From Zero to Senior Data Science

A Guide into
Advanced Data Science Standards
and Big Data Approaches

By Andreas Traut

From Zero to Senior Data Science

A Guide into Advanced Data Science Standards and Big Data Approaches

Andreas Traut

1. Edition 2021

Author: Andreas Traut

E-Mail: andreas.traut@freedommail.ch

GitHub: https://github.com/AndreasTraut

Homepage: https://andreas-traut.gitbook.io/

Contents

1 Introduction

1.1 Aim

The goal of this book is to guide you to becoming a Senior Data Scientist. My aim is to help you to get started in the Data Science topics as quickly as possible and explain the necessary Data Science knowledge. It is important to me to give you many practical examples to download and try out yourself, as well as to help you to install some very important Data Science tools (like Jupyter-Notebook, Docker, Spyder,…) on your own computer. This book is, as the title suggests, a practical guide in advanced data science standards and big data approaches. So far, this book does not cover any mathematical or algorithmic specifics. I'm saving that for a later edition of this book and will be happy to come back to it when I'm overwhelmed with positive requests.

I am not afraid to refer to good tutorials or trainings, which are necessary from my point of view to become a Senior Data Scientist. For me it doesn't make sense to re-document here every aspect that is already described much better somewhere else. My aim is more to connect many of these great sources here in in my book. I want you to get started and my motivation is to enable you to continue working independently yourself with these references to further documentation. Please also have a look at my footnotes: there I have collected many useful references. As soon as you have understood the topics from my book, it will be time to deal with the subject-specific documents anyway.

I believe in the Open-Source[1] spirit of sharing knowledge to other people. Therefore this book is subject to the CC BY-NC-SA 4.0 Licence (see sec. 1.5) and is available free of charge.

I am also willing to improve this book in the future and spend as much time and effort as I can to motivate and educate many people to become a Senior Data Scientist. Don't hesitate to contact me in case of suggestions for improving this book. You find my contact details after the title page of this book. I hope you enjoy reading and experimenting.

[1] Open-source model, Wikipedia, https://en.wikipedia.org/wiki/Open-source_model

1.2 Who is a "Data Scientist"?

1.2.1 Term "Data Scientist" is not Protected

Unfortunately, the term "Data Scientist" is not a protected term in Germany[2]! Anyone can call themselves a "Data Scientist" without any training or certificate or any kind of proof! This amazes and shocks me, because Data Science is a very promising field for the future[3]. Even HR managers struggle in correctly evaluating the meaning and requirements of "Data Science". What each individual understands by "Data Science" is as elastic as chewing gum. Nevertheless, it is used as a job title in many job descriptions, often with completely different interpretations.

For example: one HR manager writes "Data Scientist" in a job profile, but thinks that in-depth knowledge of the company and its manufacturing processes is urgently needed, and sorts out applicants according to this criterion. If you haven't worked in that specific area of the business before, you're out. This person denies the fact, that programming skills with machine learning libraries is also very essential in order to be a "Data Scientist". Also, beware: there are many good programmers who have no knowledge of machine learning libraries. This is of little use for a Data Scientist department: machine learning libraries, such as Scikit-Learn and Apache Spark Machine Learning Library are quite specialized in operation and even good programmers will need some time to get familiar with these concepts. I will explain these concepts in later chapters. Machine learning programming is different!

I have seen "Data Science" job descriptions where the activities were: adding google analytics and other tracking tools to a company site for a consistent user tracking. That's nothing more than search engine optimization[4], and it's actually a classic marketing job that's been elevated by the usual expectations of an increasingly digitized world. Of course, the marketing department is also being digitized, like so many other things. But it doesn't have much in common with Data Science.

People who describe themselves as a "Data Scientist," do not have to worry about any legal or employment lawsuits. If, on the other hand, someone describes himself as a "business economist" and cannot back this up with certificates or diplomas, then he risks to be threatened with lawsuits. For a profession of the future, such as "Data Scientists" are, this is a bitter realization.

1.2.2 Generally Accepted Definition of "Data Science"

Is there a generally accepted definition for "Data Science / Data Scientist"? Yes! There is a much discussed and generally quite accepted definition: the "**Data Science Venn Diagram from Drew Con-**

[2] Geschützte Berufsbezeichnungen, Wikipedia, https://de.wikipedia.org/wiki/Berufsbezeichnung#Gesch%C3%BCtzte_Berufsbezeichnungen

[3] Bayerische-Staatszeitung vom 07.09.2018, "Massenweise Daten, kaum Analytiker" https://www.bayerische-staatszeitung.de/staatszeitung/beruf-karriere/detailansicht-beruf-karriere/artikel/massenweise-daten-kaum-analytiker.html#topPosition

[4] Search Engine Optimization, Wikipedia, https://en.wikipedia.org/wiki/Search_engine_optimization

way"[5]. If you are not yet familiar with this "Data Science Venn Diagram", I would strongly recommend that you look into it! The misconceptions of what a "Data Scientist" is and what they can do are unfortunately still huge.

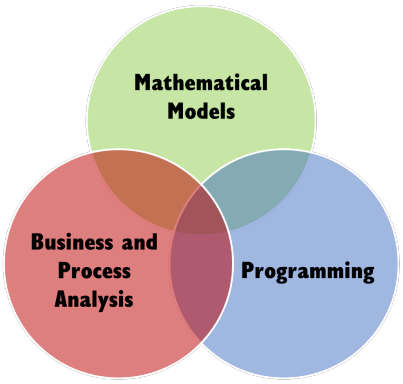


Figure 1.1: Data Science Venn Diagram - Adapted version. Creative Commons Legal Code

For me, a Data Scientist is a person who has 30% mathematical knowledge, has 30% of understanding the business and processes (which requires some years of working experience, see sec. 1.2.3) and 30% of programming experience with machine-learning libraries and she (or he) needs 10% passion for new exciting topics and willingness to dive into new tools.

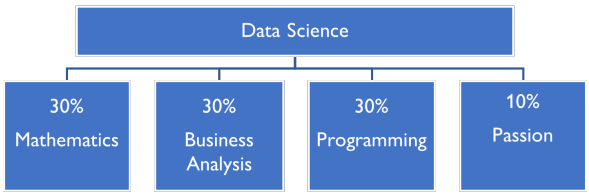


Figure 1.2: Data Science

So "Data Science" is a mix of mathematical knowledge, business knowledge and machine-learning programming experience. The balanced intersection of these three topics makes a Data Scientist. Adding the passion and motivation to learn each day makes a *good* Data Scientist, because the new technologies and big data approaches are evolving every minute. I will show you in sec. 1.4 what "passion" means to me.

It makes no sense to me to overvalue one of these areas and ignore one of the others! For example: someone who has profound in-depth knowledge of a company and its manufacturing processes but who lacks mathematical skills or programming skills with machine-learning libraries is in my opinion not a "Data Scientist". You always need all parts – mathematics, programming and business knowledge – to be a Data Scientist!

[5] Data Science Venn Diagram from Drew Conway, http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram

1.2.3 Minimum Required Years of Working Experience for a "(Senior) Data Scientist"

How long should you have worked in order to name yourself a "Senior Data Scientist"? I think that you should have at least 1-2 years of working experience before you can name yourself a "Data Scientist" and you should have at least 2-3 years of working experience before you can name yourself a "Senior Data Scientist". I would substantiate this assertion as follows: the **"Certification Manual for Data Science"**[6] from"*Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme IAIS*"[7] explains their requirements for certifying someone as a"Data Scientist". In order be become a"Data Scientist Basic Level" you need

- completed studies at a university (or equivalent)
- at least 1 year of work experience in a data science relevant field.
- the Certificate "Data Scientist Basic Level", which you can earn after passing a 3 hours written exam at the Fraunhofer-Institute

In order to become a "Senior Data Scientist" you need

- completed studies at a university (or equivalent)
- at least 2 years of work experience with at least 1 year in a data science relevant field
- the Certificate "Data Scientist Basic Level", which you can earn after passing a 3 hours written exam at the Fraunhofer-Institute
- a Specialized Certificate (several options like "Big Data" or "Deep Learning" and others are available), which you can earn with a 3 hours written exam at the Fraunhofer-Institute
- a 40 paged expose (similar to a bachelor thesis) approved by Fraunhofer and related to a data science topic

Besides this I found various articles among them also one of the Online-Magazine "Towards Data Science"[8]. It says, that

- after 2 years working experience have skills on Python, Jupyter and Kaggle, which is required to name yourself a "Junior Data Scientist" and
- after 3-5 years you have skills on Docker, Cloud and APIs, which are required to name yourself a "Senior Data Scientist".

This excludes any freshly graduated student from an university to name himself a "Data Scientist" or even a "Senior Data Scientist"! Underestimating work experience is one common issue, that also HR managers face, when they evaluate a "Data Scientist" job application. You need in-depth experience to understand the daily processes that take place in a company. I am convinced that you can't get

[6] Fraunhofer Zertifizierungshandbuch "Data Science", https://www.personenzertifizierung.fraunhofer.de/content/dam/zertifizierung/de/documents/Zertifizierungshandbuch_Data_Scientist.pdf

[7] Fraunhofer-Insitut für Intelligente Analyse- und Informationssysteme IAIS, https://www.iais.fraunhofer.de/de/data-scientist-schulungen.html

[8] "Towards Data Science", https://towardsdatascience.com/becoming-a-level-3-0-data-scientist-52641ff73cb3

that from a university lecture. To be able to apply reporting and control systems correctly requires a lot of practical experience. I was an internal and external auditor for several years and I know many peculiarities, strengths and also weaknesses that can be hidden in such reporting and control systems. I am convinced, that several years of professional experience are necessary to internalize corporate procedures and without these experiences you won't be a Data Scientist.

I hope that one day in the future the term "Senior Data Scientist" will be protected a bit better with certificates or other proofs of qualification. At the moment, unfortunately, "Data Scientist" and "Senior Data Scientist" is a very elastic and stretchable term, which everyone likes to interpret for himself.

1.3 Structure of this Book

1.3.1 Installation of Data Science Tools

In this chapter (see sec. 2) I will explain how to install the basic Data Science Tools, which you need. In my examples I use Jupyter-Notebooks[9], which is a widespread standard today, but I also use an Integrated Development Environment[10]. In my opinion Jupyter-Notebooks are good for the first examinations of data and for documenting procedures and up to a certain degree also for sophisticated data science. But it is a good idea to learn very early how to work with an IDE. In my opinion Jupyter Notebooks are not always the best environment for learning to code! Therefore I will give you a short introduction into an IDE and highly recommend that you learn how to work with an IDE.

1.3.2 Visualization of Different Datasets

For a Data Scientist the visualization is as important as the analysis itself. I worked on different datasets with the aim to visualize the data and in the **first part** of this chapter (see sec. 3.1) I will explain these examples. I used python and libraries like e.g Matplotlib[11] or Seaborn[12], which are available for free. I will show you in Section sec. 2 how to install these tools on your own computer.

Each of the datasets, which I worked on, contains different topics of necessary preliminary work before I could visualize them, e.g. converting dates or numbers, adding/extracting information and so on. I will show you how this can be done.

In the **second part** (see sec. 3.2) I will explain some Big Data visualization problems and techniques to solves these.

[9] Jupyter-Notebook, https://jupyter.org/

[10] Integrated development environment, Wikipedia, https://en.wikipedia.org/wiki/Integrated_development_environment

[11] Matplotlib, https://matplotlib.org/

[12] Seaborn, https://seaborn.pydata.org/

In the **third part** (see sec. 3.3) I will show you how to visualize and share the data with a "data app". Data Scientists often forget, that all models and visualizations, which they have built, need to be used by someone, who is probably not as skilled as themselves in all these technical requirements. Such "data apps" are helpful to make the data accessible very quickly for everyone on all devices (also mobile phones).

In the **fourth part** sec. 3.4 I will list some common professional tools, which offer visualization functionality and more. These tools cost some money, but I recommend to have look into these: many companies use these or similar tools.

1.3.3 Machine Learning with Python: Comparison Small Data versus Big Data

After having learnt visualization techniques in Python it is time to start working on different datasets with the aim to learn and apply machine learning algorithms. In chapter sec. 4 I am particularly interested in explaining the differences and similarities of "Small Data" (Scikit-Learn) approaches versus the "Big Data" (Apache Spark) approaches. Therefore in this chapter I will focus on this "comparison" question of "Small Data" coding versus "Big Data" coding. I haven't seen many comparisons of "Small Data" vs "Big Data" (neither theoretically nor in coding patterns) and I think understanding this is interesting and important.

The **first example** (see sec. 4.1) is about a Movies database and the revenues, which these movies generated. My aim is to predict the revenues. I use a Jupyter-Notebook and will explain how to apply the standard procedures (e.g. data-cleaning & preparing, model-training,…).

The **second example** (see sec. 4.3) is for being used in an IDE (Integrated Developer Environment), like the Spyder-IDE[13] from the Anaconda distribution[14] and applies the *Scikit-Learn Python Machine Learning Library*[15] (you may call this example a "Small Data" example if you want). I will show you a typical structure for a machine-learning example and put it into a mind-map. The same structure will be applied on the third example.

The **third example** (see sec. 4.4) is a "Big Data" example and will use a Docker environment[16] and apply the *Apache Machine Learning Library*[17], a scalable machine learning library. The mind-map from the second part will be extended and aligned to the second example.

[13] Spyder-IDE, https://www.spyder-ide.org/
[14] Anaconda distribution, https://www.anaconda.com/
[15] Scikit-Learn, https://scikit-learn.org/
[16] Docker environment, https://www.docker.com/
[17] Apache Machine Learning Library, https://spark.apache.org/mllib/

1.3.4 Big Data: Map-Reduce and K-Means Clustering

Big Data Approaches are very different and challenging. Someone who never heart about Big Data Approaches will be surprised about how many details and topics are very different to the standard data science approaches (like Jupyter-Notebooks on Scikit-Learn). Therefore I dedicate a separate chapter for this topic: in chapter sec. 5) I explain the Map-Reduce programming model on a Word-Count example and I show how the K-Means Clustering Algorithm in Apache Spark ML works.

1.3.5 Use Cases of Artificial Intelligence in the Industry

In the chapter sec. 6 I explain in an easily understandable way what is meant by ”artificial intelligence (AI) in the industry”, describe divisons of application and illustrate with practical examples and programming applications how AI can be implemented in concrete terms.

In the **first part**, I will explain the basic concepts of AI and describe some domains where AI is already being successfully applied. I will also describe the peculiarities of big data, deep learning and process mining.

In the **second part,** I show an example of how the car manufacturer BMW has benefited from AI techniques.

In the **third part,** I show how AI techniques can be implemented in the Python programming language when dealing with the topic of image recognition and provide my programming code in the process.

In the **fourth part,** I give some recommendations on what to look for when introducing AI techniques in a company.

I think the choice of further articles to delve into the topic ”Use cases of artificial intelligence in industry” is huge and I hope this short introduction is helpful to get started.

1.4 My Qualification

I am a graduated *Diplom-Mathematician (University of Freiburg)* and a *Certified Advanced Data Scientist (Fraunhofer Institute)*. I am a *Certified Data Scientist Basic Level* and a *Certified Data Scientist Specialized in Big Data Analytics*. Additionally I passed a *French Bachelor Degree in Mathematics (University of Besançon, France)* during my Erasmus year in France.

I have passion for being a Data Scientist and I wrote in sec. 1.2.2 how important “passion” is for being a *good* Data Scientist. Therefore I participated in two online courses and passed the exams. There are many online courses available and choosing the right one for your needs and skills is one task. Completing the course the other! You need passion to do this.

I am holding the *Certificate "Data Analysis with Python: Zero to Pandas"*[18] (see fig. 1.3) which covers topics like data visualization and exploratory data analysis on the basis of Python[19], Numpy[20], Pandas[21], Matplotlib[22] and Seaborn[23]. I can recommend this course and I wish I would have found this course before I wrote the chapter sec. 3, because it was very helpful.

I am holding the *Certificate "Data Structures and Algorithms in Python*[24] (see fig. 1.4) which covers important data structures and algorithms like binary trees[25], recursion[26], directed graphs[27] and the knapsack problem[28]. I think knowing about these topics is very important, because a high performance cluster can solve your problem only sometimes: often a good data structure and algorithm will be a lot more powerful than any hardware solution.

Figure 1.3: Certificate Data Analysis with Python: Zero to Pandas

[18] Certificate "Data Analysis with Python: Zero to Pandas", Jovian Data Science Platform, https://jovian.ai/certificate/MFQTGOBQGM

[19] Python, https://www.python.org/

[20] Numpy, https://numpy.org/

[21] Pandas, https://pandas.pydata.org/

[22] Matplotlib, https://matplotlib.org/

[23] Seaborn, https://seaborn.pydata.org/

[24] Certificate "Data Strucutres and Algorithms in Python", Jovian Data Science Platform, https://jovian.ai/certificate/MFQTINRVG4

[25] Binary Tree, Wikipedia, https://en.wikipedia.org/wiki/Binary_tree

[26] Recursion, Wikipedia, https://en.wikipedia.org/wiki/Recursion

[27] Directed Graph, Wikipedia, https://en.wikipedia.org/wiki/Directed_graph

[28] Knapsack Problem, Wikipedia, https://en.wikipedia.org/wiki/Knapsack_problem

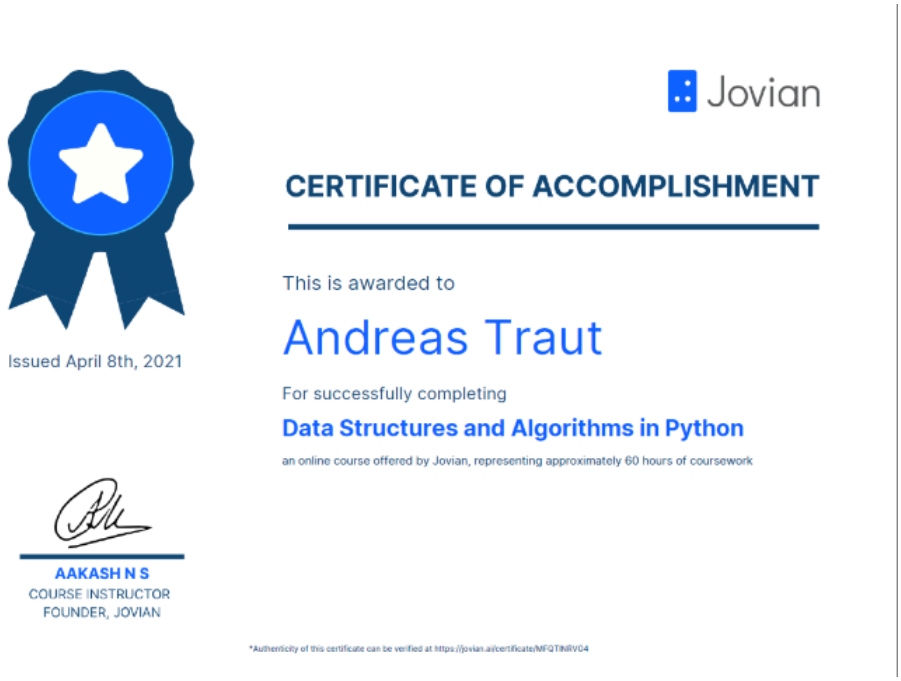


Figure 1.4: Certificate Data Structures and Algorithms in Python

1.5 Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Figure 1.5: Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

2 Installation of Data Science Tools

2.1 Jupyter-Notebooks and Spyder-IDE

I use Jupyter-Notebooks[1], which is a widespread standard today, but I also use the Spyder-IDE[2]. The IDE stands for Integrated Development Environments[3]. I think you will need both of them, as I will explain in the following chapters.

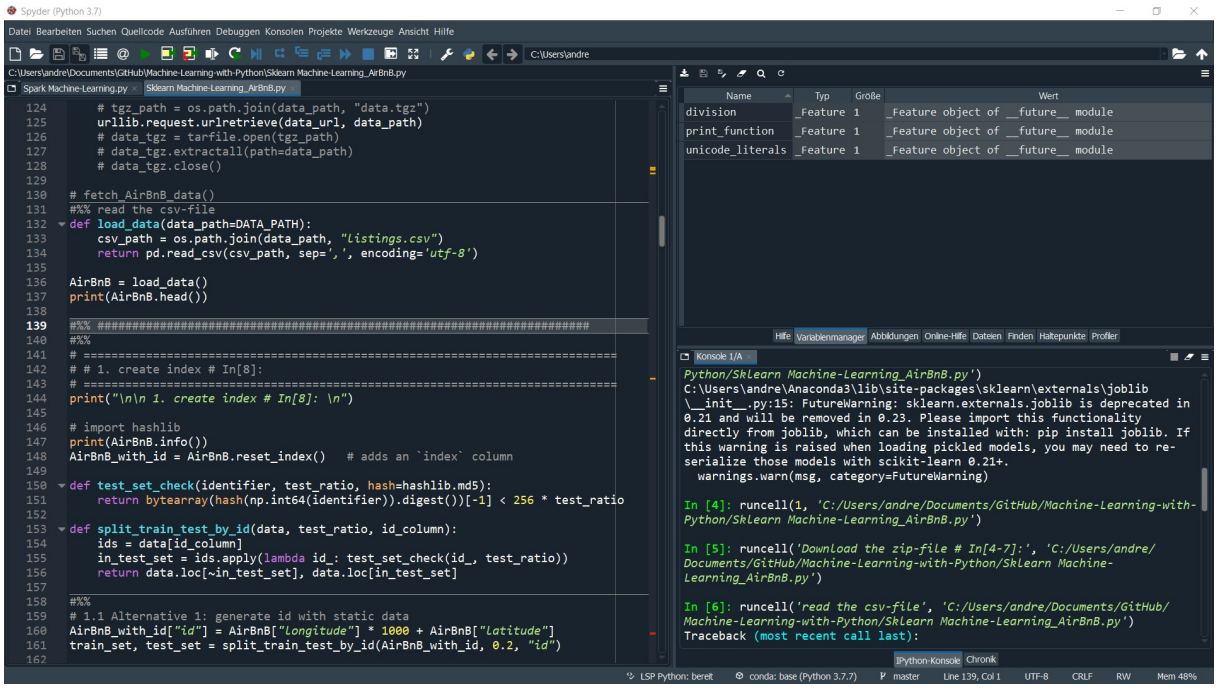


Figure 2.1: Spyder-IDE: An integrated development environment with debugger.

The Spyder-IDE is a separate software application, where you can debug your code, which makes the development of algorithms easier - a big advantage! The disadvantage is, that you need to learn to use this software application first, which is a bit more difficult than a Jupyter-Notebook but really worth the time you spent. It looks like shown in figure fig. 2.1: on the left side you can write your code. On

[1] Jupyter-Notebook, https://jupyter.org/

[2] Spyder-IDE, https://www.spyder-ide.org/

[3] Integrated development environment, Wikipedia, https://en.wikipedia.org/wiki/Integrated_development_environment

the right side on top there are different views available: the variable manager, plot view and help view are the most important to mention. On the right side below there is the console.

