` function of Pandas. The merging
# logic is similar to the ones we used before, however we omit using
# `datadate2` as the expiration date and instead use the `tolerance`
# argument.
crsp = crsp.sort_values(by=['date'])
cs = cs.sort_values(by=['datadate'])
df3 = \
    pd.merge_asof(
        crsp, cs,
        left_by='TICKER', right_by='tic',
        left_on='date', right_on='datadate',
        tolerance=pd.Timedelta('380 days')).\
    sort_values(by=['TICKER', 'date']).\
    drop(['datadate2'], axis=1).\
    reset_index(drop=True)
# Check again if the result is the same as the previous one.
df3.equals(df1)
```

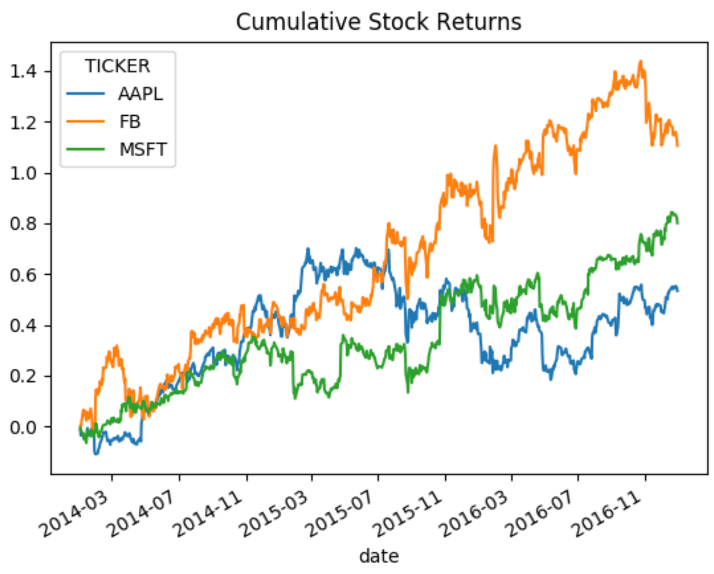
## G.7 Plotting with Pandas

The nice thing about Pandas is that you can often create plots very easily, without using many add-on packages (besides the standard Matplotlib). What Pandas does is to wrap Matplotlib so that it is straightforward to plot the data you have in a DataFrame. We have briefly discussed Matplotlib in Section H.2 starting on page 278. Here, however, we focus more specifically on plotting with the Pandas wrapper around Matplotlib.

In the following code we take the data from CRSP, calculate cumulative stock returns, and plot them. The code below can be found in the file `code-pandas-060-plotting.py` on the course website and the data files we use are also on our course website.

```
# This script calculated cumulative stock returns and plots them.

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Here we import the data and select the columns we need.
df = pd.read_csv('data-CRSP-extract.csv')
df['date'] = pd.to_datetime(df['date']) # Convert string to ‘datetime64‘.
df = df[['TICKER', 'date', 'RET']] # Just ticker, date, and stock returns.

# Calculate the cumulative stock returns. They show by how much each
# stock price has gone up or down during our sample's time period.
df.RET = np.log(1 + df.RET) # Convert to log-returns so that we can sum them up.
df = df.sort_values(['TICKER', 'date']) # Ensure data is sorted before running ‘cum
df['cum_RET'] = \
    df.\
    groupby('TICKER')['RET'].\
    apply(lambda x: x.cumsum()) # For each ticker, calculate cumulative sum.
df.cum_RET = np.exp(df.cum_RET) - 1 # Convert log returns to linear returns.
del df['RET'] # Don't need raw returns any more for this application.

# Reshape the data to have each ticker in a separate column. The
# ‘pivot()‘ method can save a lot of work if you compare it to
# reshaping the data yourself using for-loops! Keep in mind that here
# we specify `date` as the index, which is useful for the subsequent
# plotting commands, as the date will automatically be put on the
# x-axis.
df = df.pivot(index='date', columns='TICKER', values='cum_RET')