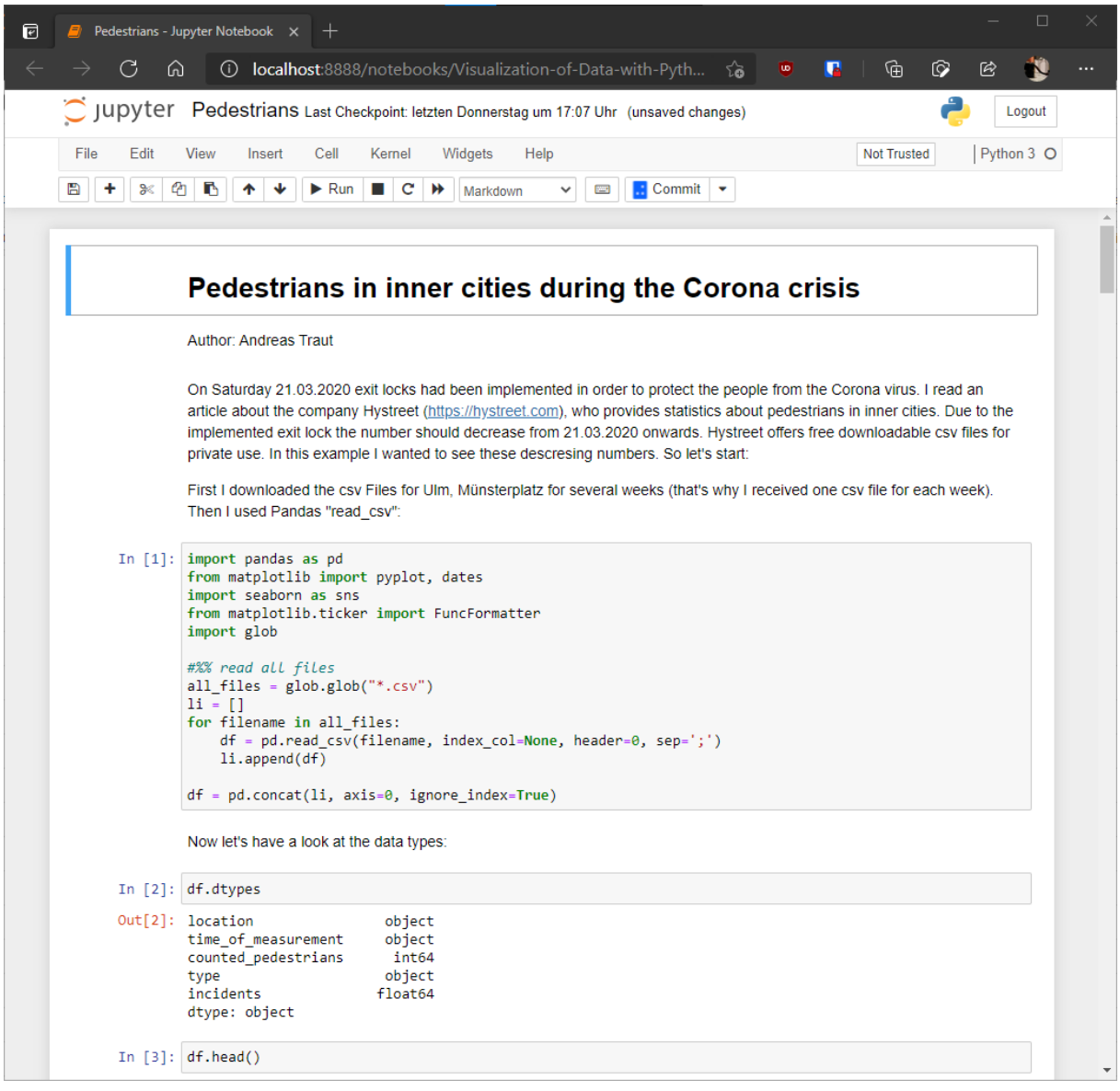


Figure 2.2: Jupyter-Notebook: Code and Documentation in one place.

A Jupyter-Notebook runs in a browser (like Chrome, Edge, Firefox) and combines code and documentation. The advantages are, that it looks beautiful and easy to share with other people. The disadvantage is, that there is no code-debugger. It looks like shown in figure fig. 2.2.

As I think that you should get familiar with both (the Jupyter-Notebooks and the Spyder-IDE), I recommend installing the Anaconda distribution[4]. The installation is very simple and includes the Jupyter

[4] Anaconda Distribution Installation, https://www.anaconda.com/products/individual#Downloads

Notebooks as well as the Spyder-IDE and a lot more Data Science applications, which might be useful for you later.

2.2 Motivation for IDEs

The first Jupyter-Notebooks have been developed 5 years ago (in 2015). Since my first programming experience was more than 25 years ago (I started with GW-Basic[5] then Turbo-Pascal[6]and so on and I am also familiar with MS-DOS[7]). I quickly learnt the advantages of using Jupyter-Notebooks. **But** I missed the comfort of an IDE from the very first days!

Why is it important for me to mention the IDEs out so early in a learning process? In my opinion Jupyter-Notebooks are good for the first examinations of data and for documenting procedures and up to a certain degree also for sophisticated data science. But it is a good idea to learn very early how to work with an IDE. I point this out here, because after having read several e-Books and having participated in seminars I see that IDEs are not in the focus.

In my opinion Jupyter Notebooks are **not** always the best environment for learning to code! I agree, that Jupyter Notebooks are nice for doing documentation of python code. It really looks beautiful. But I prefer debugging python code in an IDE instead of a Jupyter-Notebook: having the possibility to set a breakpoint can be a pleasure for my nerves, specially if you have longer programs. Some of my longer Jupyter Notebooks feel from the hundreds line of code onwards more like pain than like anything helpful. Using an IDE makes it easier for you to split a program into several subprograms.

In an IDE I also appreciate having a “help window” or a “variable explorer”, which is smoothly integrated into the IDE user interface. And there are a lot more advantages why getting familiar with an IDE is a big advantage compared to the very popular Jupyter Notebooks!

I am very surprised, that everyone is talking about Jupyter Notebooks but IDEs are only mentioned very seldom. But maybe my preferences are also a bit different, because I grew up in a MS-DOS environment. :-)

[5] GW-Basic, https://de.wikipedia.org/wiki/GW-BASIC

[6] Turbo-Pascal, https://de.wikipedia.org/wiki/Turbo_Pascal

[7] MS-DOS, https://de.wikipedia.org/wiki/MS-DOS

2.3 Docker

For more advanced Data Science techniques, like the Big Data approaches on Apache Spark[8]and Hadoop[9], you will need Docker[10]. Download it from "Docker - Get Started"[11] and also create an account on the Docker website. The installation is easy.

What is Docker? Docker is *"an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux."*[12]

After having installed it, you are able to pull my "machine-learning-pyspark" image to your computer and run my Jupyter-Notebooks. I will explain in section sec. 4 "Machine Learning with Python" how it works.

Please read also the Docker-Curriculum[13] for more information. It is a very well structured and nice tutorial which I can recommend for learning about docker images, docker containers and more.

[8] Apache Spark, https://spark.apache.org/
[9] Hadoop, https://hadoop.apache.org/
[10] Docker, https://www.docker.com/
[11] Docker Get Started, https://www.docker.com/get-started
[12] What is Docker, https://digital.ai/technology/docker
[13] Docker-Curriculum, https://docker-curriculum.com/

3 Visualization of Different Datasets

In the data scientist environment the visualization is as important as the analysis itself and the work of a "Data Scientist" with data covers many areas. The Cross-Industry Standard Process for Data Mining (CRISP-Cycle)[1], which "Data Scientists" like to quote and follow in their work, describes what these are (see fig. 3.1).

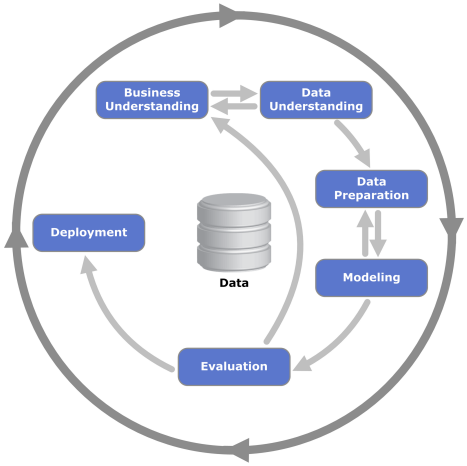


Figure 3.1: Cross-Industry Standard Process for Data Mining - CRISP-Cycle, Wikipedia, CC BY-SA 3.0

Here, the visualization of data at any point in the cycle plays a role:

Visualization helps to get a **better understanding of the data** ("Data understanding"), but also to get a **better understanding of the business model** ("Business understanding"). Visualization **supports data preparation** ("Data preparation"), for example, when it comes to identifying gaps or outliers or to get an indication by means of distribution diagrams which models might be suitable to solve a specific problem (e.g. classification, clustering, regression). Visualization helps in **modeling** ("modeling") to illustrate whether the models are useful. Visualization is also particularly important in **validation** ("evaluation") to show how meaningful the applied models are compared to the real world.

So you can see: Visualization techniques are useful at every development step (CRISP cycle). Often visualizations are as important for a "Data Scientist" as the analysis itself. Without a good visualization it is often difficult for a data scientist to communicate the result of his analysis to an outsider. This is

[1] Cross-Industry Standard Process for Data Mining (CRISP-Cycle), Wikipedia, CC-BY SA 3.0, https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining

also due to the fact that the human eye can recognize patterns in visualizations very quickly, but on the other hand has difficulty understanding a table with numbers at a glance. In individual cases, a good visualization can even have the effect that a complicated model can be completely dispensed with and one only deals with patterns. There are numerous examples of this, which I will not go into here.

I worked on different datasets with the aim to visualize the data and in the **first part** of this chapter (see sec. 3.1) I will explain these examples. In the **second part** (see sec. 3.2) I will explain some Big Data visualizations problems and techniques to solves these. In the **third part** (see sec. 3.3) I will show you how to visualize and share the data with a "data app". In the **fourth part** sec. 3.4 I will list some common professional tools, which offer visualization functionality and more.

3.1 Examples

In this chapter I work on different examples with the aim to visualize the data with e.g. line plots, histograms, bar chars. These examples consider different topics (consumer price, music, pedestrians,…), different format of the data (csv-file, API[2]) and cover various problems, that arise, when you work with data (e.g. converting, encoding formats,…). I used python and libraries like e.g Matplotlib or Seaborn, which are available for free. My examples are:

1. A public dataset of the "Consumer Price Index" from the official statistics website of the "Bayerisches Landesamt für Statistik"[3].
2. A dataset of my own songs, which I listened to (66'955 songs since 2016) and downloaded from LastFM[4].
3. An artificially treated dataset of "Marathon run-times", where I showed how systematics in the data can be found by visualizing.
4. The number of pedestrians in inner cities when the Corona-lockdown had been implemented.
5. The data from the Deutsche Bahn API to monitor status of their station elevators. Working with an API is a bit different, but interesting.
6. A very brief introduction into visualization of Big Data as this can be challenging with standard visualizing techniques.

The examples are available:

- as `.py` files for being used for example in an Spyder-IDE[5] and
- as `.ipynb` files, which are Jupyter-Notebooks[6].

[2] Application Programming Interface (API), Wikipedia, https://en.wikipedia.org/wiki/API
[3] Bayerisches Landesamt für Statistik, www.statistikdaten.bayern.de
[4] LastFM, www.last.fm
[5] Spyder-IDE, https://www.spyder-ide.org/
[6] Jupyter-Notebook, https://jupyter.org/

I explained in sec. 2.2, why I think, that working with an IDE is very important and has a lot of advantages compared to the Jupyter-Notebooks, which you find in nearly every Data Science book.

3.1.1 Consumer Price Index Example

In this example I will convert dates and visualize the data using the seaborn regression plot. First download the consumer prices[7]. The CSV-file has the following format (the columns are separated by semicolons):

	A	B	C	D	E
1	GENESIS-Tabelle: 61111-202z				
2	Verbraucherpreisindex (2015=100): Bayern, Verbraucherpreise,				
3	Monate, Jahre				
4	Verbraucherpreisindex				
5	Bayern				
6			Verbraucherpreisindex		
7			2015=100		
8	1970	Januar	29,6		
9	1970	Februar	29,7		
10	1970	März	29,8		
11	1970	April	29,8		
12	1970	Mai	29,9		
13	1970	Juni	30,0		
14	1970	Juli	30,0		
15	1970	August	30,1		
16	1970	September	30,1		

Figure 3.2: Consumer Price Index

Delete the first 7 lines (which are only some metadata) and save the file as `61111-202z-bearbeitet.csv` (this is the easiest way to do it, but you can also program this task in Python). The new file has the following format:

```
1970    Januar  29,6
1970    Februar 29,7
1970    März    29,8
1970    April   29,8
1970    Mai     29,9
1970    Juni    30,0
1970    Juli    30,0
```

The preliminary work here is to convert the months (e.g. “Januar”, “Februar”) to a date. Therefore I import some libraries:

[7] Consumerprice of “Bayerisches Landesamt für Statistik”, https://www.statistikdaten.bayern.de/genesis/online?sequenz=statistikTabellen&selectionname=61111. Code: 61111-202z

```
import pandas as pd
from matplotlib import pyplot, dates
from matplotlib.ticker import FuncFormatter
import seaborn as sns
from time import strptime
import locale
locale.setlocale(locale.LC_ALL, '')
```

Pandas is a library that simplifies working with data structures[8]. It includes the "dataframe" which has many useful functions for combining, splitting, grouping datasets and has a better performance than the original Python data structures.

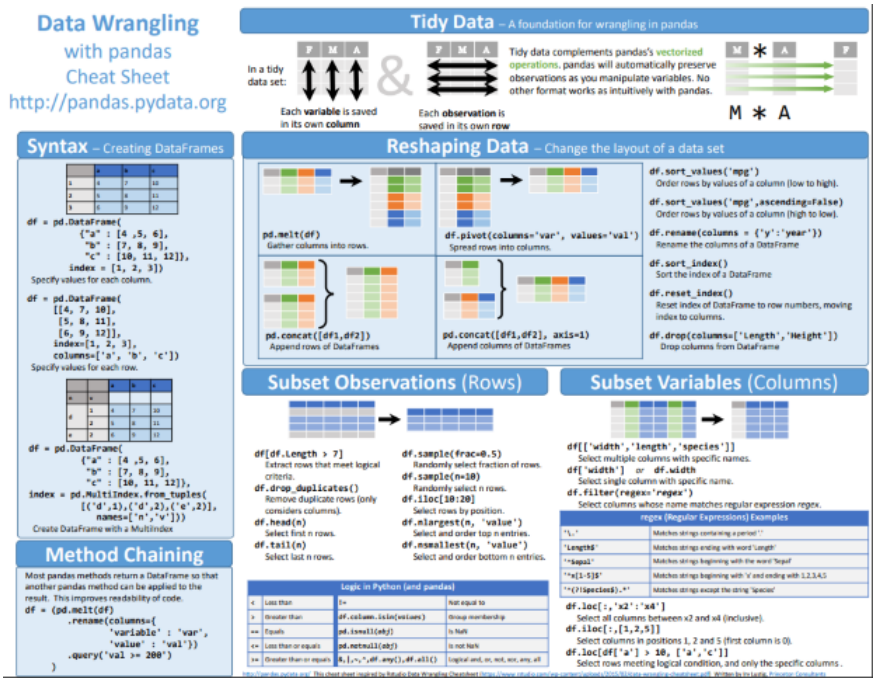


Figure 3.3: Pandas - Overview, Written by Irv Lustig[9]

Matplotlib is a library for the creation of high quality plots and usually already fulfills many needs, when you want to customize your plots. Another interesting library for creating graphics is the seaborn library, which is based on matplotlib and provides high-level interfaces and even more functionality on creating and customizing your plots.

[8] Pandas Licence, https://github.com/pandas-dev/pandas/blob/master/LICENSE

[9] Pandas Overview, Written by Irv Lustig, https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf

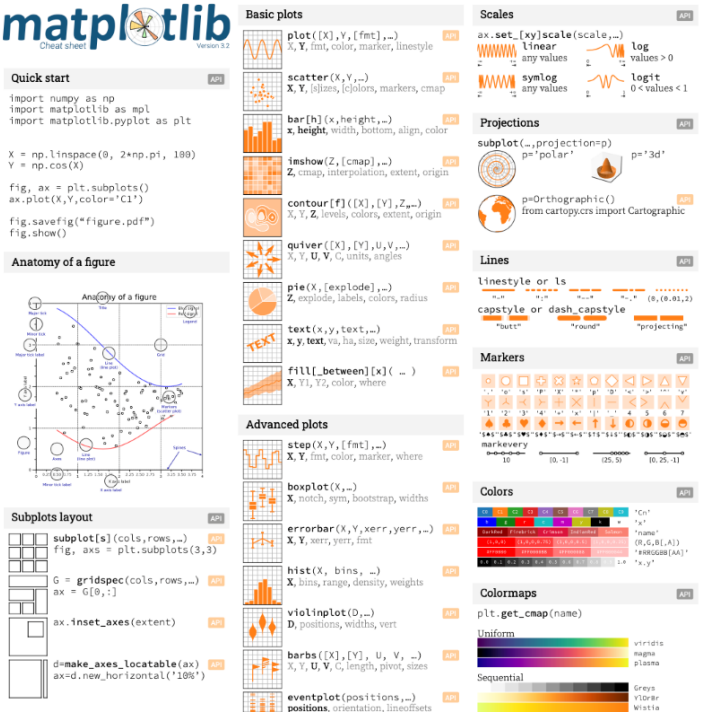


Figure 3.4: Matplotlib - Overview, Nicolas P. Rougier, BSD 2 Licence[10]

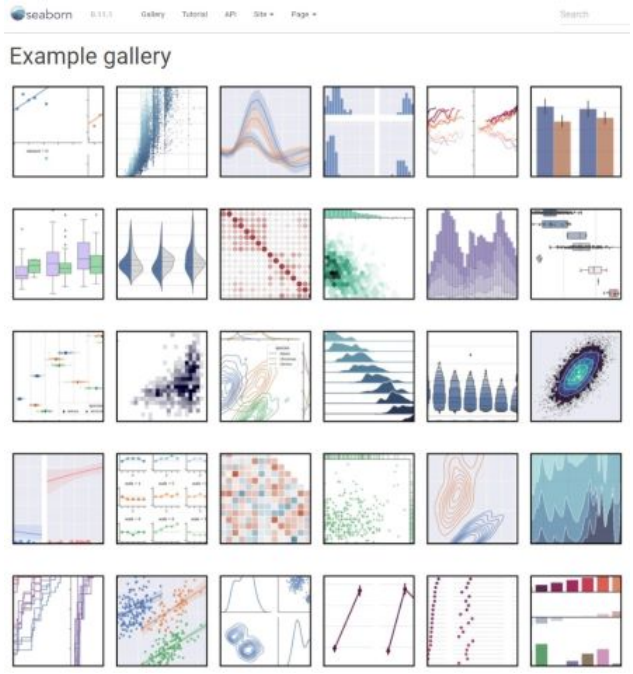


Figure 3.5: Seaborn - Statistical Data Visualization Examples

[10] Matplotlib - Overview, Nicolas P. Rougier, BSD 2 Licence, see https://github.com/matplotlib/cheatsheets/blob/master/LICENSE.txt

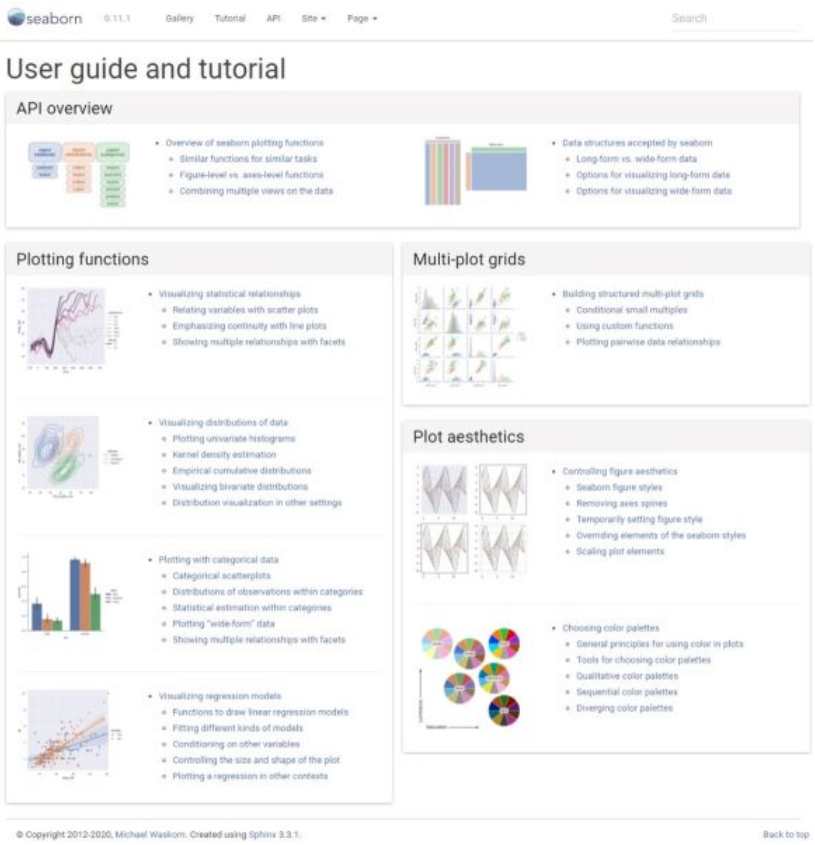


Figure 3.6: Seaborn - Statistical Data Visualization, Journal of Open Source Software 6(60) 3021, Copyright Michael Waskom[11]

As a consequence the result when importing directly with `pandas.read_csv` would be as follows, which is not what we want:

	year	month	value
0	1970;Januar;29	6.0	NaN
1	1970;Februar;29	7.0	NaN
2	1970;März;29	8.0	NaN
3	1970;April;29	8.0	NaN
4	1970;Mai;29	9.0	NaN

We can see, that the `pandas.read_csv` did not use the semicolon as a separator for the columns,

[11] Seaborn - Statistical Data Visualization, Journal of Open Source Software 6(60) 3021, Copyright Michael Waskom, see https://seaborn.pydata.org/citing.html and https://doi.org/10.21105/joss

but instead comma (which we saw in the number 29,6 and 29,7 and so on). It is a good exercise to learn how to convert different language settings, because you might have the same issue, when downloading datafiles from US and using it in your country.

We have to define `converters`, which I named `myMonthConverter` (converts "Januar" to 1, "Februar" to 2, ...) and `myValueConverter` (converts 29,6 to 29.6) as follows:

```
def myMonthConverter(s):
    return strptime(s,'%B').tm_mon

def myValueConverter(s):
    return s.replace(',', '.')

def fake_dates(x, pos):
    """ Custom formater to turn floats into e.g., 2016-05-08"""
    return dates.num2date(x).strftime('%Y-%m-%d')
```

Now read the csv file again, with defining the semicolon as separator and using Encoding=latin-1. It also uses the converter functions `myMonthConverter` and the `myValueConverter` from above.

```
df=pd.read_csv('61111-202z-bearbeitet.csv',
               sep=";",
               encoding="latin-1",
               names=['year', 'month', 'value'],
               converters={'month':myMonthConverter,
                           'value': myValueConverter})
df.head()
```

The result is as follows:

	year	month	value
0	1970	1	29.6
1	1970	2	29.7
2	1970	3	29.8
3	1970	4	29.8
4	1970	5	29.9

This looks much better. Now have a look at the column `value`. The type of the column `value` is not a number, but an object. Type `df.dtypes` to know which kind of data type we have:

```
year      int64
month     int64
value    object
dtype: object
```

Therefore I need to convert the column `value` from `object` into a Pandas `numeric` :

```
df['value'] = df['value'].apply(pd.to_numeric, errors='coerce')
```

Now we are done with reading the csv and converting the columns. Of course you can convert any kind of value by defining an appropriate function. Imagine, that you have for example a column, which contains the banking ratings and you want to convert them to numbers (e.g. AAA=1, AA=2, A=3 and so on). We know that there are different rating companies (Moodys&Fitch, Standard&Poors), with slightly different rating tables. Aligning both to a numeric rating scale is needed to work with the data. This kind of exercise is now easy, when you use a `myconverter` function as above.

As a next step I created new columns `datenum` and `date`, which I need for the graphics (the regression plot). The column `datenum` is a step, which I needed because of the Seaborn regression plot function `sns.regplot`. First the real date (e.g. "1970-01-01") need to be converted to a number (e.g. 719163.0), which is used in the x-Axis of the plot. Then the description of the x-Axis is transformed from 719163.0 back to the real date in a string-format. This seems to be a bit strange, but according to forums this is how it has to be done.

```
df['date'] = df['year'].astype(str) + "-" + df['month'].astype(str) + "-1"
df['datenum'] = dates.datestr2num(df['date'])
df['date'] = df['date'].apply(pd.to_datetime, errors='coerce')
df.head()
```

	year	month	value	date	datenum
0	1970	1	29.6	1970-01-01	719163.0
1	1970	2	29.7	1970-02-01	719194.0
2	1970	3	29.8	1970-03-01	719222.0
3	1970	4	29.8	1970-04-01	719253.0
4	1970	5	29.9	1970-05-01	719283.0

The data types are (use `df.dtypes`):

```
year                 int64
month                int64
value              float64
date       datetime64[ns]
datenum            float64
dtype: object
```

Finally I have everything I need for the regression Plot (`regplot`) and I am reading for the visualizing task. I use "seaborn" for this and I recommend to have a look into the official seaborn documentation for learning more about the seaborn library:

```
#Color settings
sns.set(color_codes=True)

#Plot 'datenum' (=float) and 'value' (=float)
fig, ax = pyplot.subplots()
sns.regplot('datenum',
            'value',
            data=df,
            ax=ax)

#Create the x-axis which is 'datenum' converted to %Y-%m-%d
ax.xaxis.set_major_formatter(FuncFormatter(fake_dates))
ax.tick_params(labelrotation=90)
fig.tight_layout()
```

This is what we wanted: a Seaborn regression plot (`seaborn.regplot`), which required me to convert the x-axis from date to a number (see fig. 3.7).

I would say, that the fig. 3.7 doesn't show anything noticeable or interesting (apart from a fairly steady rise in consumer prices over the whole period from 1970 until 2020). A bit disappointing so far.

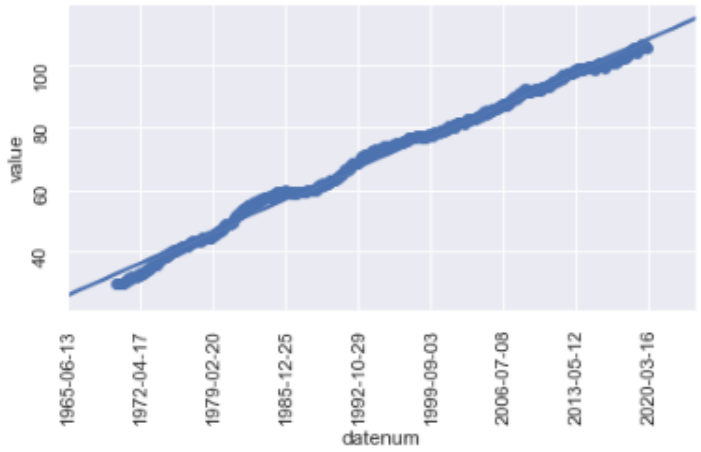


Figure 3.7: Consumer Price Index Figure

Now let's examine the increments (absolute and relative):

```
import matplotlib.ticker as mtick
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()

#Examine increments (absolute and relative)
df['increment_abs'] = df['value']-df['value'].shift(+1)
df['increment_rel'] = (1-df['value'].shift(+1)
                       /df['value'])*100
```

Replace the "NaN" values:

```
#Replace "NaN" by 0
df['increment_abs'].fillna(0, inplace=True)
df['increment_rel'].fillna(0, inplace=True)
```

And create a plot:

```
figin, axin = pyplot.subplots(2)
axin[0].plot(df['date'], df['increment_abs'])
axin[1].plot(df['date'], df['increment_rel'])

#Range of axis
axin[0].set_ylim([-1.1, +1.1])
axin[1].set_ylim([-1.1, +1.7])
```

```
#Title of axis
axin[0].set_title('absolute increment')
axin[1].set_title('relative increment')
for ax in figin.get_axes():
    ax.label_outer()

#Format y axis in percent
axin[1].yaxis.set_major_formatter(mtick.PercentFormatter(decimals=0))
```

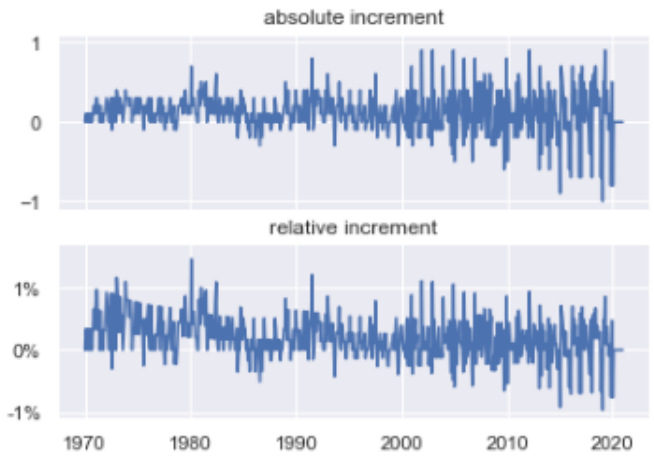


Figure 3.8: Consumer Price Figure - absolute and relative increments

The fig. 3.8 shows the increments (absolute and relative increments) and now we can see something interesting: perhaps one can say that the consumer prices grew more evenly between 1970 and 1995 and that the growth was almost entirely positive (abs/rel increments is above zero). On the other hand, the changes between 1995 and 2020 were somewhat more volatile and increases (positive changes of consumer prices) alternated with decreases (negative changes).

As this was interesting to me I tried to find an explanation or some evidence if we really could split up the whole period (from 1970 until 2020) in one going from 1970 until 1995 and in another one going from 1995 until 2020. It was funny for me, when I found the image fig. 3.9, which shows the consumer prices and stock rates. Wouldn't you say, that the stock prices were also more volatile between 1995 and 2020? Even more interesting: the volatility of the stock prices increased already in 1990 (five years ahead of the consumer prices).

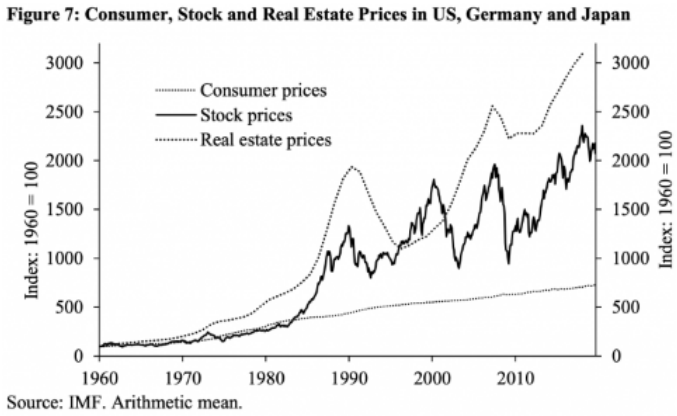


Figure 3.9: Consumer price index versus Stockprices, Real estate prices

Let's have a look into the tail of the data:

```
df.tail(15)
```

	year	month	value	date	datenum	increment_abs	increment_rel
597	2019	10	106.6	2019-10-01	737333.0	0.1	0.093809
598	2019	11	105.8	2019-11-01	737364.0	-0.8	-0.756144
599	2019	12	106.3	2019-12-01	737394.0	0.5	0.470367
600	2020	1	105.5	2020-01-01	737425.0	-0.8	-0.758294
601	2020	2	NaN	2020-02-01	737456.0	0.0	0.000000
602	2020	3	NaN	2020-03-01	737485.0	0.0	0.000000
603	2020	4	NaN	2020-04-01	737516.0	0.0	0.000000
604	2020	5	NaN	2020-05-01	737546.0	0.0	0.000000
605	2020	6	NaN	2020-06-01	737577.0	0.0	0.000000
606	2020	7	NaN	2020-07-01	737607.0	0.0	0.000000
607	2020	8	NaN	2020-08-01	737638.0	0.0	0.000000
608	2020	9	NaN	2020-09-01	737669.0	0.0	0.000000
609	2020	10	NaN	2020-10-01	737699.0	0.0	0.000000
610	2020	11	NaN	2020-11-01	737730.0	0.0	0.000000
611	2020	12	NaN	2020-12-01	737760.0	0.0	0.000000

There are some "NaN" values and I will give you right now a short glance into how machine-learning works come back to this later in sec. 4 again. I want to predict the values for the "NaN", which we have from February 2020 on. In order to do this I have to eliminate some columns and split the whole dataset `df_num` in one which has the value, which I want to predict `df_value` and one, which contains the remaining columns `df_prepared`:

```
df_num = df.dropna(subset=["value"])
df_value = df_num["value"]
df_prepared = df_num.drop(["datenum", "date", "value",
                           "increment_abs", "increment_rel"],
                          axis = 1)
```

I am ready for fitting the `LinearRegression`:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(df_prepared, df_value)
```

I can now use `predict` for example for the last 20 entries:

```
some_data = df_prepared.iloc[-20:]
some_values = df_value.iloc[-20:]
df_some_predictions = lin_reg.predict(some_data)
df_some_predictions
```

```
array([105.7954812 , 105.91550781, 106.03553442, 106.15556103, 106.27558764,
 ↪  106.39561425, 106.51564086, 106.70133425, 106.82136086, 106.94138747,
 ↪  107.06141408, 107.18144069, 107.3014673 , 107.42149391, 107.54152052,
 ↪  107.66154713,107.78157374, 107.90160035, 108.02162696, 108.20732036])
```

My linear regression has found the values for the last 20 entries. I can also do the same for the whole dataset as follows:

```
df_predictions = lin_reg.predict(df_prepared)
print("Predictions:", list(df_predictions))
```

The "mean squared error" is 1.775 can be calculated as follows:

```
from sklearn.metrics import mean_squared_error
import numpy as np
lin_mse = mean_squared_error(df_value, df_predictions)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

Obviously I could have done this also in Excel, but as I am now in the Python framework, I can apply more tools on the data, which I will do in a next step. For example: as it seems that there is a connection between stock prices and consumer prices wouldn't it be nice to analyze if more "variables" (like the stock prices) could be found? And wouldn't it be interesting to create some sort of "predicting tool", which calculates the consumer prices index for me for a future date (remember, that the volatility of stock prices increased years before the consumer price index did, so the stock price could perhaps be a "predicting variable" for the consumer price index)? We already know, that there are some nice Python packages for doing this. This would be a task for a next step.

On my GitHub-profile you can download my Jupyter-Notebook[12].

3.1.2 Last-FM Statistics of my Songs

I am listening quiet a lot to music, either with my app on my mobile phone or my home sound-system. Since 2016 I am using Last-FM[13] to upload my music statistics (so called "scrobbling"). Last-FM is creating nice graphics for 2019 as shown in fig. 3.10. You can see that in 2019 I listened to 12,052 songs in total, which is 33 songs (=scrobbles) per day. The bar charts in the middle splits this up to a monthly view. A listening clock (="Höruhr"), shows when I was mainly listeing during the day. Not surprisingly the main part is in the evening around 18:00.

In this example I download my complete history of played songs since 2016 from Last-FM (66'955 songs in total) and re-built some of these nice statistics and figures, which last.fm provides. This are for example a bar-plot with monthly aggregates of total played songs. Or top 10 songs of the week and so on. Having the same plots at the end as last.fm has proves, that my results are correct. :-)

[12] Consumer-Prices Jupyter-Notebook, https://github.com/AndreasTraut/Visualization-of-Data-with-Python/blob/main/ConsumerPricesExample/ConsumerPrices.ipynb

[13] LastFM, www.last.fm

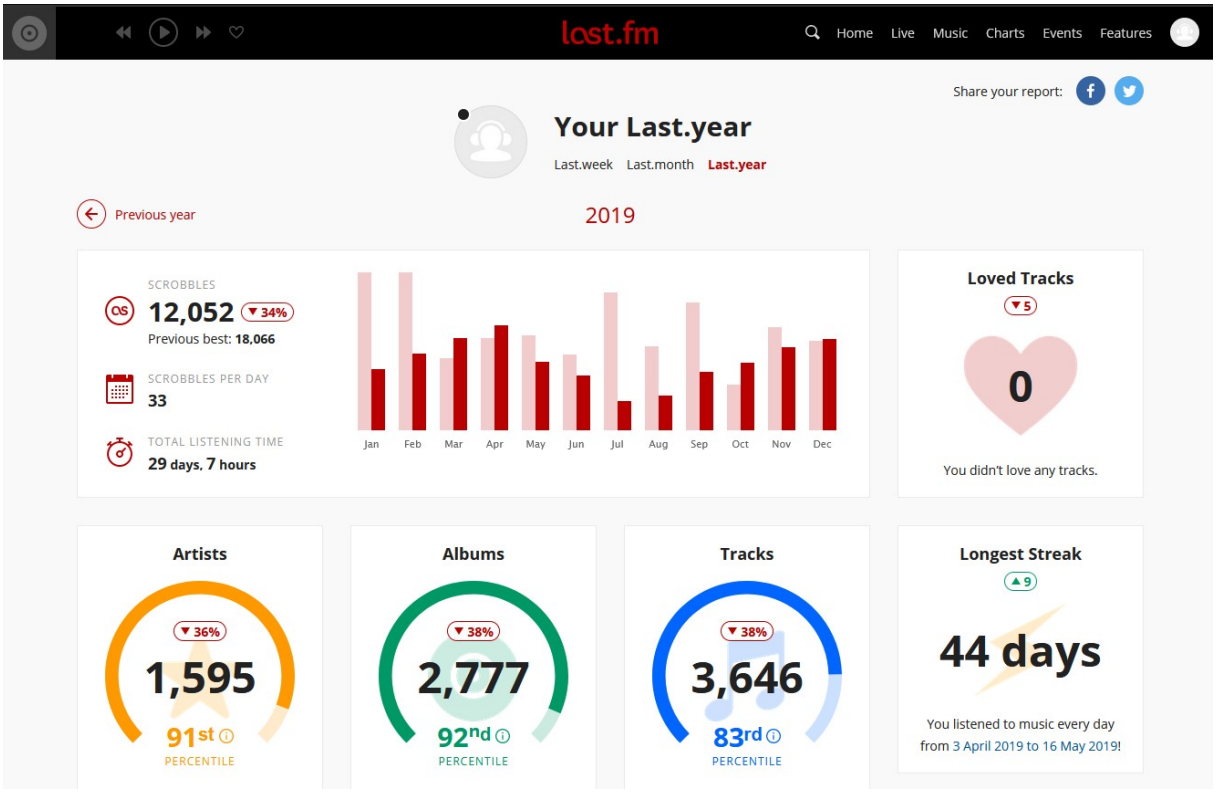


Figure 3.10: Last-FM Music Statistics - Overview year 2019

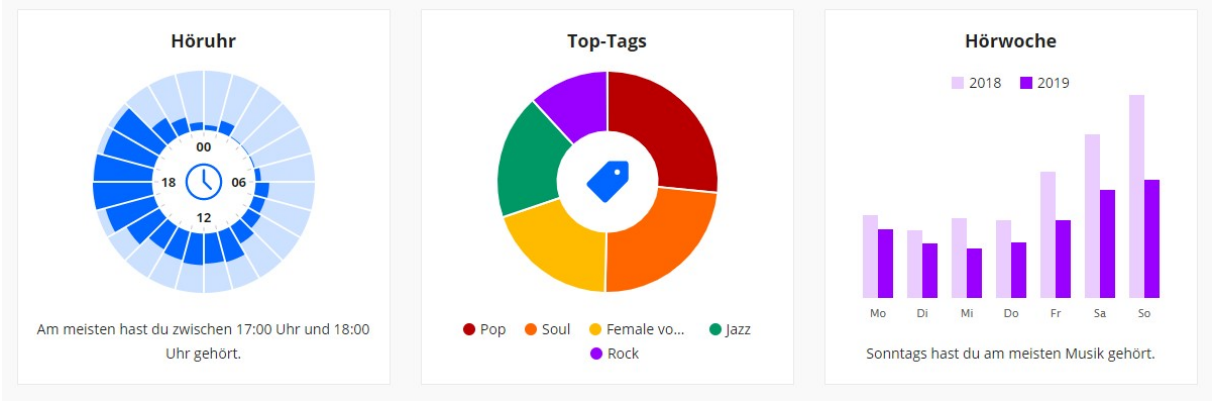


Figure 3.11: Last-FM Music Statistic - Listening clock 2019

The CSV file had the following shape:

	A	B	C	D
1	Daniel Santacruz		Lento	06.02.2020 16:45
2	Mau y Ricky	Para Aventuras y Curiosidades	Mi Mala	06.02.2020 16:27
3	Nelson Freitas	Elevate	Something Good	06.02.2020 16:23
4	Jennifer Dias	Love U	Love U	06.02.2020 16:22
5	Nelson Freitas	Sempre VerÃ£o	Every Day All Day	06.02.2020 16:18
6	Daniel Santacruz	Lento	Lento	06.02.2020 16:16
7	Mogli	Wanderer (Expedition Happiness Soundtrack)	Road Holes	05.02.2020 15:49
8	Serena Ryder	Harmony (Deluxe)	For You	04.02.2020 17:36
9	Y'akoto	Perfect Timing	Perfect Timing	04.02.2020 17:32
10	Awa Ly	FIVE AND A FEATHER	LET ME LOVE YOU	04.02.2020 17:28

Figure 3.12: Last-FM Music Statistics - Format of the Databasis

Obviously the columns are 'artist', 'album', 'song', 'timestamp'. First I wanted to reproduce the overall statistics, which is (as you can see from the screenshot above) 12'052 songs in total for 2019 and 33 songs per day.

```
import pandas as pd
import numpy as np
from matplotlib import pyplot
df = pd.read_csv('lastfm_data.csv',
                 names=['artist', 'album', 'song', 'timestamp'],
                 converters={'timestamp':pd.to_datetime})
```

We already know the libaries `pandas` and `matplotlib` from sec. 3.1.1.

Numpy is a library that provides an infrastructure for scientific computation[14]. It contains powerful arrays, lists and matrices and many useful numerical functions.

First I extracted the `year / month / date / weeofyear / hour / weekday` from the timestamp:

```
#%% Extracting year/month/... from timestamp and adding as new columns
dates = pd.DatetimeIndex(df['timestamp'])
df['year'] = dates.year
df['month'] = dates.month
df['weekofyear'] = dates.weekofyear
df['hour']= dates.hour
df['weekday'] = dates.weekday    #Monday=0
```

Next I wanted to have the overall statistics as for example "played songs per year" or "scrobbels per day".

[14] NumPy Licence, https://numpy.org/devdocs/license.html and Academic Publication: Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). DOI: 0.1038/s41586-020-2649-2. (Publisher link).

```
#%% Overall statistics
print("\nPlayed songs per year:\n{}"
      .format(df['year'].value_counts(sort=False)))
print("\nScrobbels per day:\n{}"
      .format(df['year'].value_counts(sort=False)/365.))
```

This is what I found:

```
2018:   18'066 songs in total and 49.495890 songs-per-day.
2017:   12'521 songs in total and 34.304110 songs-per-day.
2016:   10'726 songs in total and 29.386301 songs-per day.
```

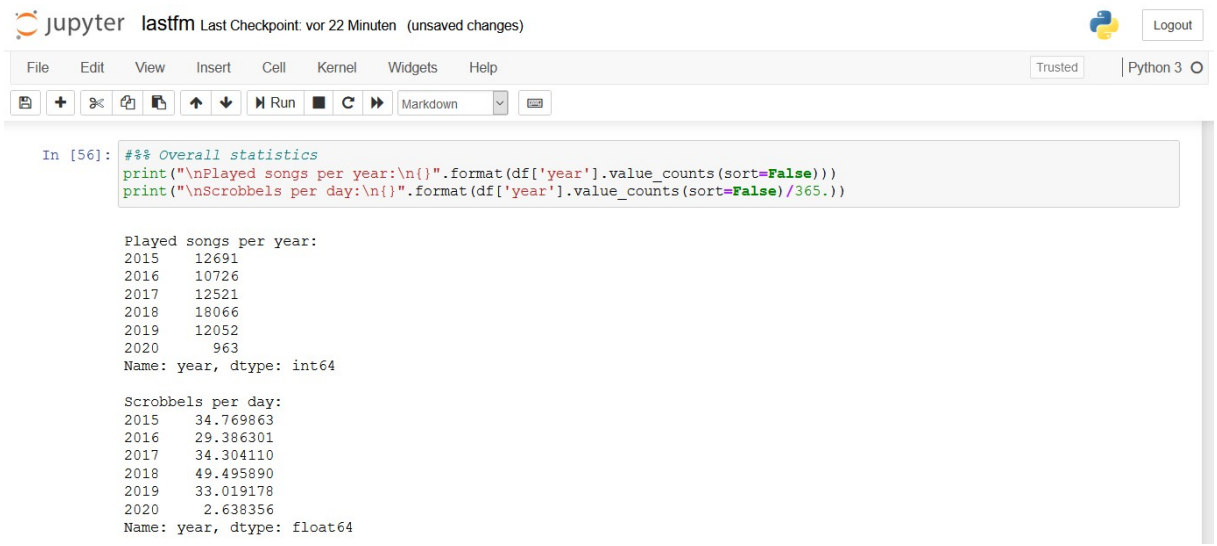


Figure 3.13: Last-FM Music Statistics - Overall statistics

These are exactly the same numbers, as Last-FM showed me. So everything is fine so far. Now I even know that the accurate number is 33.019 songs per day! For the year 2018 I calculated 49.495890.