# Plot the cumulative stock returns for all stocks (i.e. in this case,
# for all columns).
df.plot()
plt.title('Cumulative␣Stock␣Returns')
plt.show()
```

The code above should create the following plot, showing the cumulative returns of the stocks in our partial CRSP data extract. In case the lines are not easy to see in the printout, please take a look at the electronic version on the course website.



## G.8 Downloading Equities Data from AlphaVantage Into Pandas

AlphaVantage is a data provider for financial market data. They provide the following data for free. All you need is to get a free API key from their websites.

1. Realtime and historical data in JSON and CSV format

2. Technical indicators

3. Bitcoin and other digital currencies

A such, it is a great alternative to Yahoo Finance or Google Finance, which had some problems recently. There are of course also other data providers out there who provide at least partially free data, for example

1. Yahoo Finance
2. Google Finance
3. AlphaVantage
4. Stooq
5. Quandl
6. BarChart
7. IEX

There are (at least) two ways to access AlphaVantage:

1. Since AlphaVantage provides the data as CSV (as well as JSON), you can directly import the relevant CSV files into Pandas. We will show such an example at the beginning of the following script.
2. Specific Python packages, such as `alpha_vantage`. Packages like these are usually just a thin wrapper around the AlphaVantage API. Their goal is to make it easier to get the data you want quickly. You can also directly specify the format you would like to data to receive in, e.g. JSON, CSV, or a Pandas DataFrame. As DataFrames are so practical, we will mostly focus on this data format.

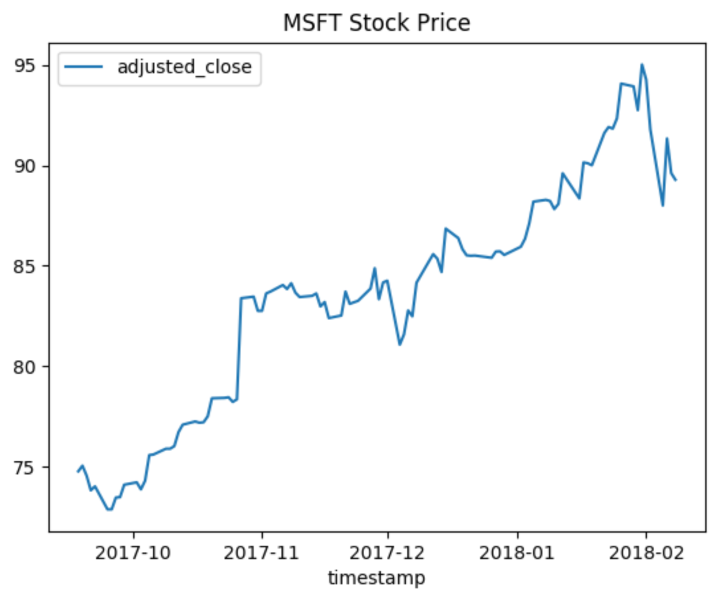
The code below can be found in the file `code-pandas-070-alphavantage.py` on the course website.

```
# This script illustrates downloading financial data from
# AlphaVantage.

# Your AlphaVantage key. In order to download the data, you can get a
# key directly from the AlphaVantage website for free. If you would
# like to save your API key in a text file (e.g. for several of your
# scripts to use), you can read it as follows:
# exec(open('../code-API-key-alphavantage.py').read())
mykey = 'demo'

# Specify the stock ticker here. You can also use stock indices,
# e.g. IXIC (Nasdaq Composite), DJI (Dow Jones Industrial Average),
# INX (S&P 500) etc. You can also get data from stock markets around
# the world. For example, 'NSE:TITAN' gets the stock value of TITAN
# from the Indian Exchange NSE, or 'AI.PA' is for Air Liquide at the
# Paris stock exchange (PA).
myticker = 'MSFT'
```

```
import matplotlib.pyplot as plt
import pandas as pd

# Here we manually download some data from AlphaVantage, using only
# the Pandas package.
df = \
    pd.read_csv(
        'https://www.alphavantage.co/query?' + \
        'function=TIME_SERIES_DAILY_ADJUSTED&' +\
        'symbol=' + myticker + \
        '&apikey=' + mykey + \
        '&datatype=csv')
# Take a look at the data we just downloaded. Keep in mind that
# AlphaVantage gives you the raw data sorted DESCENDINGLY based on the
# timestamp.
df.head()
df.sort_values(by='timestamp', inplace=True) # Sort by `timestamp`.
df.reset_index(drop=True, inplace=True) # Reset the index to 0,1,2,3...
df.timestamp = pd.to_datetime(df.timestamp).dt.date # Convert to datetime and discar
# Plot the stock price, adjusted for dividends and stock splits.
df.plot(x='timestamp', y='adjusted_close')
plt.title(myticker + '␣Stock␣Price')
plt.show()

# Calculate stock return.
df['L_close'] = df.adjusted_close.shift() # Lag the closing price.
df['RET'] = df.adjusted_close / df.L_close - 1 # Calculate stock index return.
df.dropna(inplace=True)        # Drop missing data.
df.reset_index(drop=True, inplace=True) # Reset the index to 0,1,2,3...