Now lets examine the “top artist”, “top album”, “top songs”:

```
print("\nTop artists:\n{}"
      .format(df['artist'].value_counts().head()))
print("\nTop album:\n{}"
      .format(df['album'].value_counts().head()))
print("\nTop songs:\n{}"
      .format(df['song'].value_counts().head(10)))
```

```
Top artists:
```

```
Aretha Franklin     1284
Caro Emerald         925
Paloma Faith         895
Dionne Bromfield     739
Nikki Yanofsky       678
Name: artist, dtype: int64

Top album:
Soul Queen                                          576
Good for the Soul                                   508
Do You Want the Truth or Something Beautiful?       451
Greatest Hits                                       405
Emerald Island EP                                   394
Name: album, dtype: int64

Top songs:
Without You                     185
He's So Fine                    158
Good for the Soul               152
It's A Beautiful Day            149
Hallelujah                      149
This Guy's In Love With You     135
White Christmas                 126
Cheek to Cheek                  123
Yeah Right                      117
Tangled Up                      115
Name: song, dtype: int64
```

Next, I want to define a year / month / weekofyear and see some more detailed statistics:

```
#%% Defining a year / month / weekofyear for examination
myYear = 2018
myMonth = 5
myWeekofYear = 21

#%% Examine selected year
print("\nAll songs in year %s:\n"%(myYear),
      df.loc[df['year'] == myYear,
             ['artist', 'album', 'song']])
selection = df.loc[df['year'] == myYear,
```

```
                    ['artist', 'album', 'song', 'month']]
selectionPrev = df.loc[df['year'] == myYear-1,
                       ['artist', 'album', 'song', 'month']]
print("\nTop artists:\n{}"
      .format(selection['artist'].value_counts().head()))
print("\nTop songs:\n{}"
      .format(selection['song'].value_counts().head(10)))
```

As a next step I wanted to reproduce the bar chart (monthly aggregates of songs):

```
index = np.arange(12)
pltperMonth = pyplot.bar(index, perMonth, width=0.3,
                         label=myYear, color='red')
pltperMonthPrev = pyplot.bar(index - 0.3, perMonthPrev,
                             width=0.3, label=myYear-1,
                             color='peachpuff')
pyplot.title('Year {}. Scrobbels per month.'.format(myYear))
pyplot.xticks(index, ('Jan', 'Feb', 'Mär', 'Apr', 'Mai',
                      'Jun', 'Jul', 'Aug', 'Sep', 'Okt',
                      'Nov', 'Dez'))
pyplot.legend()
pyplot.show()
```

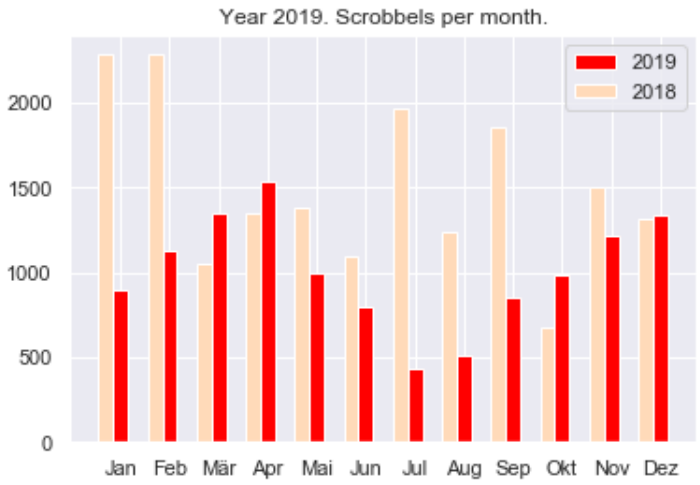


Figure 3.14: Last-FM Music Statistics - Reproduced Statistics of 2019

Nice: it looks also the same, but I can customize mine as I want. For example: I always missed the y-Axis in the Last-FM Chart, which I have now.

Last-FM graphics for 2018:

Figure 3.15: Last-FM Music Statistics - Overview year 2018

My graphics for 2018:

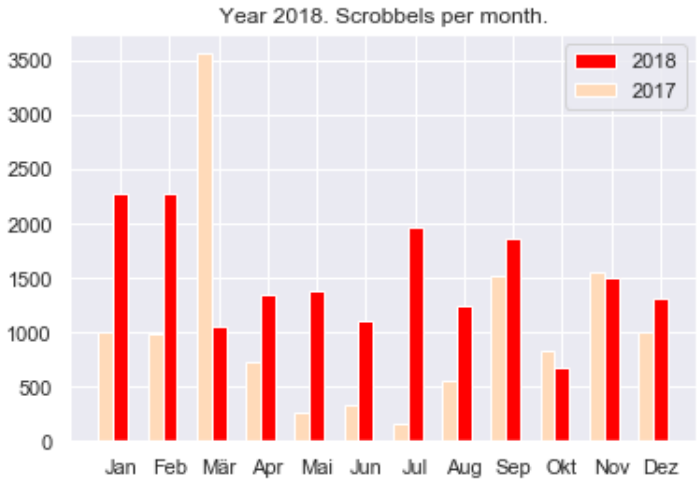


Figure 3.16: Last-FM Music Statistics - Reproduced Statistics of 2018

As you can see, Last-FM shows 49 songs per day for 2018. Remember, that I recalculated 49.495890 (as you can see in the Screenshot above), based on 365 days per year (when I take 365.25 days per year in order to reflect the leap years, I get 49.462). Applying the rounding rules both is rounded to 49 (not 50!). So Last-FM is correct.

Last-FM graphics for 2017:

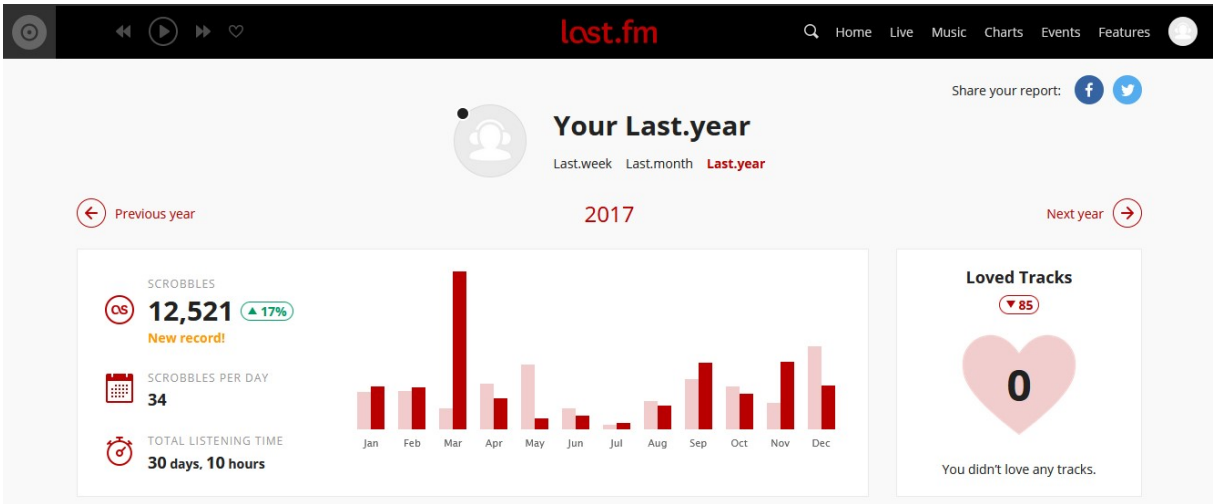


Figure 3.17: Last-FM Music Statistics - Overview year 2017

My graphics for 2017:

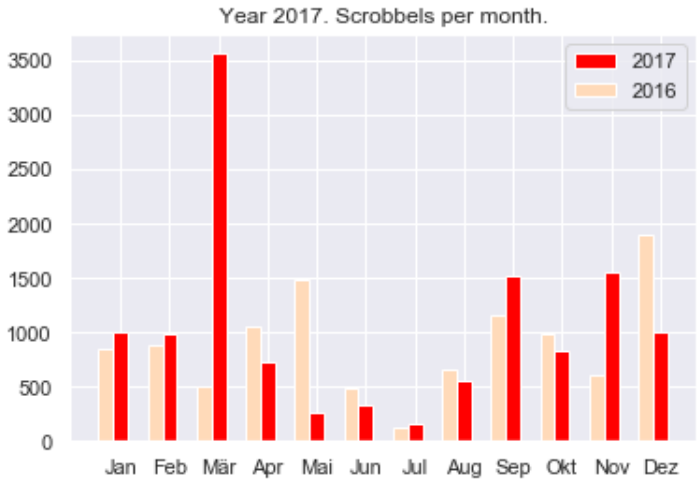


Figure 3.18: Last-FM Music Statistics - Reproduced Statistics of 2017

And as a last exercise I want to create a “Listening Clock”, which is a barplot showing the hours and number of songs I listened to. Obviously I listened mostly in the evening:

```
#%% Listening Clock
isweekofyear= (df['weekofyear'] == myWeekofYear)
selection = df.loc[isyear & isweekofyear, ['hour']]
index = np.arange(24)
perHour = myselection['hour'].value_counts().sort_index()
pltperHour = pyplot.subplot(111)
```

```
pltperHour.bar(perHour.index,
               perHour,
               width=0.3,
               color='blue',
               alpha=0.5)
pltperHour.set_xticks(index)
pyplot.title('Year {}, Week-of-Year {}. Scrobbels per hour.'
             .format(myYear, myWeekofYear))
pyplot.show()
```

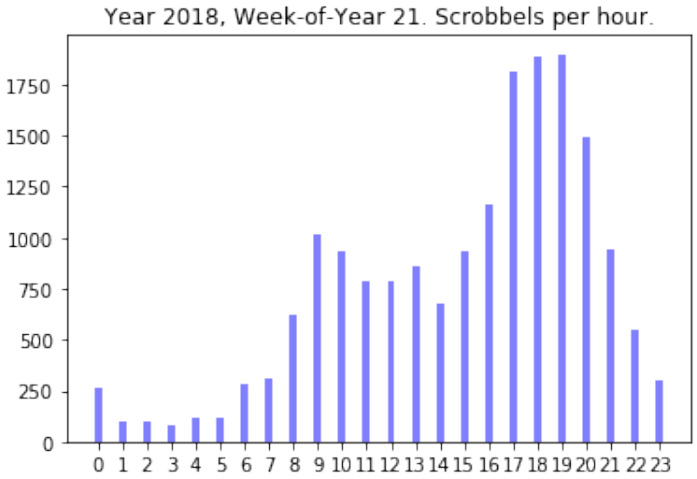


Figure 3.19: Last-FM Music Statistics - Listening clock 2018 (week 21)

On my GitHub profile you can find my Jupyter-Notebook for this example[15]

3.1.3 Marathon Runtimes: Finding Systematics

This example shows how different visualization techniques in Python (by using the libraries seaborn and matplotlib) can be used to find out whether there are dependencies, systematics or relationships in a dataset.

Imagine that you receive the following csv data record of 37'250 lines (it is an artificially treated dataset and only for exercise purposes). They show the age, gender and times of marathon runs as well as their nationality, size and weight. Are there any hidden relationships in the data records?

[15] Last-FM Jupyter-Notebook, https://github.com/AndreasTraut/Visualization-of-Data-with-Python/blob/main/LastFMExample/lastfm.ipynb

```
age,   gender,  split,      final,     nationality,  size,     weight
33,     M,      01:05:38  02:08:51,      DE,       183.41,    84.0
32,     M,      01:06:26  02:09:28,      DE,       178.61,    87.7
31,     M,      01:06:49  02:10:42,      IT,       171.94,    82.2
38,     M,      01:06:16  02:13:45,      IT,       172.29,    82.4
31,     M,      01:06:32  02:13:59,      IT,       178.59,    79.8
....
```

Download the csv file from my repository and examine the data. At the first glance you won't find anything unusual, but using the following visualization techniques in Python will lead to some conclusions. First we need to import the csv (see step 1) and convert the columns, which contain a time (hh:mm:ss) to seconds (see step 2).

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

#%% 1 Read the data. The function "convert" will split the Data after ":"
def convert(s):
    h, m, s = map(int, s.split(':'))
    return pd.Timedelta(hours=h, minutes=m, seconds=s)

data=pd.read_csv('marathon-data_extended.csv',
                 converters={'split':convert, 'final':convert})
print(data.dtypes)

#%% 2 Apply the converter "convert" to transform the hh:mm:ss.
data['split_sec'] = data['split'] / np.timedelta64(1, 's')
data['final_sec'] = data['final'] / np.timedelta64(1, 's')
```

Since we already suspect that there are connections in certain variables, we form corresponding quotients (see step 3) as e.g. "size to weight" quotient.

```
#%% 3 Add more colums.
data['size_to_weight'] = data['size'] / data['weight']
print(data.head())
```

We receive the following dataset:

```
age,   gender,  split,      final,     nationality,  size,     weight,
 ↪  split_sec, final_sec, size_to_weight
33,     M,       01:05:38  02:08:51,      DE,       183.41,    84.0,
 ↪  3938.0,    7731.0, 2.183452
32,     M,       01:06:26  02:09:28,      DE,       178.61,    87.7,   3986.0,
 ↪  7768.0, 2.036602
31,     M,       01:06:49  02:10:42,      IT,       171.94,    82.2,   4009.0,
 ↪  7842.0, 2.091727
...
```

Next we will use a jointplot from the seaborn module (`sns.jointplot`, see step 4) and see the following:

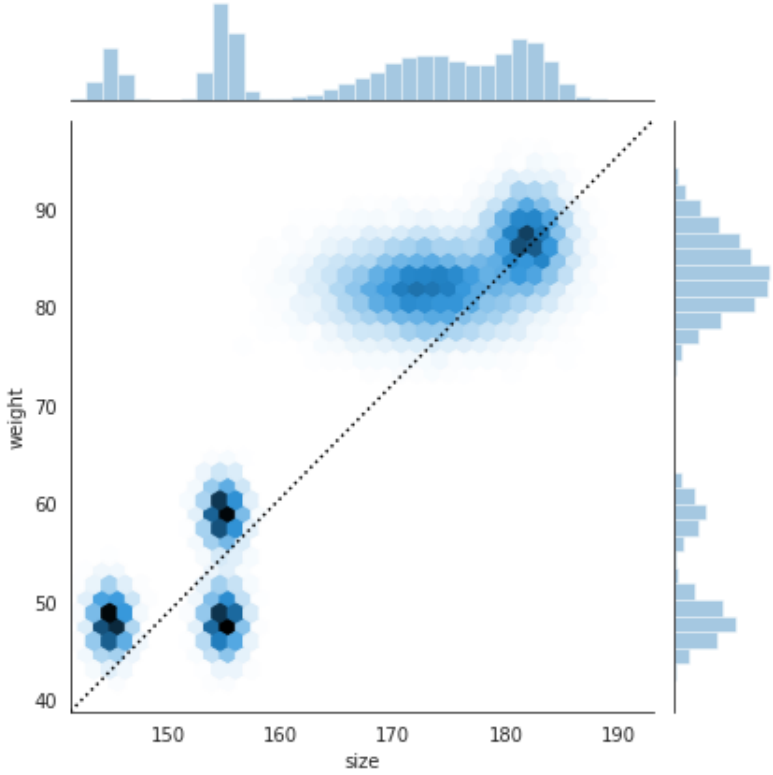


Figure 3.20: Marathon Example - Seaborn Jointplot

```
#%% 4 Joint-Plot with x=size and y=weight
with sns.axes_style('white'):
    g = sns.jointplot("size",
                      "weight",
                      data,
                      kind='hex')
```

```
    g.ax_joint.plot(np.linspace(min(data['size']),
                                max(data['size'])),
                    np.linspace(min(data['weight']),
                                max(data['weight'])),
                    ':k')
```

Obviously there are some dependencies in the data records. So we will dig a bit deeper and use the sns.distplot (Histogram, see step 5) which will show the following:

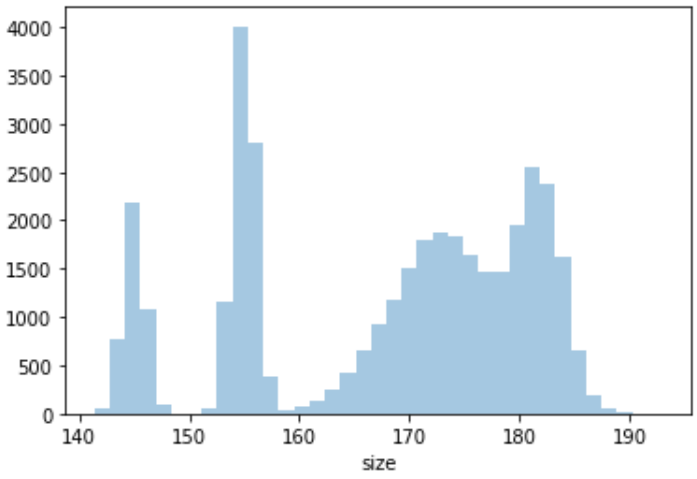


Figure 3.21: Marathon Example - Seaborn Histogram “size”

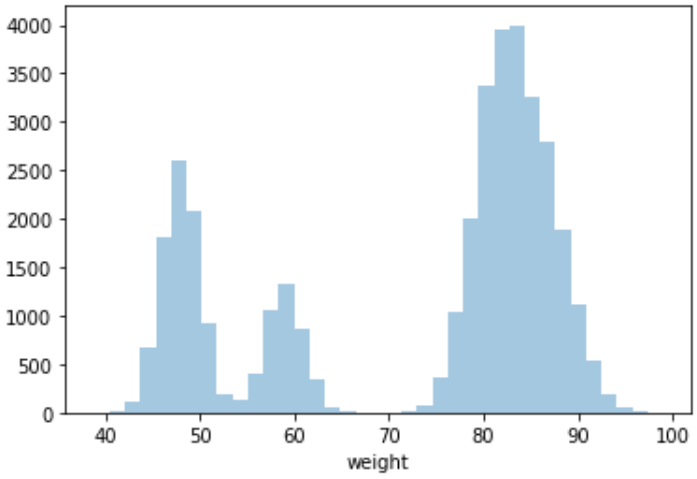


Figure 3.22: Marathon Example - Seaborn Histogram “weight”

```
#%% 5 Histogram for 'size' and 'weight' (distplot=Distribution Plot)
sns.distplot(data['size'], kde=False);
plt.show()
sns.distplot(data['weight'], kde=False)
```

As a next step we use the `sns.PairGrid` for examining if there are any correlations between the variables “nationality”, “size”, “final_sec” and “weight” (see step 6):

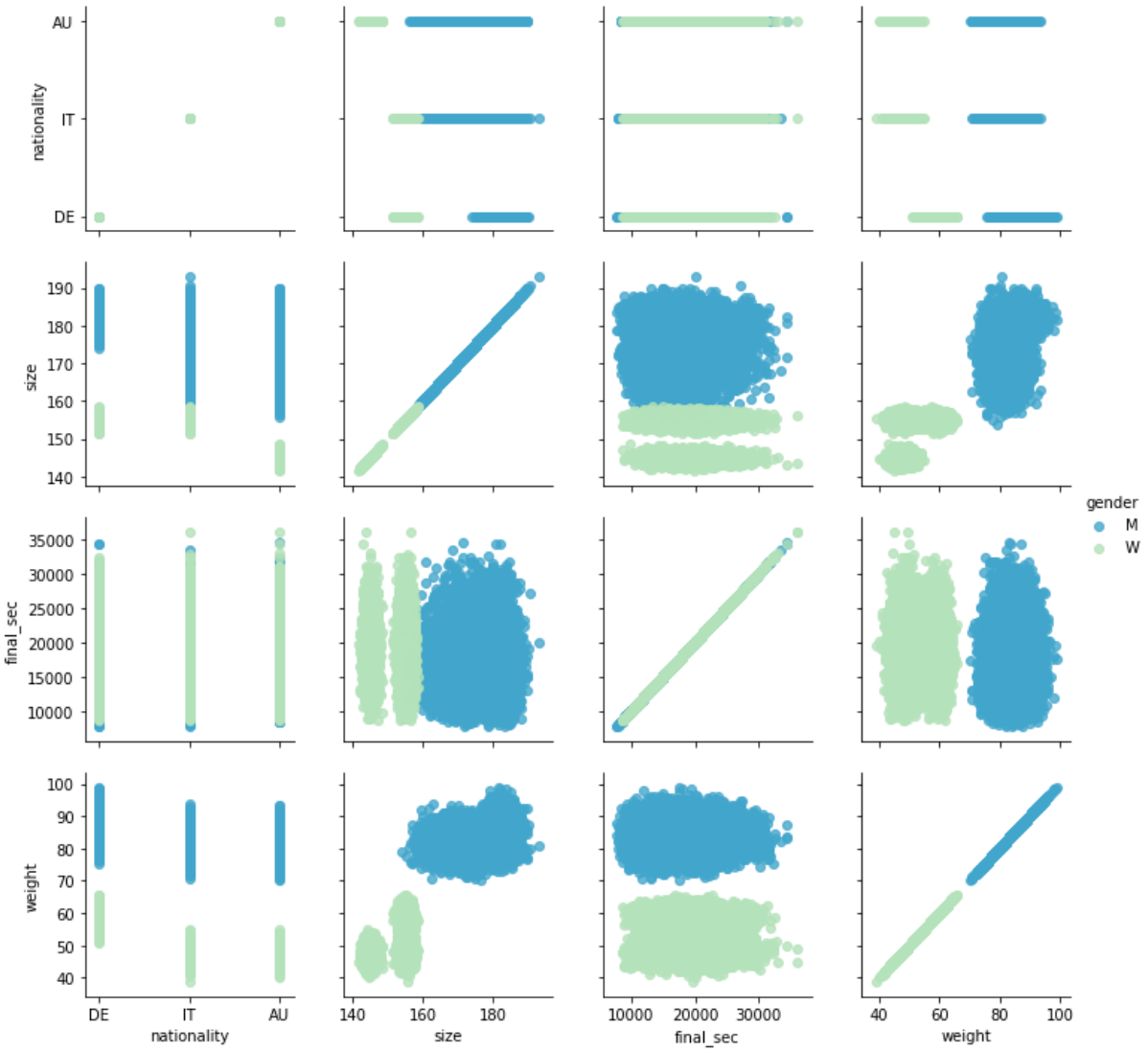


Figure 3.23: Marathon Example - Seaborn PairGrid

```
#%% 6 PairGrids with variables 'nationality', 'size', 'final_sec', 'weight'
 ↪  colors for gender (hue) is GreenBlue (GnBu)
g = sns.PairGrid(data,
```

```
                 vars=['nationality', 'size', 'final_sec', 'weight'],
                 hue='gender',
                 palette='GnBu_r')
g.map(plt.scatter, alpha=0.8)
g.add_legend();
```

This visualization already reveals a lot of information: German people have a higher weight (women as well as men). Austrian women are the smallest people and so on. Let's use the Kernel density functions next for the variable "size to weight" (step 7) and "size" (step 8):