# Cut returns into low (0), medium (1), and high (2) returns based on
# quantiles.
df['q_RET'] = pd.qcut(df.RET, 3, labels=False)
df[df.q_RET == 2].head()        # Show examples of high returns.
df[df.q_RET == 2].RET.mean()    # Mean return of high quantile.
```

```
from pprint import pprint       # For pretty printing.
# Here we import a bunch of things from the `alpha_vantage`
# package. Depending on what you want to download, you just need some
# of these imports.
from alpha_vantage.timeseries import TimeSeries
from alpha_vantage.techindicators import TechIndicators
from alpha_vantage.sectorperformance import SectorPerformances
from alpha_vantage.cryptocurrencies import CryptoCurrencies
from alpha_vantage.foreignexchange import ForeignExchange


# You can get the data in JSON format if you like. However, most of
# the time it is better to ask for a Pandas DataFrame as we do in the
# remaining examples.
ts = TimeSeries(key=mykey)       # In JSON by default.
data, meta_data = ts.get_daily_adjusted(symbol=myticker)
pprint(data)

# Here we request a Pandas DataFrame, which often is more convenient
# to work with.
ts = TimeSeries(key=mykey, output_format='pandas')
# Get object with data and another with the call's metadata. By
# default, it just returns a subset of the data. If you want the whole
# dataset, use ‘outputsize='full'‘ as an additional function argument.
data, meta_data = ts.get_daily_adjusted(symbol=myticker)
# Plot the data.
data['5.␣adjusted␣close'].plot()
plt.title('Times␣Series␣for␣' + myticker)
plt.show()

# Example of technical analysis: Bollinger Bands.
ti = TechIndicators(key=mykey, output_format='pandas')
data, meta_data = ti.get_bbands(symbol=myticker, time_period=20)
data.tail(252).plot()            # Plot last year only.
plt.title('BBbands␣indicator␣for␣' + myticker)
plt.show()

# Sector performance.
sp = SectorPerformances(key=mykey, output_format='pandas')
data, meta_data = sp.get_sector()
data['Rank␣G:␣Year␣Performance'].plot(kind='bar')
plt.title('Sector␣Performance␣(Year)')
plt.tight_layout()
plt.grid()
plt.show()

# Crypto currencies.
cc = CryptoCurrencies(key=mykey, output_format='pandas')
data, meta_data = \
    cc.\
    get_digital_currency_daily(
        symbol='BTC',
        market='CNY')
data['4b.␣close␣(USD)'].plot(logy=True)
plt.tight_layout()
plt.title('Daily Value for Bitcoin (BTC)')
plt.grid()
plt.show()

# Foreign exchange data is only available as JSON format (not in CSV
# or Pandas format). There's also no metadata in this call, so we just
# use a placeholder '_'.
cc = ForeignExchange(key=mykey)
data, _ = \
    cc.\
    get_currency_exchange_rate(
        from_currency='EUR',
        to_currency='USD')
pprint(data)
```

The first part of the previous code prints the stock price from AlphaVantage similar to the following plot.



# Appendix H

# Summarizing, Displaying, and Visualizing Your Results

Once you have finished your analysis of the textual data, it is important to present it in a way that other people can understand what you are doing and act on it. In other words, you need to summarize, display, and visualize your results. Of course, it is often an **iterative process** where you go back and forth between analyzing, summarizing, and presenting your work.

It would be possible to have a whole course just on these topics, so we will necessarily have to be brief here. However, it is still possible to point you into the right direction, which is often the most important part of this journey. We break down this big topic into three different parts:

- Summarizing your data in Section H.1
- Plotting your results in Section H.2
- Creating web apps in Section H.3

Importantly, in this section we are going to focus on the top packages only. If you want to get an **overview of related packages**, see Appendix E starting on page 231 for Python-related packages.

## H.1 Summarizing Your Data

Once you have obtained your data from the textual analysis steps, you often want to summarize it. Another thing you often need to do is to **merge it with other data, e.g. stock market data or accounting/fundamental data**. For example, you are testing whether there are some users on Twitter whose textual output consistently predicts market returns in the next day or week, you need to add further data about stock or stock index returns and ultimately you need to bring all that data together.

In the Python world, the pandas package rules (for more details **see Appendix G starting on page 255**). Originally pandas was written by Wes McKinney while he was working at the hedge fund AQR. He was trying to address some of the frustrations people had when using R (although my guess is that these people were not using data.table in R at that time, otherwise they might not have been so frustrated). In any case, **pandas was the big bang event for data science in Python** and it is fair to say that it was a total game-changer. Before pandas, people were mostly using Python in areas other than data science. Pandas changed all that, and it made great strides because a lot of infrastructure in Python (mainly for scientific computing) had already been built with the NumPy and SciPy packages. Pandas took a lot of inspiration from R. For example, R has data.frames while pandas has DataFrames. The pandas syntax is also relatively similar to working in R with data.frames. In fact, switching between pandas and R is relatively easy because many of the concepts are the same or similar.

If your data is larger than memory, but not yet huge as in big data, Python is in the sweet spot with dask. The dask package builds upon pandas and allows it to split up the data and process it in smaller chunk. It thus avoids running out of memory. Importantly, it achieves this largely transparently, so you don't have to worry about how it actually splits up the data. Furthermore, it tries to stay consistent with the pandas syntax, so if you know how to use pandas, you also know how to use dask automatically.

## H.2 Plotting

This course is not mainly focused on plotting. However, sometimes it is necessary to create a graphical representation of your data. Below are pointers to some of the popular plotting packages in Python.

Python has various plotting packages available.

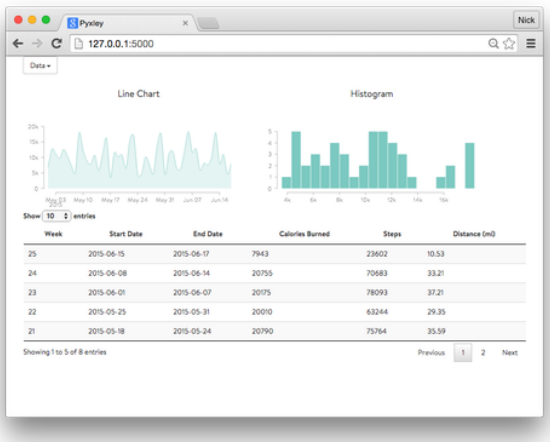
1. For an overview of the most popular visualization packages in Python, please take a look at Appendix E starting on page 231.