```
#%% 7 KernelDensity (kde) for column "size_to_weight"
sns.kdeplot(data.size_to_weight[data.nationality=='DE'],
            label='Deutschland',
            shade=True)
sns.kdeplot(data.size_to_weight[data.nationality=='AU'],
            label='Österreich',
            shade=True)
plt.xlabel('size_to_weight');
plt.show()
```

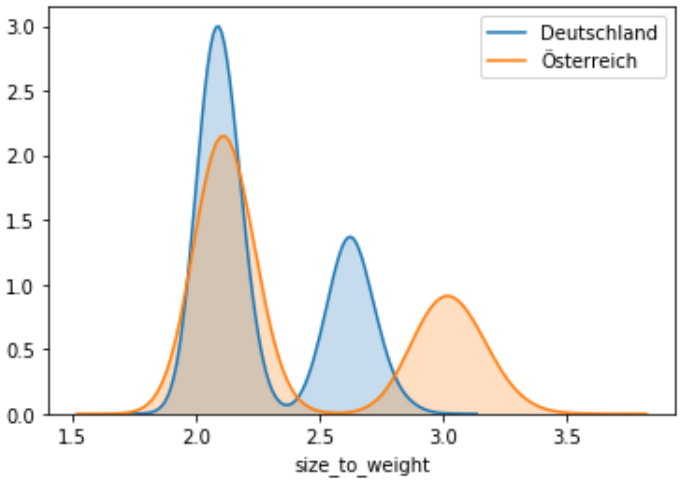


Figure 3.24: Marathon Example - Seaborn Kernel Density "size-to-weight"

```
#%% 8 KernelDensity (kde) for column "size"
sns.kdeplot(data.weight[data.nationality=='DE'],
            label='Deutschland',
            shade=True)
sns.kdeplot(data.weight[data.nationality=='AU'],
            label='Österreich',
```

```
            shade=True)
plt.xlabel('size');
```

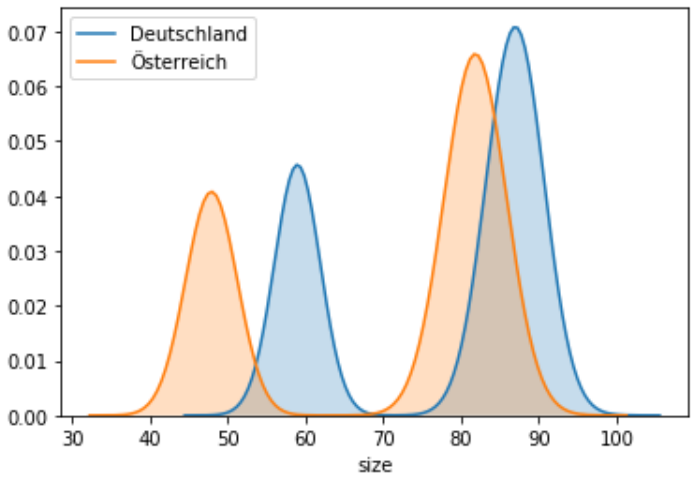


Figure 3.25: Marathon Example - Seaborn Kernel Density "size"

Now it would interesting to see some regression plots (step 9). In fig. 3.26 for men and in fig. 3.27 for women:

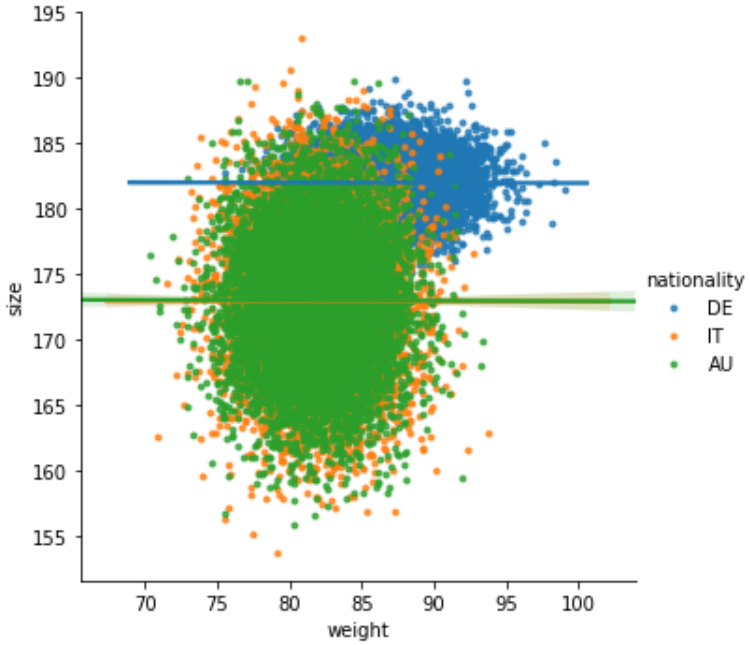


Figure 3.26: Marathon Example - Seaborn Regression Plots "men"

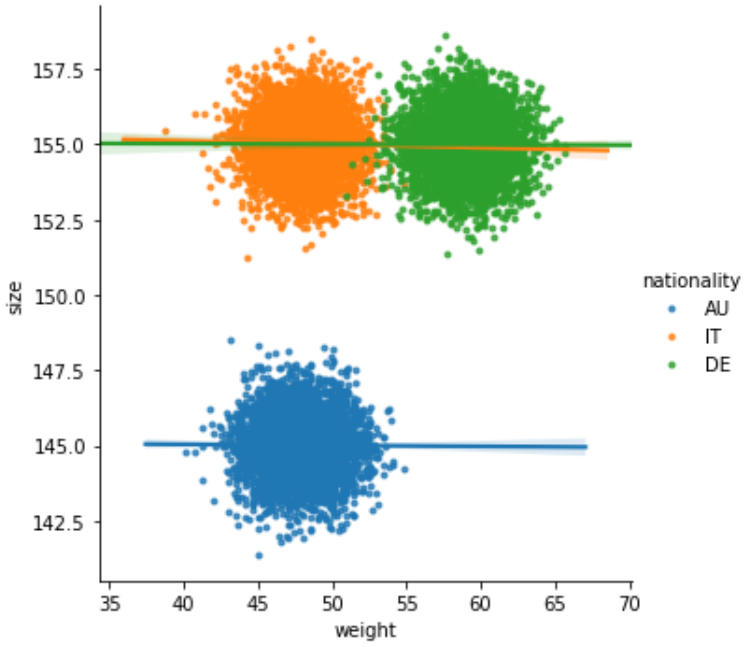


Figure 3.27: Marathon Example - Seaborn Regression Plots “women”

```
#%% 9 Regression Plot for "weight" and "size"
h = sns.lmplot('weight', 'size', hue='nationality',
               data=data[data.gender=="M"],
               markers=".")
h = sns.lmplot('weight', 'size', hue='nationality',
               data=data[data.gender=="W"],
               markers=".")
```

Here again we see, that Austrian women are smaller (see dots in blue). And finally we will use the Seaborn-Violinplots, `sns.violinplos` (step 10 and 11), which finally reveals all details, which have been hidden in this dataset:

```
#%% 10 Violinplot using "size" and "nationality"
men = (data.gender == 'M')
women = (data.gender == 'W')
with sns.axes_style(style=None):
    sns.violinplot("size", "nationality",
                   hue="gender",
                   data=data,
                   split=True,
                   inner="quartile",
                   palette=["lightblue", "lightpink"]);
plt.show()
```

For example: we see in fig. 3.28, that German men are taller (180 cm) with a more narrow distribution (standard deviation), than Italian and Austrian men. The distribution of Italian men and Austrian men seems to be be identical (normal distribution with the same mean, but a bigger standard deviation). In contrast: Italian women are taller (155 cm) than Austrian women (145 cm).

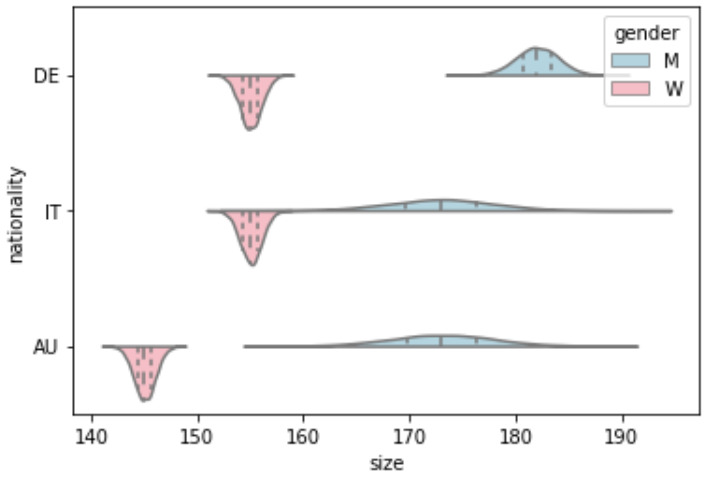


Figure 3.28: Marathon Example - Seaborn Violinplot “size”

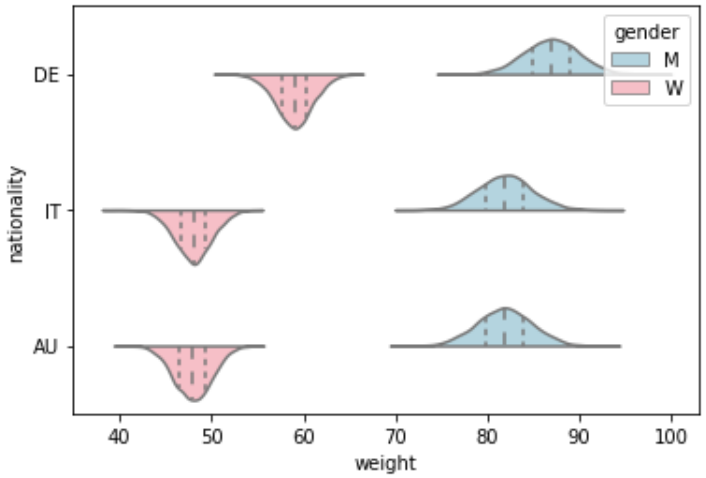


Figure 3.29: Marathon Example - Seaborn Violinplot “weight”

```
#%% 11 Violinplot using "weight" and "nationality"
with sns.axes_style(style=None):
    sns.violinplot("weight",
                   "nationality",
```

```
                hue="gender",
                data=data,
                split=True,
                inner="quartile",
                palette=["lightblue", "lightpink"]);
```

In fig. 3.29 we see, that the weight of Italian women and Austrian women seem (in contrast to their size) to be identically distributed with about 48 kg in average. German women are heavier with about 58 kg in average. German men are the heaviest (with about 88 kg in average). A deeper examination of the distributions would need some background in mathematics, which we won´t do here.

Obviously the underlying data has been treated artificially by me (I apologize for any negative sentiment I might have pushed to Austrian, Italian or German people).

The example above shows, how easy visualization techniques can be and how powerful Python is (combined with the libraries seaborn and matplotlib). Imagine doing the same in Excel: it would take a lot longer. A few lines of code are sufficient for revealing a lot of hidden information of a dataset. Without knowing too much about mathematics or statistics, the systematics in the underlying data are found. The same logic applies to any kind of data your company may hold in their hands (invoices, number of contracts, overtime hours, ...).

Data Scientist often forget, that all their visualizations (and also model), which they have built, need to be used by someone, who is probably not as skilled in all these technical requirements! Therefore it is important to find a solution, which is **easy to deploy** and **easy to use** for everyone (as well on a computer as also on a mobile phone), **stable** and **quickly customizable**. In Section sec. 3.3 I will show you how to share this data with an "Data App".

For the "Marathon runtimes" example I also wrote a testing file and included Travis[16] and Codecov[17]. Travis and Codecov provide small icons, like the following:

Figure 3.30: Build Status passing

The advantage of doing this is: when I share my python code or Jupyter-Notebooks on code-sharing platforms, like GitHub[18] other people will know, that my code has been tested and does not contain bugs. If you plan to share your code frequently then I recommend to have a look at least into Travis and Codecov (and there are a lot more services like these two, which might also be interesting).

[16] Travis, https://travis-ci.com/
[17] Codecov, https://codecov.io/
[18] GitHub, https://github.com

On my GitHub profile you can find my Python-File for this example[19]

3.1.4 Pedestrians during the first Corona-Lockdown

On Saturday 21.03.2020 (which is end of week number 12) exit locks had been implemented in order to protect the people from the Corona virus. The company Hystreet[20], provides statistics about the number of pedestrians walking in inner cities. I expected, that due to the implemented exit lock the number of pedestrians in the inner cities should decrease from 21.03.2020 onwards and this was, what I wanted to visualize.

Hystreet offers free downloadable csv files for private use. I downloaded the statistics for Ulm, Münsterplatz, Munich and Augsburg. The following plot shows the number of pedestrians walking on Ulm, Münsterplatz from 21.03.2020 onwards:

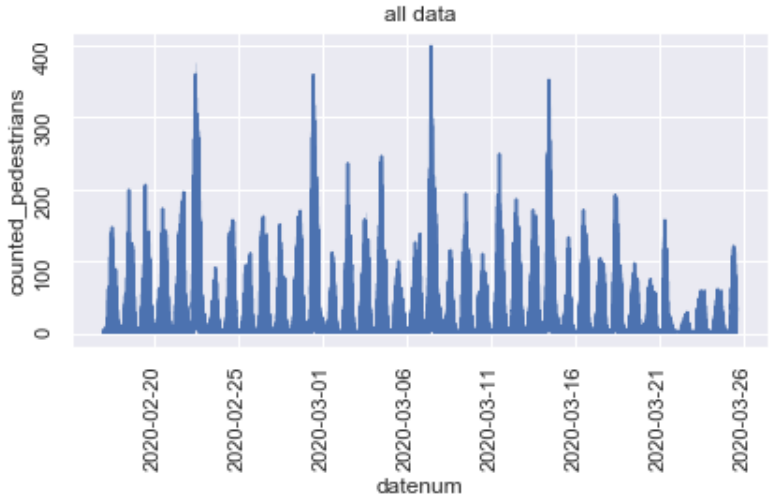


Figure 3.31: Pedestrians in Corona-Lockdown - Frequency

We can see the decrease, but it is not clear, which week-days are meant with the spikes. In order to make this a bit clearer, I wanted barplots with colour for each work-day.

[19] Marathon Python-Files, https://github.com/AndreasTraut/Visualization-of-Data-with-Python/blob/main/Example_Marathon_extended.py

[20] Hystreet, https://hystreet.com

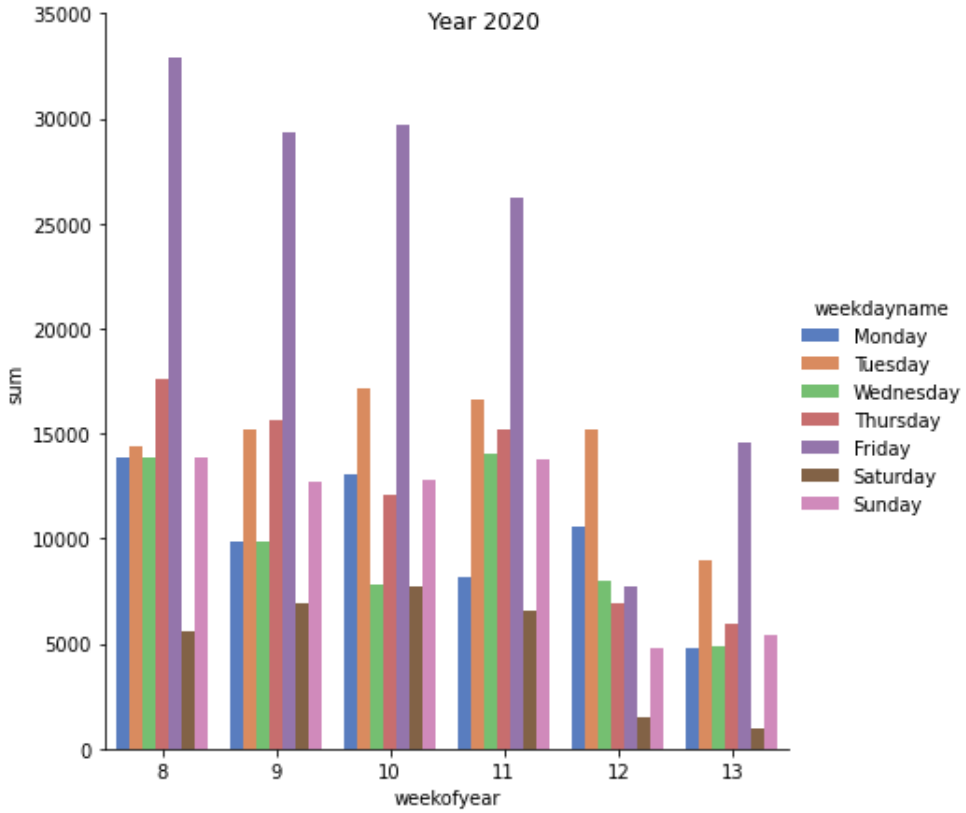


Figure 3.32: Pedestrians in Corona-Lockdown - Barplot “Ulm” Year 2020

The numbers for Augsburg, Annastraße and Munich, Maximilianstraße in barplots looked similar and you can download them from my GitHub repository.

In 2021, one year after I made this analysis, I wanted to see what changed. Therefore my aim was to compare the week number 8, 9,..,13 of year 2020 to the ones of year 2021. But first I will show you how to create plot fig. 3.32. I needed the csv files from Hystreet and read them in python:

```
import pandas as pd
from matplotlib import pyplot, dates
import seaborn as sns
import glob
import numpy as np

#%% read all files
all_files = glob.glob("*.csv")
li = []
for filename in all_files:
    df = pd.read_csv(filename,
                     index_col=None,
```

```
                    header=0,
                    sep=';')
    li.append(df)
df = pd.concat(li, axis=0, ignore_index=True)
```

The pandas dataframe has the following format (use `df.info()`):

```
Data columns (total 7 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   location              100 non-null    object
 1   time of measurement   100 non-null    object
 2   weekday               100 non-null    object
 3   pedestrians count     100 non-null    int64
 4   temperature in °c     99 non-null     float64
 5   weather condition     99 non-null     object
 6   incidents             0 non-null      float64
dtypes: float64(2), int64(1), object(4)
```

In a next step I needed to convert the “time of measurement” to a date and extract the week-of-year and the week-day. We saw this converting issues already in previous examples.

```
df['datenum'] = dates.datestr2num(df['time of measurement'])
df['date'] = df['time of measurement'].apply(pd.to_datetime,
 ↪  errors='coerce', utc=True)
df['weekofyear'] =  df['date'].dt.isocalendar().week
df['weekday'] = pd.DatetimeIndex(df['date']).weekday #Monday=0
df['weekdayname'] = pd.DatetimeIndex(df['date']).day_name()
df['year'] = pd.DatetimeIndex(df['date']).year
df['pedestrianscount'] = df['pedestrians count']
df = df[(df['weekofyear']>=8) & (df['weekofyear']<=13)]
```

Now I am ready to group and sum the data as follows:

```
df.groupby(['year', 'weekday','weekdayname'])['pedestrians count'].sum()
```

```
year  weekday  weekdayname
2020  0        Monday          60360
      1        Tuesday         87535
      2        Wednesday       58441
```

```
      3        Thursday        73327
      4        Friday         140543
      5        Saturday        29406
      6        Sunday          63435
2021  0        Monday          54814
      1        Tuesday         73365
      2        Wednesday       55381
      3        Thursday        54211
      4        Friday          99952
      5        Saturday        30120
      6        Sunday          44725
Name: pedestrians count, dtype: int64
```

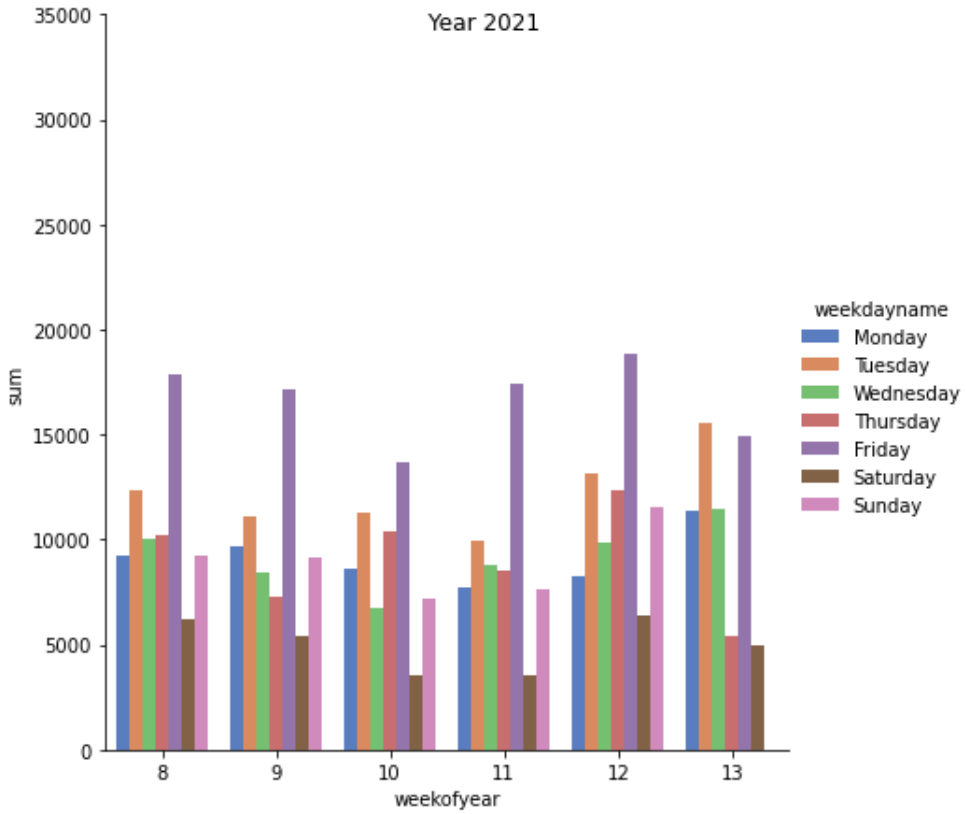


Figure 3.33: Pedestrians in Corona-Lockdown - Barplot “Ulm” Year 2021

We can also see from the numbers, that the sum of pedestrians decreased. With the following python code I created the plot in fig. 3.32 and by replacing `myYear = 2021` also the plot in fig. 3.33 .

```
myYear = 2020
df_grouped=(df.groupby(['year', 'weekofyear', 'weekdayname'],
 ↪  as_index=False).pedestrianscount.
            agg({'pedestrianscount': lambda x: list(x), 'sum': 'sum'}))
g = sns.catplot(x='weekofyear', y='sum', hue='weekdayname',
            hue_order=['Monday', 'Tuesday', 'Wednesday', 'Thursday',
 ↪  'Friday', 'Saturday', 'Sunday'],
            data=df_grouped[df_grouped['year']==myYear],
            height=6, kind='bar', palette='muted')

g.set(ylim=(0,35000))
g.fig.suptitle("Year {}".format(myYear))
```

Compare these two barplots (see fig. 3.32 and fig. 3.33) and examine how the numbers of pedestrians in inner cities decreased. I am not sure, if people are willing to go back for shopping in the inner cities after the Corona crisis or if they continue to use the online-shopping offers from Amazon. But this is not what I am working on here in my example.

Next I wanted to compare the changes between 2020 and 2021 on a weekly basis. Therefore I selected a week `myWeek=10` and created some more interesting barplots:

```
myWeek = 10
myYear = 2021
a = df_grouped[(df_grouped['year']==myYear) &
 ↪  (df_grouped['weekofyear']==myWeek)]['sum']
b = df_grouped[(df_grouped['year']==myYear-1) &
 ↪  (df_grouped['weekofyear']==myWeek)]['sum']

index = np.arange(7)
pltpera= pyplot.bar(index, a, width=0.3, label=myYear, color='red')
pltperb = pyplot.bar(index - 0.3, b, width=0.3, label=myYear-1,
 ↪  color='peachpuff')
pyplot.title('Week {}'.format(myWeek, myYear, myYear-1))
pyplot.xticks(index, ('Monday', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat',
 ↪  'Sunday'))
pyplot.legend()
pyplot.show()
```

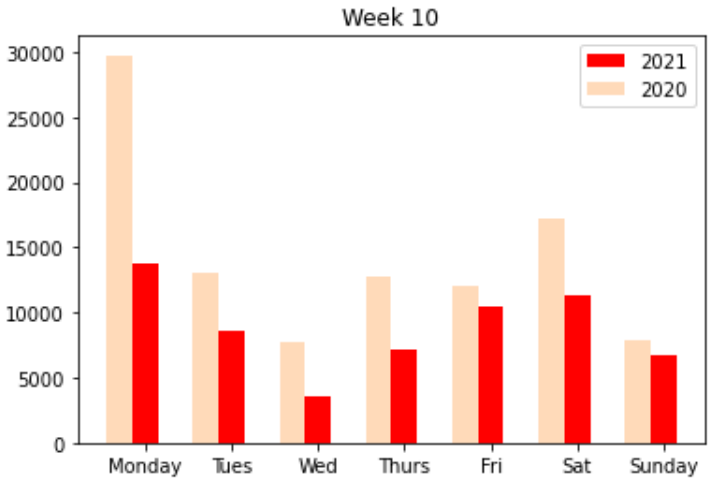


Figure 3.34: Pedestrians in Corona-Lockdown - Barplot “Ulm” Week 10

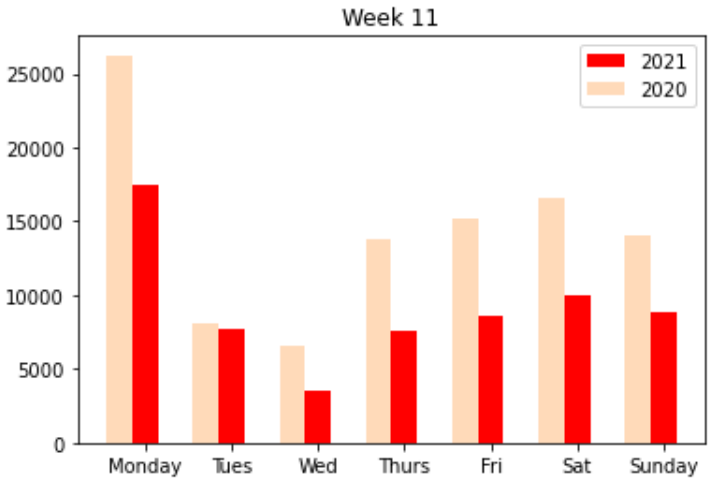


Figure 3.35: Pedestrians in Corona-Lockdown - Barplot “Ulm” Week 11

On my GitHub profile you can find my Python-File for this example[21]

3.1.5 Station Elevators of Deutsche Bahn: API Example

The visualization is sometimes a bit difficult, because the dataset is not yet available in the form you need to have them. Until now I worked with csv files, which is the easiest way to use a dataset. If you work with APIs instead of csv files, the data preparation and visualization will be a bit different. An API is an “Application Programming Interface”. It defines the interactions with a software compo-

[21] Pedestrians during Corona-Lockdown Jupyter Notebook, https://github.com/AndreasTraut/Visualization-of-Data-with-Python/blob/main/Pedestrians/Pedestrians.ipynb

nent and the kind of calls or requests, that can be made. An API allows the users to use the interface independently of the implementation[22].

I wanted to know which elevators from Deutsche Bahn are currently working and which ones are damaged. I knew, that I could extract this information from the Deutsche Bahn API "FaSta"[23], but I would need to implement some Python-Code (`.py` File) to extract the information I needed:

- the "station number" (the station number for Ulm it is 6323)
- "equipment number" (the number of the elevator) and
- the "status" ("available"/"monitoring disrupted").

I subscribed to the Deutsche Bahn FaSta API (see fig. 3.36) and received a consumer key and consumer secret for a production environment and a sandbox environment (=testing environment). I am not allowed to share my keys here, but you can create some yourself, if you want.

You need to insert your tokens in order to get access to the FaSta-API with the following python code:

```
myToken = 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx' #Produktion
myUrl = 'https://api.deutschebahn.com/fasta/v2'
head = {'Authorization': 'Bearer {}'.format(myToken)}
response = requests.get(myUrl, headers=head)
```

Each elevator has an equipment number and the four elevators in Ulm these are 10500702, 10500703, 10500704 and 10499292. I didn't find an documentation for these numbers and I found them by trial-and-error. Maybe the Deutsche Bahn didn't want the transparency over these numbers in order to hide the number of damaged elevators a bit.

To get access to the equipment number 10500702, use the following code:

```
myEquipmentnumber = myUrl + "/facilities/10500702"
response = requests.get(myEquipmentnumber, headers=head)
print("Response status code is: ", response.status_code)
result = response.content
```

[22] Application Programming Interface (API), HubSpire, https://www.hubspire.com/resources/general/application-programming-interface/

[23] Deutsche Bahn API FaSta, https://developer.deutschebahn.com/store/apis/info?name=FaSta-Station_Facilities_Status&version=v2&provider=DBOpenData

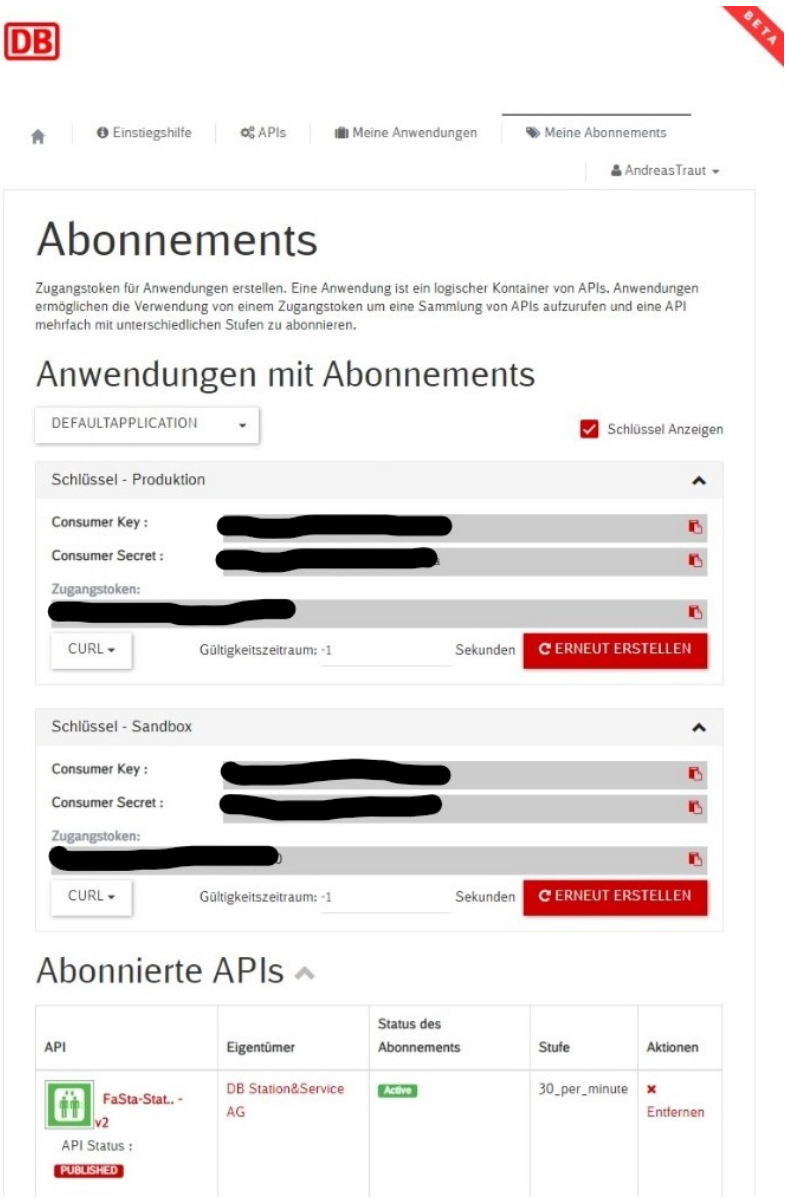


Figure 3.36: Deutsche Bahn - Account for FaSta API

You will receive the response status code 200 if everything works. Now let's read some more details from this equipment:

```
result_Eq = response.content
data = json.loads(result_Eq)
print("stationnumber: {}".format(data["stationnumber"]))
print("equipmentnumber: {}".format(data["equipmentnumber"]) )
print("description: {}".format(data["description"]) )
print("state: {}".format(data["stateExplanation"]))
```

```
stationnumber: 6323
equipmentnumber: 10500702
description: BAHNSTEIG 2/3 SÜD
state: available
```

This means, that the elevator with equipment number 10500702 at station number 6323 (which is Ulm) is available. Now I want to have a closer look into all elevators (including their GPS coordinates) at the station Ulm. After having understood and having extracted these meta data (station number, equipment number) I was able to visualize them: as you can see in the screenshot fig. 3.37, when I handed over a station number (e.g. 6323 for Ulm) to the Deutsche Bahn API with and received the number of elevators (here 4) and also the longitudes X=9.98278 and latitudes Y=48.39846 of this elevator.

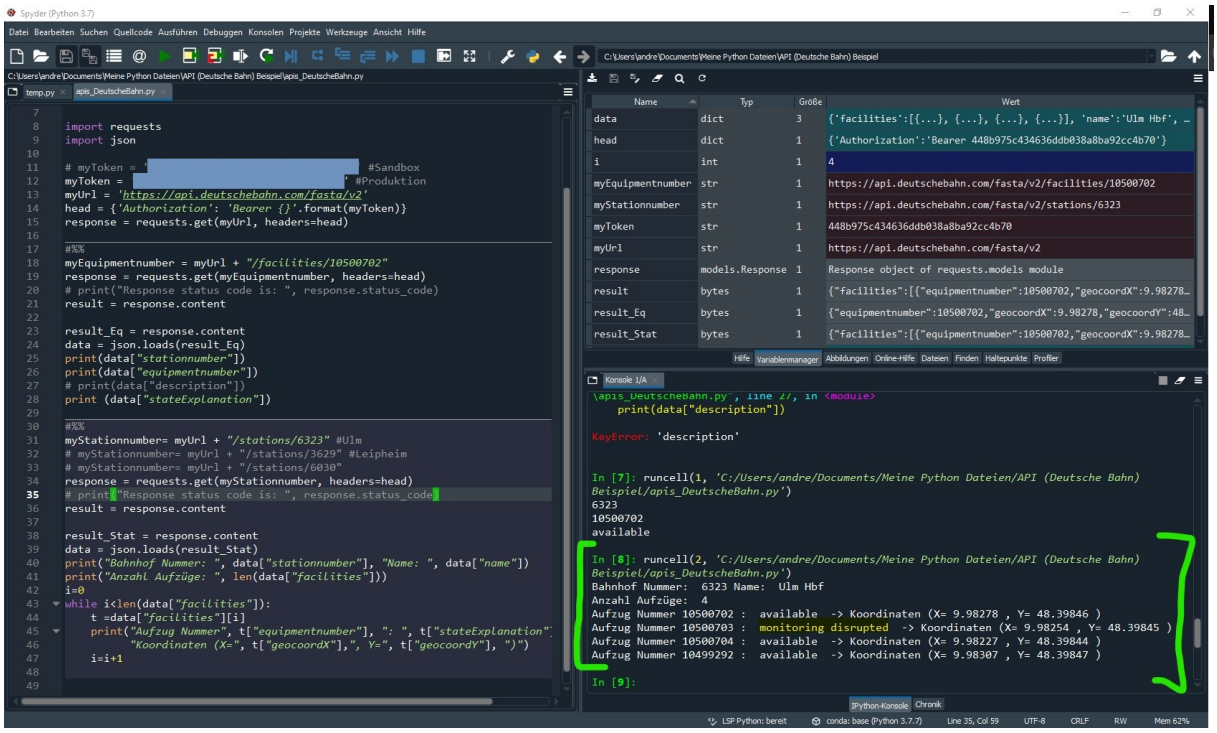


Figure 3.37: Deutsche Bahn - Elevators

```
myStationnumber= myUrl + "/stations/6323" #Ulm
response = requests.get(myStationnumber, headers=head)
result = response.content
result_Stat = response.content
data = json.loads(result_Stat)
print("Bahnhof Nummer: ", data["stationnumber"], "Name: ", data["name"])
print("Anzahl Aufzüge: ", len(data["facilities"]))
i=0
```

```
while i<len(data["facilities"]):
    t =data["facilities"][i]
    print("Aufzug Nummer", t["equipmentnumber"], ": ",
     ↪  t["stateExplanation"]," -> "
          "Koordinaten (X=", t["geocoordX"],", Y=", t["geocoordY"], ")")
    i=i+1
```

```
Bahnhof Nummer:  6323 Name:  Ulm Hbf
Anzahl Aufzüge:  4
Aufzug Nummer 10500702 :  available  -> Koordinaten (X= 9.98278 , Y=
 ↪  48.39846 )
Aufzug Nummer 10500703 :  available  -> Koordinaten (X= 9.98254 , Y=
 ↪  48.39845 )
Aufzug Nummer 10500704 :  available  -> Koordinaten (X= 9.98227 , Y=
 ↪  48.39844 )
Aufzug Nummer 10499292 :  available  -> Koordinaten (X= 9.98307 , Y=
 ↪  48.39847 )
```

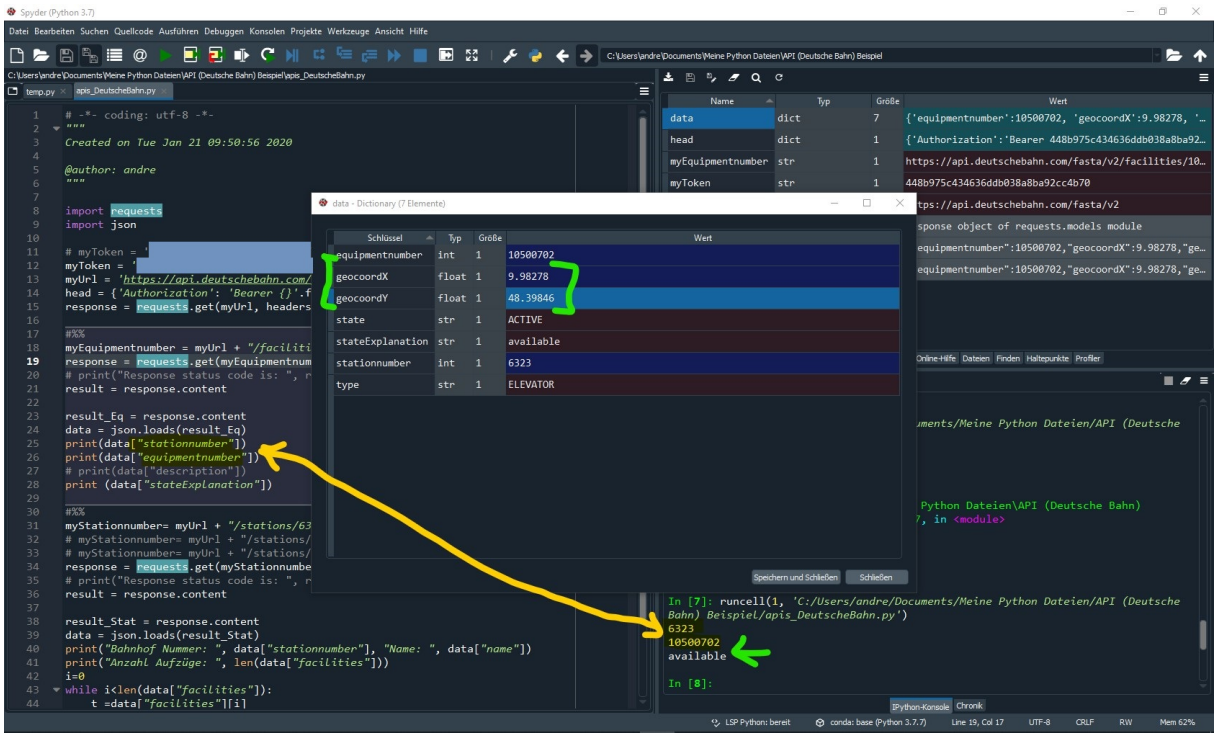


Figure 3.38: Deutsche Bahn - Elevators and Geo Information

Taking these and using for example GPS-Coordinates[24] you can easily visualize these longitudes and

[24] GPS-Coordinates, www.gps-coordinates.net

latitudes as shown in fig. 3.39:

Another possibility would have been to use Geopy[25] and type the following code (please verify the ToS for using this service on your own: it's limited!):

```
from geopy.geocoders import Nominatim
geolocator = Nominatim(user_agent="http")
location = geolocator.reverse("48.39846, 9.98278")
print(location.address)
```

The result would have been

`"Steig 2 + 3, Bahnhofplatz, Fischerviertel, Weststadt, Ulm, Baden-Württemberg, 89073, Deutschland"`

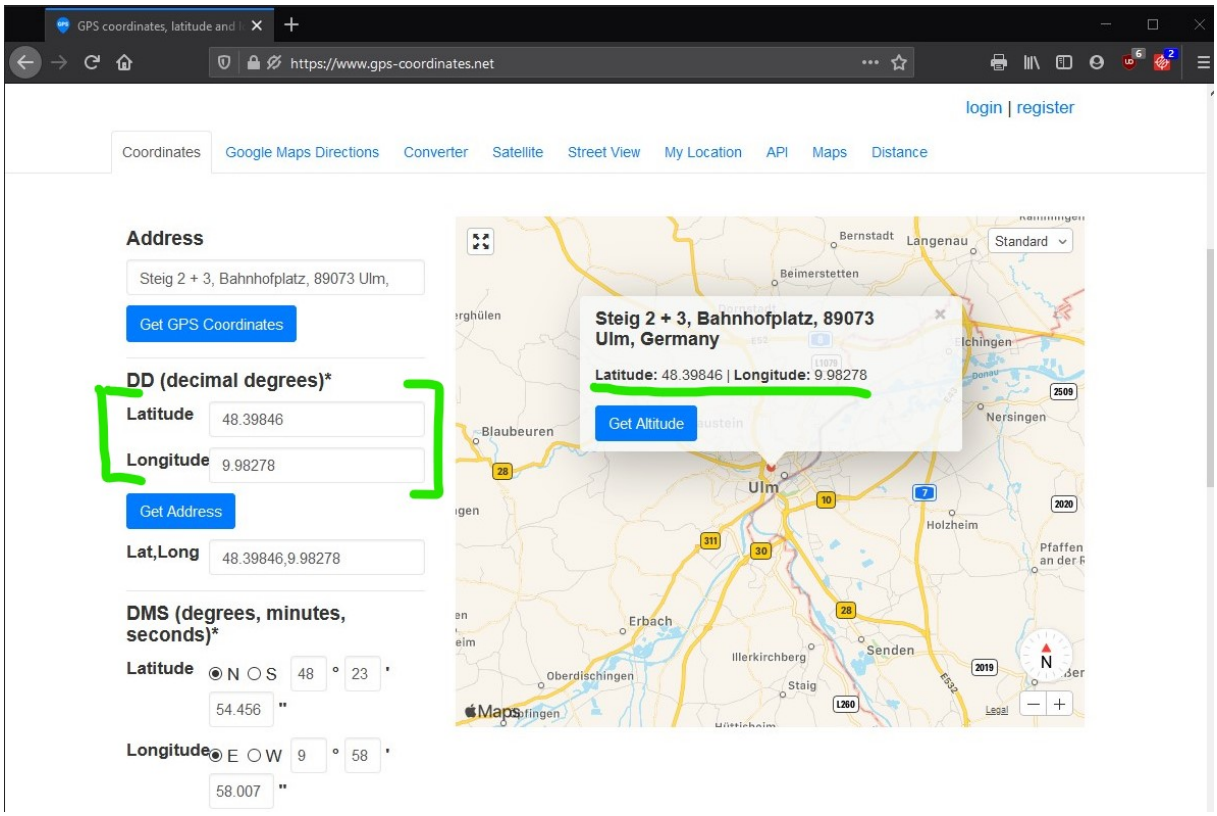


Figure 3.39: Deutsche Bahn - Latitude and Longitude

Building a longer history (not only one extraction of data, but many over a longer period) to show how many elevators are damaged in Germany during one year would also be possible. It would mean, that I have to loop each day over all "station numbers", then over all "equipment numbers" and store

[25] Geopy, https://github.com/geopy/geopy

the number of status=damaged. After one year I would have the history of damaged elevators. This is what an journalist did in order to write an article (which was available on www.br.de, but unfortunately disappeared).

3.1.6 Interactive Pivots

Another nice possibility to visualize pandas data frames is by using the "*pivottablejs*"[26] implementation. I loved how easy this implementation is to use (basically one line of code) and how nice it creates pivot tables in a Browser (like Firefox, Chrome,...). I think everyone knows from Excel already, what a pivot table is and possibly loves its high intractability: grouping and aggregating is very easy. Data Scientist often forget, that their models and visualizations need to be used by people, who are not as familiar with the technical concepts as a Data Scientist. Therefore my recommendation for a Data Scientist is to always think about how the data and model should be transferred to the users: "*pivottablejs*" is one solution, which fulfils this criteria. I will describe another solution ("*Streamlit App*", which is a lot more powerful than "*pivottablejs*") in sec. 3.3.

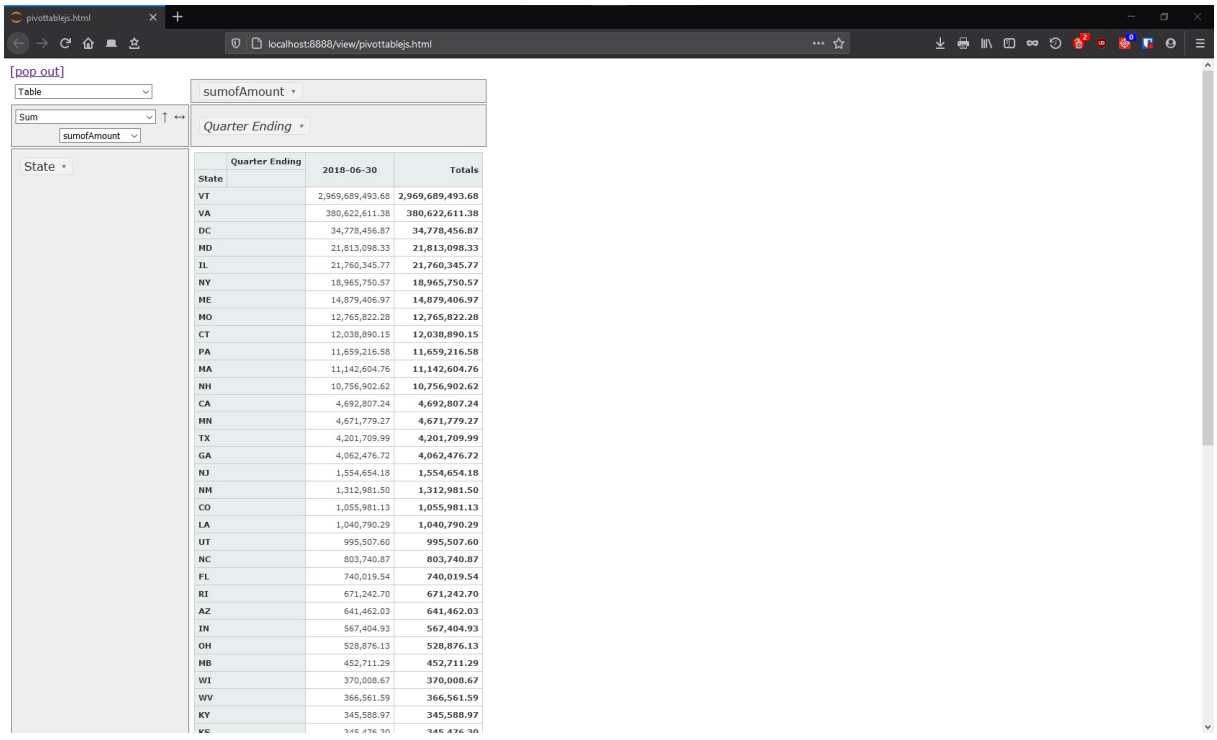


Figure 3.40: Interactive Pivots - Grouping and aggregating