2. For plotting with the Pandas package (which wraps Matplotlib for plotting), see Section G.7 starting on page 270.
3. In the current section we provide a brief introduction to Matplotlib.

Matplotlib historically often has been a good place to start, although there are many other plotting packages available by now. For example, to plot a histogram using Matplotlib, you can simply type:

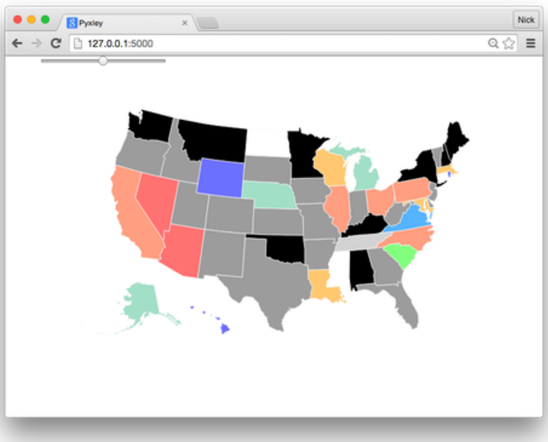
```
import matplotlib.pyplot as plt
import numpy as np
plt.hist(np.random.randn(1000))
plt.show()
```

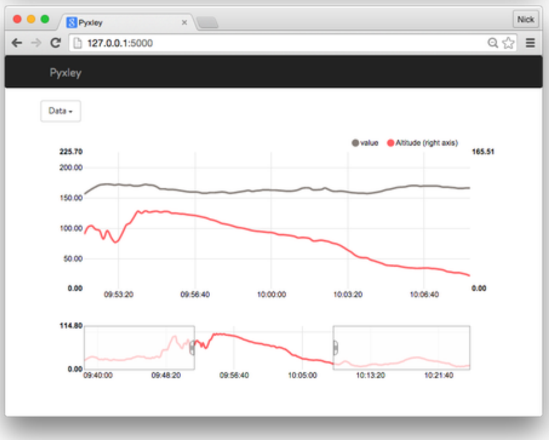
## H.3 Creating Web Apps

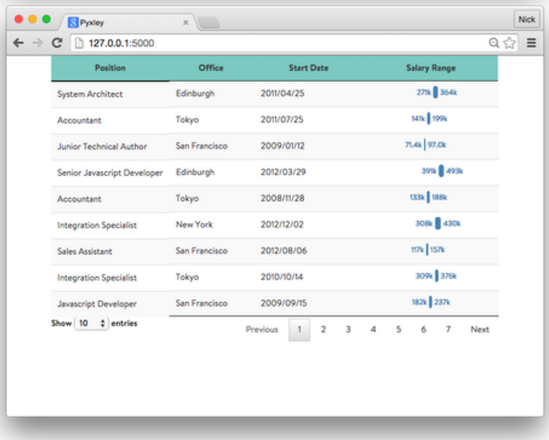
Often it makes sense to create a dashboard, which is a web-based analytics app, where you can showcase your work. In Python there are many options available, the most popular ones being Dash and Pyxley, although there are many others as discussed in Section E.9 starting on page 236. A dashboard can be very useful, e.g. if you have a client meeting and would like to demonstrate and visualize the functionality of your code. People can look at charts and diagrams and interact with them in real-time. A few example dashboards can be seen below:



| Week | Start Date | End Date | Calories Burned | Steps | Distance (mi) |
|---|---|---|---|---|---|
| 25 | 2015-06-15 | 2015-06-17 | 7943 | 23602 | 10.53 |
| 24 | 2015-06-08 | 2015-06-14 | 20755 | 70683 | 33.21 |
| 23 | 2015-06-01 | 2015-06-07 | 20175 | 78093 | 37.21 |
| 22 | 2015-05-25 | 2015-05-31 | 20010 | 63244 | 29.35 |
| 21 | 2015-05-18 | 2015-05-24 | 20790 | 75764 | 35.59 |







| Position | Office | Start Date | Salary Range |
|---|---|---|---|
| System Architect | Edinburgh | 2011/04/25 | 271k 364k |
| Accountant | Tokyo | 2011/07/25 | 141k 199k |
| Junior Technical Author | San Francisco | 2009/01/12 | 71.4k 97.0k |
| Senior Javascript Developer | Edinburgh | 2012/03/29 | 391k 493k |
| Accountant | Tokyo | 2008/11/28 | 133k 188k |
| Integration Specialist | New York | 2012/12/02 | 308k 430k |
| Sales Assistant | San Francisco | 2012/08/06 | 117k 157k |
| Integration Specialist | Tokyo | 2010/10/14 | 309k 376k |
| Javascript Developer | San Francisco | 2009/09/15 | 182k 237k |

# Appendix I

# Common Terms and Abbreviations

When reading this book you may come across a term or abbreviation whose meaning is not fully clear. In this case, you can look it up here:

- Semantics: The intrinsic meaning of natural language.
- Valence: Usually it refers to whether a word has positive or negative meaning.
- Term: It basically means "word," although it sometimes might refer to a more abstract word representation (e.g. a linear combination of words or an n-gram), see for example Section 7.3.
- Tokenization: Turning a string or document into tokens, i.e. smaller parts. For example, these could be words, word combinations (e.g. n-grams), sentences, or paragraphs, see Sections 13.1 and 20.5.2 starting on pages 161 and 210, respectively.
- Corpus: A large and structured set of texts, typically a set of text documents. You can think of a text document as a book and the corpus as a library containing many books. Note that the plural of corpus is "corpora."
- Bag-of-words (BoW): Representing a text by the word counts, see Section 14.2 and Chapter 15.
- Stop word: A word that occurs frequently in text but does not have significant meaning such as "the," "is," "at," "which," or "on." Often these words are removed when preprocessing text. See also Section 13.2 starting on page 163.
- Render: It basically means to "display" something, e.g. a browser renders a HTML document.
- Parsing: Analyzing a string of symbols conforming to the rules of a formal grammar. For example, you could parse a HTML document to find all links, see Section F.3.2 starting on page 245.
- API: Short for "application programming interface." It means that you can write a programm to access the application, e.g. to download some data. An example is the

Twitter API that lets you download tweets programmatically instead of scraping the Twitter website, see Section F.4 starting on page 248.