```
# Install: !pip install pivottablejs
import pivottablejs
```

[26] Pivottable, https://pivottable.js.org/examples/

```
pivottablejs.pivot_ui(pandas_df)
```

Now you can drag and drop the columns and also use the filter select boxes as shown in fig. 3.40. And creating graphs is also possible (see fig. 3.41.

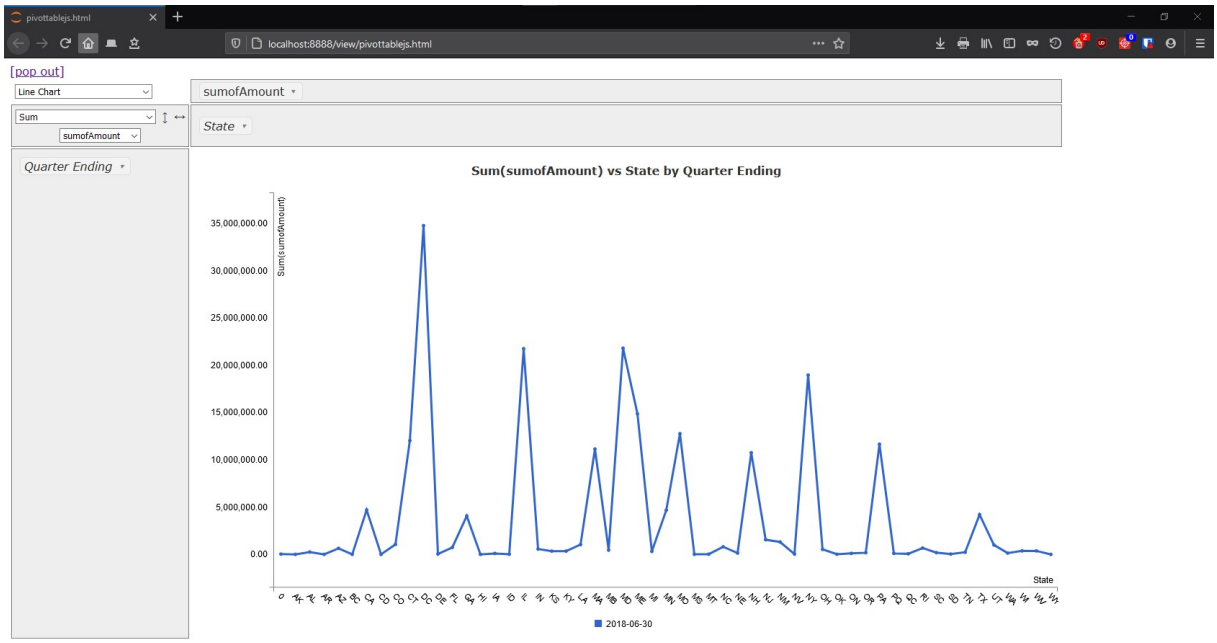


Figure 3.41: Interactive Pivots - Graph

3.2 Introduction to Big Data Visualization Techniques

3.2.1 Basic Problems in Big Data Visualizations

I will provide a short introduction into how Big Data visualization needs to be approached. Visualization of Big Data datasets can be challenging. There are different problems, which will become relevant if you want to do Big Data visualizations:

Can you load all the data into your memory? Probably not. Can you transfer the data from your database over your network (intranet / internet) to your terminal (a computer or mobile phone)? You might slow down the entire network in your company.

What about if you need slide controllers, filters, selections boxes for a user who wants to interact with the data and who wants to produce different visualizations? Then you would create a lot of back and forth transactions between the data base and the terminal. What would you do then?

What about the screen-resolution of your terminal? The screen resolution of a mobile phone is limited to some thousand pixels: visualizing Big Data on a screen with low resolution would end in an ugly black blob on the screen. A computer monitor might show a bit more detail, but can really assume that everyone has a monitor with 4K resolution?

3.2.2 Vermont Payments Example

First I downloaded the list of all state of Vermont payments to vendors [27] (Open Data Commons License), which is a 298 MB huge csv file with 1.6 million lines (exactly 1'648'466 lines). As said: visualization of such huge datasets can be difficult with common tools like Excel. For this example you can use Power-Query[28] or something similar. If you are already familiar with this tool, then you might solve this problem immediately and without additional efforts. But you loose for example the advantages of machine-learning algorithms from the Apache Spark Machine Learning Library[29], which is only one big difference to be mentioned.

First let's get ready for the Big Data environment and open my machine-learning Docker container. If you don't know how, then have a look into sec. 4.4.1.1.

```
import os
print("APACHE_SPARK_VERSION: ",
      os.environ["APACHE_SPARK_VERSION"])
print("HADOOP_VERSION: ",
      os.environ["HADOOP_VERSION"])
```

I am on `APACHE_SPARK_VERSION: 3.0.0` and `HADOOP_VERSION: 3.2`. For initializing spark we need a `SparkContext`and a `SparkSession`:

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
sc = pyspark.SparkContext(appName='Spark Modelling Context')
spark = SparkSession.builder \
        .appName('Spark Modelling Session') \
        .config('spark.executor.memory','5g') \
        .config('spark.executor.cores','4') \
        .getOrCreate()
```

[27] Vermont payments to vendors, https://data.vermont.gov/Finance/Vermont-Vendor-Payments/786x-sbp3

[28] PowerQuery, https://support.microsoft.com/de-de/office/einf%C3%BChrung-in-microsoft-power-query-f%C3%BCr-excel-6e92e2f4-2079-4e1f-bad5-89f6269cd605

[29] Spark Machine Learning Library, https://spark.apache.org/mllib/

Next step is to read the huge csv file:

```
import os
datapath = os.environ['PWD']
filename = datapath + "/data/Vermont_Vendor_Payments.csv"
#read in data from csv
data = spark.read.csv(path=filename, sep=',', encoding='utf-8', header=True,
 ↪  inferSchema=True)
```

First we want more details about the dataset:

```
data.describe().show(truncate=False, n=1, vertical=True)
```

```
-RECORD 0--------------------
 summary             | count
 Quarter Ending      | 1648418
 Department          | 1648418
 UnitNo              | 1648418
 Vendor Number       | 1648418
 Vendor              | 1648418
 City                | 906137
 State               | 1648418
 DeptID Description  | 1647881
 DeptID              | 1648418
 Amount              | 1648418
 Account             | 1648418
 AcctNo              | 1648418
 Fund Description    | 1648416
 Fund                | 1648417
only showing top 1 row
```

Once you are into the Apache Spark environment you can easily aggregate, sort, group by what ever you want. Take for example the columns “Department” and “Amount” (it should be obvious, what is in these columns, I guess). Then this line of code will show you the sum of column “Amount” grouped per department (sorted descending):

```
data.groupBy('Department') \
    .agg(F.sum('Amount') \
         .cast('decimal(20,2)') \
         .alias('sumofAmount')) \
    .sort('sumofAmount', ascending=False) \
    .show(truncate=False)
```

```
+------------------------------+--------------+
|Department                    |sumofAmount   |
+------------------------------+--------------+
|Buildings & Gen Serv-Prop     |254664862145.07|
|Vermont Health Access         |7316059819.96 |
|Natural Res Central Office    |6115935633.73 |
|Education Agency              |5156573496.88 |
|Education                     |3166972698.64 |
|Transportation Agency         |2795155337.06 |
|Department of VT Health Access|2393175142.17 |
|Finance & Management          |2331413298.90 |
|Agency of Transportation      |1920833560.49 |
|Children and Families         |1850543830.45 |
|null                          |1501137106.31 |
|Office of VT Health Access    |1466094349.44 |
|Disabilities Aging Ind. Living|1371142726.00 |
|Children and Family Services  |1281368750.70 |
|Mental Health                 |1259316412.60 |
|Human Resources-Prop          |1195825754.22 |
|Human Resources-Gov'tal       |1191172666.22 |
|Treasurer's Office            |882007501.28  |
|Health                        |853538376.05  |
|Aging and Independent Living  |789393840.12  |
+------------------------------+--------------+
only showing top 20 rows
```

The following line of code will group by “Quarter Endings”, aggregate the “Amounts” (sum) and sort the result ascending:

```
spark_df = data.groupBy('Quarter Ending') \
               .agg(F.sum('Amount') \
               .cast('decimal(20,2)') \
               .alias('sumofAmount')) \
               .sort('Quarter Ending', ascending=True)
spark_df.show(truncate=False)
```

```
+--------------+--------------+
|Quarter Ending|sumofAmount   |
+--------------+--------------+
|03/31/2010    |28598462713.41|
|03/31/2011    |15900537688.65|
|03/31/2012    |17106963001.97|
```

```
|03/31/2013    |7799135775.53 |
|03/31/2014    |2078065068.29 |
|03/31/2015    |3191476205.73 |
|03/31/2016    |2135004787.94 |
|03/31/2017    |3281966008.12 |
|03/31/2018    |4471045973.70 |
|03/31/2019    |3294727802.75 |
|06/30/2010    |14937551502.74|
|06/30/2011    |11468901690.66|
|06/30/2012    |18502465970.34|
|06/30/2013    |5774960317.50 |
|06/30/2014    |4655109691.64 |
|06/30/2015    |5824948519.54 |
|06/30/2016    |3499716477.98 |
|06/30/2017    |4680238788.00 |
|06/30/2018    |3552314458.12 |
|06/30/2019    |3598682370.54 |
+--------------+--------------+
only showing top 20 rows
```

If you want to focus on the top 6 states (which are VT, MA, NH, PA, VA, GA), then the following code will filter on these states, aggregate the "Amount" (sum) and sort by "Quarter Ending" (ascending):

```
spark_df = data.filter((F.col('State') == 'VT') |
                       (F.col('State') == 'MA') |
                       (F.col('State') == 'NH') |
                       (F.col('State') == 'PA') |
                       (F.col('State') == 'VA') |
                       (F.col('State') == 'GA')) \
                .groupBy('State', 'Quarter Ending') \
                .agg(F.sum('Amount') \
                .cast('decimal(20,2)') \
                .alias('sumofAmount')) \
                .sort('Quarter Ending', ascending=True)
spark_df.show()
```

```
+-----+--------------+--------------+
|State|Quarter Ending|   sumofAmount|
+-----+--------------+--------------+
|   VT|    03/31/2010|26122164537.74|
|   PA|    03/31/2010|    5857825.21|
|   NH|    03/31/2010|    6802959.20|
```

```
|   MA|    03/31/2010| 2328199767.23|
|   GA|    03/31/2010|    5204426.97|
|   VA|    03/31/2010|    1392512.09|
|   GA|    03/31/2011|    6103226.21|
|   VA|    03/31/2011|     336391.83|
|   NH|    03/31/2011|    7040302.64|
|   PA|    03/31/2011|    8241318.26|
|   MA|    03/31/2011| 1168511063.45|
|   VT|    03/31/2011|14564606674.20|
|   PA|    03/31/2012|    6260265.01|
|   GA|    03/31/2012|    4402372.46|
|   VT|    03/31/2012|14613390813.89|
|   NH|    03/31/2012|   15779425.80|
|   VA|    03/31/2012|     392115.68|
|   MA|    03/31/2012| 2333892646.13|
|   VA|    03/31/2013|     680498.51|
|   PA|    03/31/2013|    6297576.34|
+-----+--------------+--------------+
only showing top 20 rows
```

You can continue working in Apache Spark with grouping, aggregating and sorting your data until you have a result table, which might be interesting for you. This result can easily be moved into Pandas[30] or Excel where creating of bar-plots is very easy:

```
import pandas as pd
import matplotlib.pyplot as plt
pandas_df = spark_df.toPandas()
```

We have now left the “Big Data” environment Spark and are entering the “Pandas”, where creating plots is easier, but only possible on a “small” number of data points. Here we only a small amount of data points remains. For example:

[30] Pandas, https://pandas.pydata.org/

	Department	sumofAmount
0	Buildings & Gen Serv-Prop	254664862145.07
1	Vermont Health Access	7316059819.96
2	Natural Res Central Office	6115935633.73
3	Education Agency	5156573496.88
4	Education	3166972698.64
5	Transportation Agency	2795155337.06
6	Department of VT Health Access	2393175142.17
7	Finance & Management	2331413298.90
8	Agency of Transportation	1920833560.49
9	Children and Families	1850543830.45

Figure 3.42: Big Data Visualization - Data Format

Similarly you may want to plot a chart of the aggregated “Amount” for a “Quarter Ending”:

```
spark_df = data.groupBy('Quarter Ending') \
               .agg(F.sum('Amount') \
                    .cast('decimal(20,2)') \
                    .alias('sumofAmount')) \
               .sort('Quarter Ending', ascending=True)
```

The result is a series with timestamps and the “sumofAmount”, which can be plotted with Pandas very easily.

```
spark_df = data.groupBy('Quarter Ending') \
               .agg(F.sum('Amount') \
                    .cast('decimal(20,2)') \
                    .alias('sumofAmount')) \
               .sort('Quarter Ending', ascending=True)
pandas_df = spark_df.toPandas()
pandas_df['Quarter Ending'] = pandas_df['Quarter
↪  Ending'].astype('datetime64')
pandas_df['sumofAmount'] = pandas_df['sumofAmount'].astype('float')
pandas_df.plot(x='Quarter Ending',
               y='sumofAmount')
```

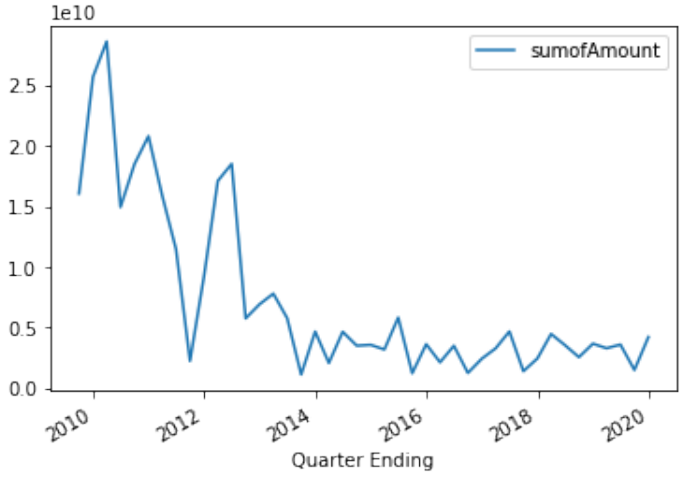


Figure 3.43: Big Data Visualization - Line Chart

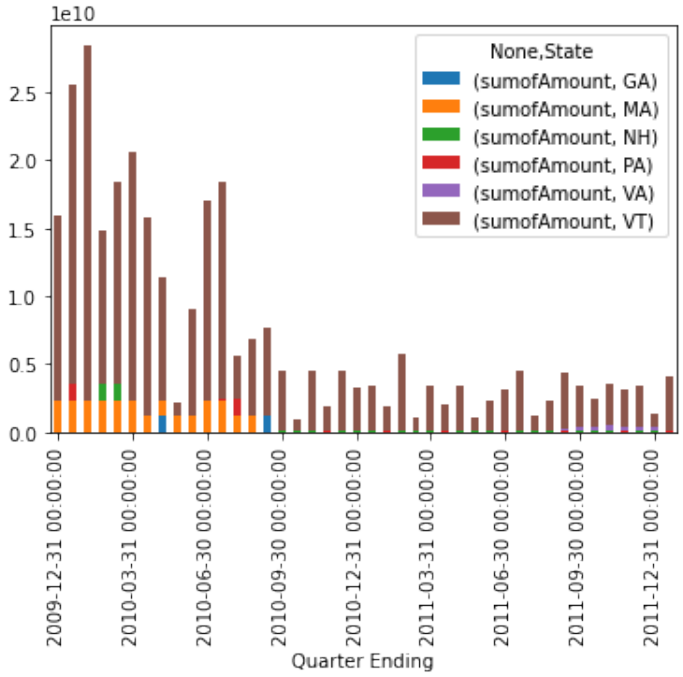


Figure 3.44: Big Data Visualization - Stacked Barplot

```
pandas_df['Quarter Ending'] = pandas_df['Quarter
 ↪  Ending'].astype('datetime64')
pandas_df['sumofAmount'] = pandas_df['sumofAmount'].astype('float')
ax = pandas_df.groupby(['Quarter Ending',
 ↪  'State']).sum().unstack().plot(kind='bar', stacked=True)
ax.xaxis.set_major_locator(plt.AutoLocator())
```

```
plt.show()
```

I think this example shows, how easy the visualization of Big Data datasets can be done, if you use more advanced tools instead of Excel.

On my Docker machine-learning repository[31] you will find the Jupyter-Notebook, with this example.

3.3 Visualize Data with Data Apps

Now, in the second part of this chapter, I will show you how to visualize and share the data with a "data app". I used Streamlit[32], which is surprisingly easy if you want to connect your data with python code directly to a very intuitive and easy to use application.

Data Scientist often forget, that all their visualizations (and also model), which they have built, need to be used by someone, who is probably not as skilled in all these technical requirements! Therefore it is important to find a solution, which is **easy to deploy** and **easy to use** for everyone (as well on a computer as also on a mobile phone), **stable** and **quickly customizable**.

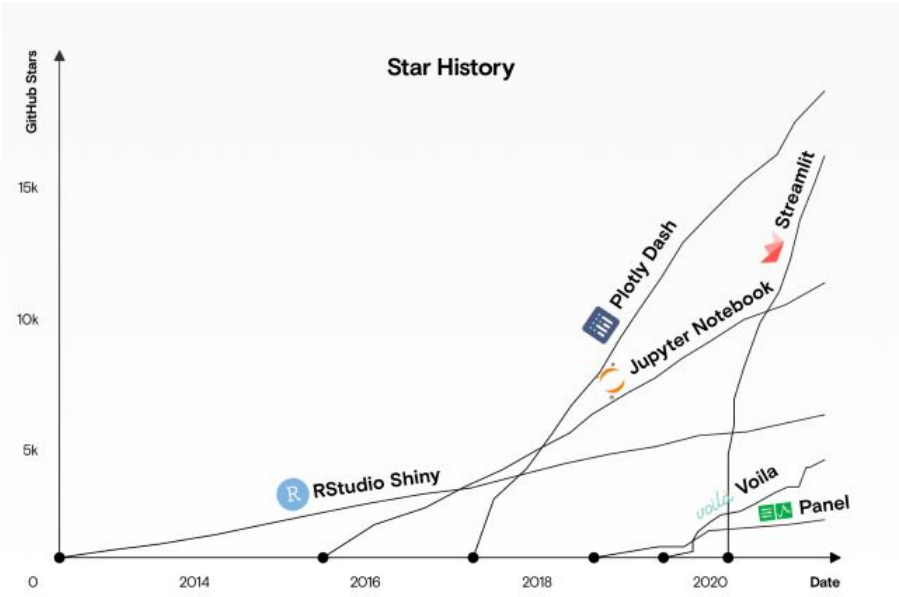


Figure 3.45: Data App - Comparison of Streamlit, RStudio-Shiny and others, Quelle: DataRevenue-Blog

[31] My docker machine-learning repository, https://hub.docker.com/repository/docker/andreastraut/machine-learning-pyspark

[32] Streamlit, https://www.streamlit.io/

There are different solutions: when you are using the programming language R then the combination of Tidyverse[33] and Shiny-App[34] will be an interesting option for you. But to me the *"R / Tidyverse / Shiny"* bundle seems a be the *"old-standard"* or even a bit *"old-fashioned"* as an article on Data-Revenue[35] reveals a strong increasing popularity for Streamlit).

I tested Streamlit[36] and think: it is fantastic, because I didn't have to spend time on building a webpage or learn HTML, CSS or Wordpress. Everything is in Python and once the setup is done (which is easy) all I have to do for updating the whole data app is to save the Python file (no compiling needed). I believe that Python in combination with Streamlit is a very strong combination which will beat the *"R / Tidyverse / Shiny"* alternative! Here are some examples:

I used the data of the "Marathon runtimes" example and as you can see I only had to change some very minor things in the python code (like `import streamlit as st` and write `st.pyplot(g)` instead of `plt.show()`) in order to create a "data app". You can upload another Excel-csv file by pressing the "Browse files" button, which will then be visualized. Using the checkboxes below will open more graphics (like histograms, kernel density, violin plots,...). See my "data app"[37] and play around yourself. I uploaded the results of these two datasets (the left and right side of fig. 3.46) in the results folder.

[33] Tidyverse, https://www.tidyverse.org/

[34] Shiny-App, https://shiny.rstudio.com/

[35] Data-Revenue, https://www.datarevenue.com/de-blog/streamlit-vs-dash-vs-shiny-vs-voila-vs-flask-vs-jupyter

[36] Streamlit, https://www.streamlit.io/

[37] Marathon Streamlit "data app", https://share.streamlit.io/andreastraut/visualize-results-in-apps/main/app_Example_Marathon_extended.py

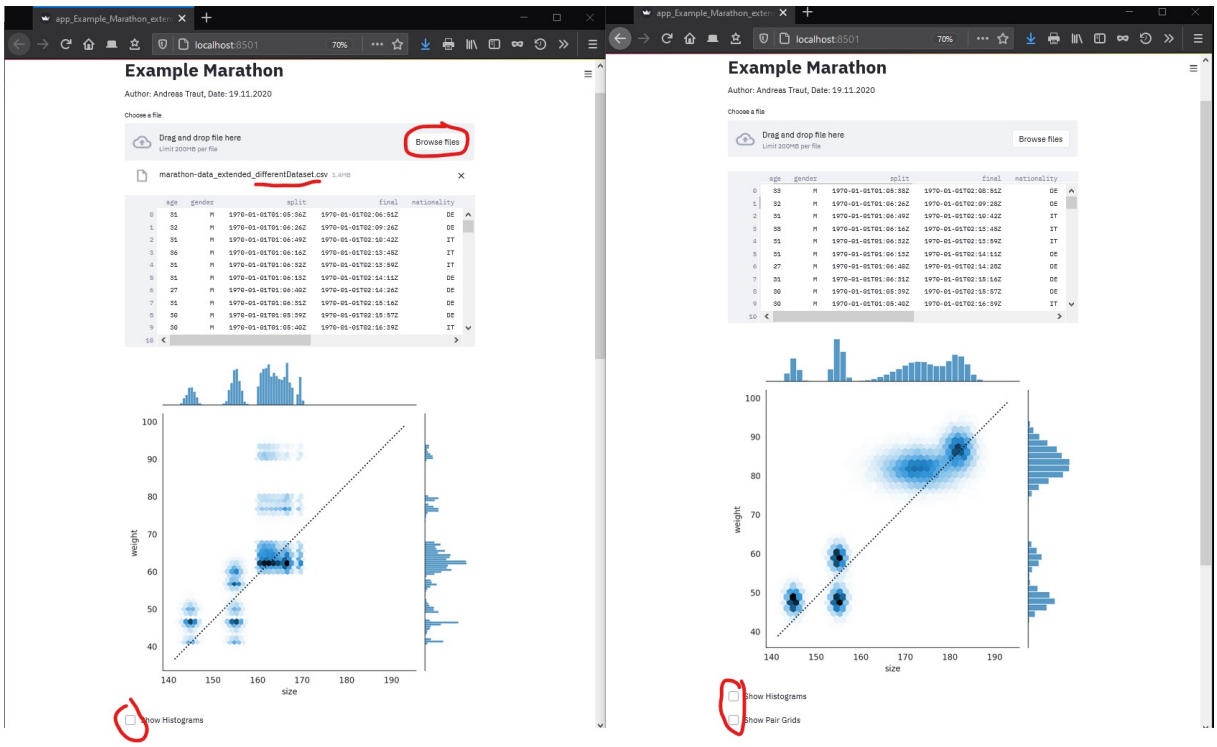


Figure 3.46: Data App - Marathon Example

And here is a second example: as everyone is talking about Corona/Covid and epidemiological models I thought that implementing the SEIR-Model would be an interesting example. Believe me: I read the Wikipedia SEIR-article[38] and implemented a Streamlit app[39] in less than half an hour. This is why I love Streamlit: highly efficient, lovely design and easy to deploy.

[38] Wikipedia SEIR model, https://de.wikipedia.org/wiki/SEIR-Modell

[39] SEIR modell app, https://share.streamlit.io/andreastraut/visualize-results-in-apps/main/app_SEIR_model

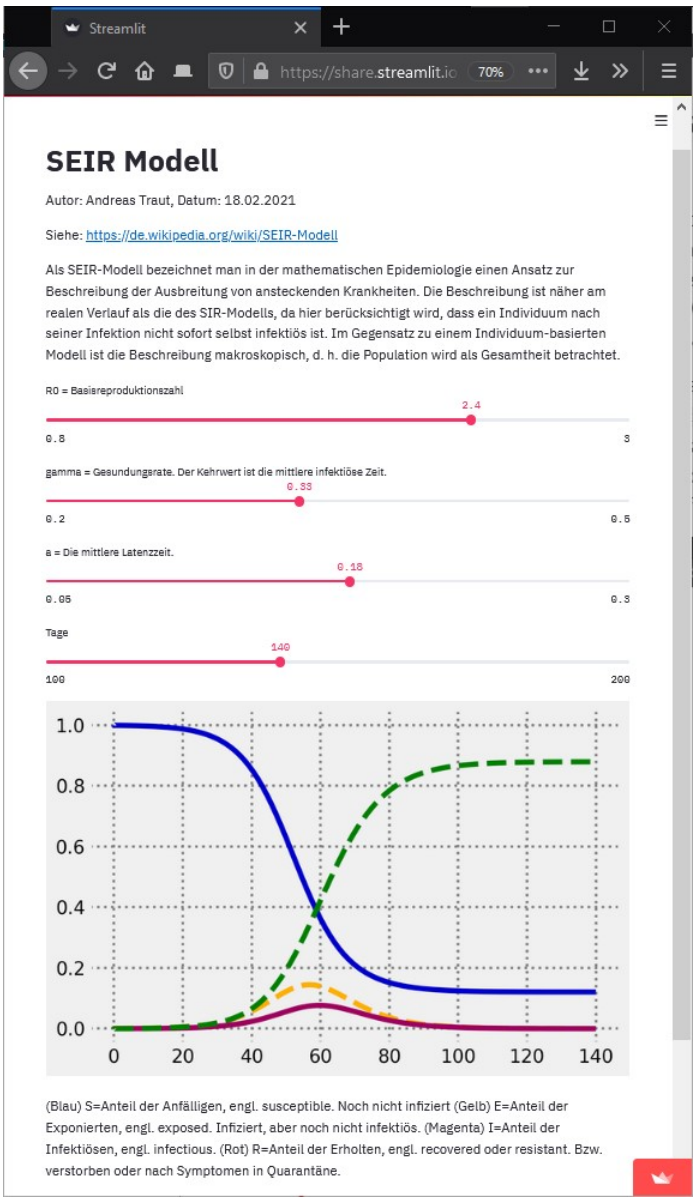


Figure 3.47: Data App - SEIR Model Example

3.4 Professional Tools

I will list some common professional tools, which offer visualization functionality and more. These tools cost some money, but I recommend to have look into these: many companies use these or similar tools. There are different tools, which provide fantastic possibilities for visualization of data.

Additionally these tools provide a lot of functionality concerning other topics, like

- **"data integration"** (how can different data sources be connected?) or

- **“reporting”** (how can beautiful dashboards, which show all relevant graphics, be created?).

Obviously the best tools are not for free. I will only list up some examples here in my book and I recommend to read the official documentation on their websites for getting a feeling about what these tools can do and maybe also their limitations.

3.4.1 Power BI

See here: https://powerbi.microsoft.com/de-de/

Power BI from Microsoft is a very popular and probably the leading visualization tool for Big Data.

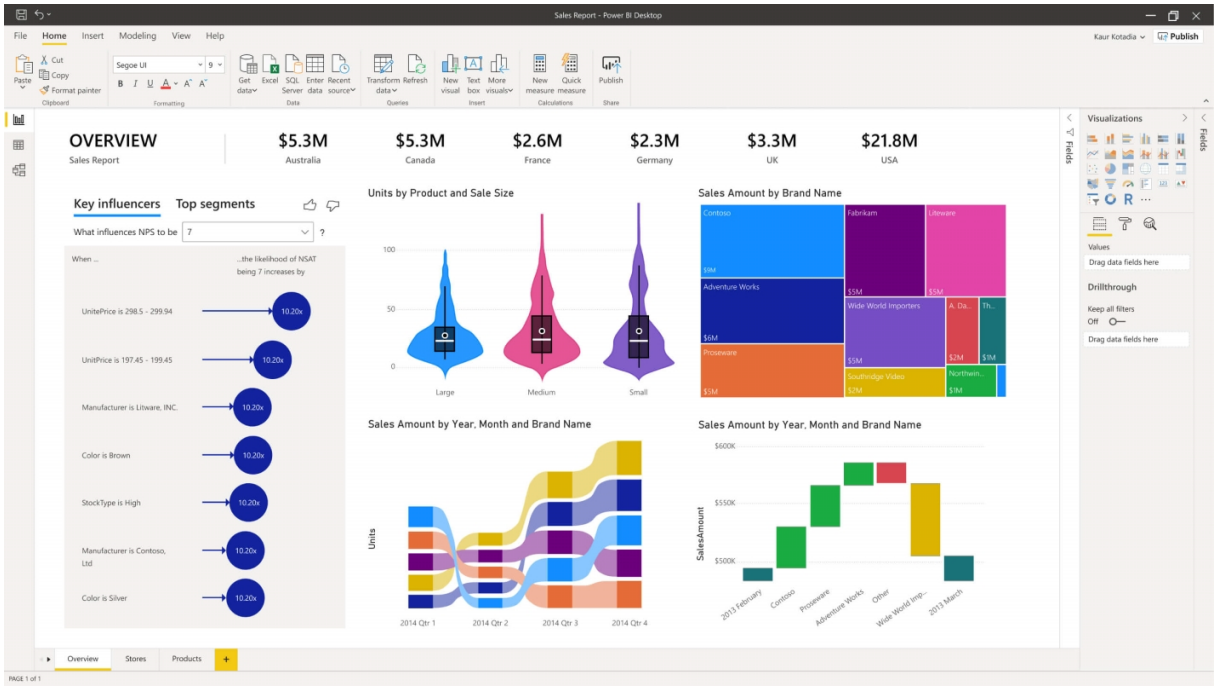


Figure 3.48: Professional BI - Power BI

Figure 3.49: Professional BI - Power BI Prices

3.4.2 Tableau

See here: https://www.tableau.com/de-de

Tableau from Salesforce (which is an Oracle subsidiary) is a very interesting alternative.

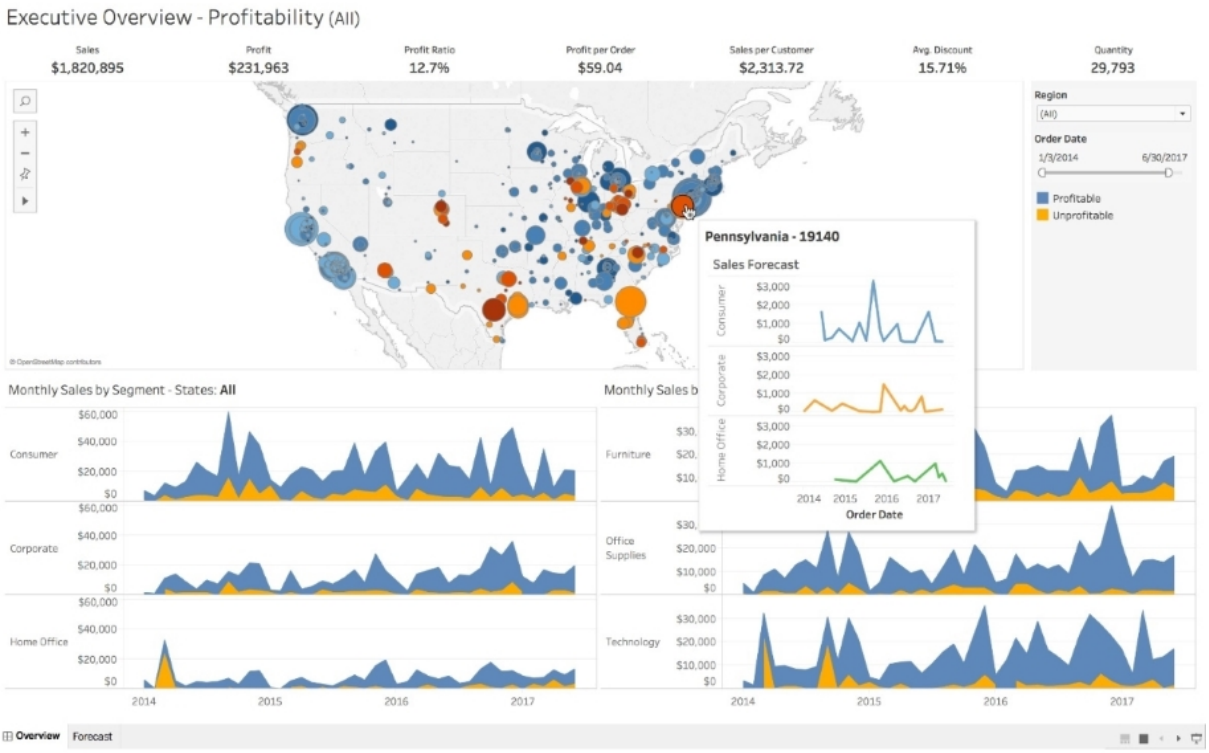


Figure 3.50: Professional BI - Tableau 1

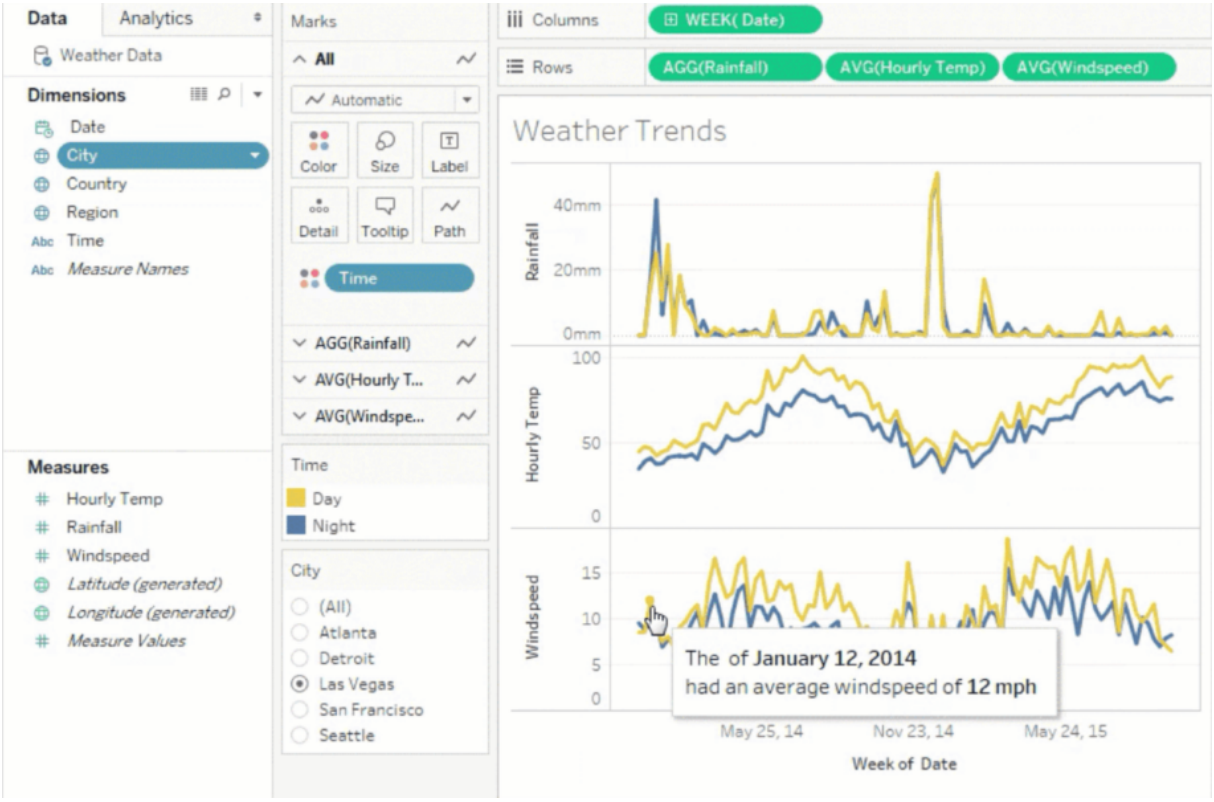


Figure 3.51: Professional BI - Tableau 2

Figure 3.52: Professional BI - Tableau Prices

3.4.3 QLink

See here: https://www.qlik.com/de-de/

QLink is a fast growing tool for business intelligence and data visualization.

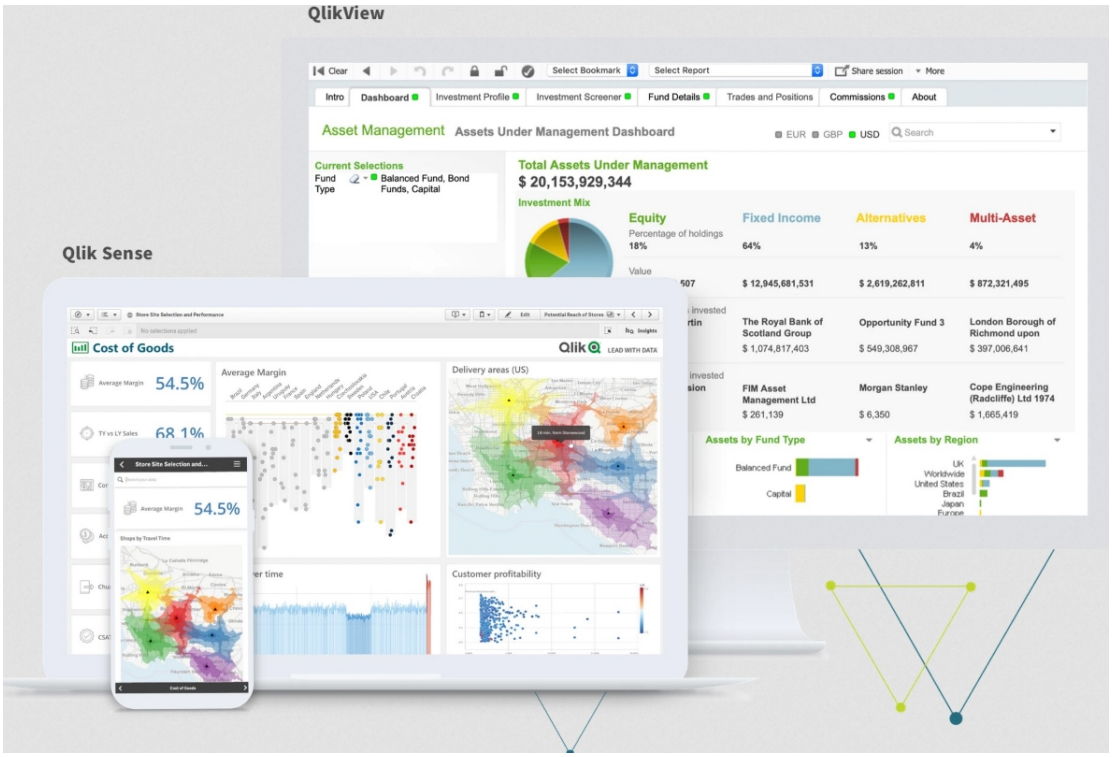


Figure 3.53: Professional BI - QLink

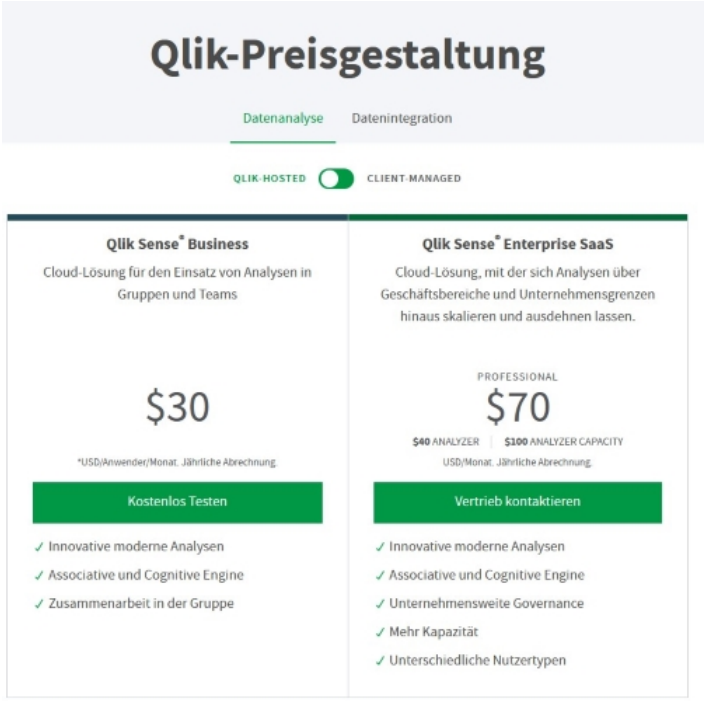


Figure 3.54: Professional BI - QLink Prices

4 Machine Learning with Python: Comparison Small Data vs Big Data

4.1 Movies Database Example

This first example should give you an impression of how machine learning works in Python. I think I should start without any "formal structure" first and develop the machine-learning problem before generating too much formalism. In sec. 4.2 I will develop a generally valid structure in the form of a mind map. And in the two examples that follow (see sec. 4.3 and sec. 4.4), I will apply and explain this generally valid structure and adapt mind map slightly for each of these specific problems.

A good starting point for finding data science datasets is Kaggle[1]. I downloaded the movies dataset[2]. The dataset from Kaggle contains the following columns:

```
Rank | Title | Year | Score | Metascore | Genre | Vote |
Director | Runtime | **Revenue** | Description | RevCat
```

In this example I want to predict the **Revenue** based on the other information, which I have for each movie (e.g. every movie has a year, a scoring, a title …). There are some NULL-values in the column "Revenue" and instead of filling them with an assumption (e.g. median-value) as I did in another Jupiter-Notebook[3], I wanted to predict these values. You might guess the conclusion already: predicting the revenue based on the available information as shown above (the columns) might not work. But essential to me is more to follow a well established standard-process of data-cleaning, data-preparing, model-training and error-calculation in this example in order to learn how to apply this process to better datasets, than the movies-dataset, later.

Therefore, here is how I approached the problem step-by-step:

---

[1] Kaggle, www.kaggle.com

[2] Movies Dataset from Kaggle, https://www.kaggle.com/isaactaylorofficial/imdb-10000-most-voted-feature-films-041118

[3] Movies Stratified Sample Extended Jupyter-Notebook, https://github.com/AndreasTraut/Machine-Learning-with-Python/blob/master/Movies%20Machine%20Learning%20-%20StratifiedSample.ipynb

4.1.1 Import the Data

First we need to import the data. I already showed in sec. 3.1.1 how regional settings and conversions can be tricky sometimes. But here it is a bit easier as we will see, when we examine the `head` of the dataset and the datatypes (most columns are integer or float numbers already):

```
def load_data(path=PATH):
    csv_path = os.path.join(path, "movies.csv")
    return pd.read_csv(csv_path)
movies = load_data()
movies.head(3)
```

Rank	Title	Year	Score	Metascore	Genre	Vote	Director	Runtime	Revenue
1	The Shawshank Redemption	1994	9.3	80.0	Drama	2011509	Frank Darabont	142	28.34
2	The Dark Knight	2008	9.0	84.0	Action, Crime, Drama	1980200	Christopher Nolan	152	534.86
3	Inception	2010	8.8	74.0	Action, Adventure, Sci-Fi	1760209	Christopher Nolan	148	292.58

The datatypes of the columns are (use `movies.dtypes`):

```
Rank             int64
Title           object
Year             int64
Score          float64
Metascore      float64
Genre           object
Vote             int64
Director        object
Runtime          int64
Revenue        float64
Description     object
dtype: object
```

And with `movies.info()` we get also information about the count of non-null entries:

```
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Rank         10000 non-null  int64
 1   Title        10000 non-null  object
 2   Year         10000 non-null  int64
 3   Score        10000 non-null  float64
 4   Metascore    6781 non-null   float64
 5   Genre        10000 non-null  object
 6   Vote         10000 non-null  int64
 7   Director     9999 non-null   object
 8   Runtime      10000 non-null  int64
 9   Revenue      7473 non-null   float64
 10  Description  10000 non-null  object
dtypes: float64(3), int64(4), object(4)
```

4.1.2 Separate NULL-Values

First of all, let me mention that you talk about **NULL** when a record is missing or empty and you talk about **NaN** when a record is "Not-a-Number" which means 0/0. There are numerous discussions about the **difference of NULL to NaN**, but for me it is not too crucial to distinguish between one or the other. Remember that a NULL record is "junk" just like a NaN record: then you are not fooling a sophisticated specialist. Completely independent of whether you see a NULL record or a NaN record: you need to have look into them at they are "junk" and you have to eliminate the problem. There are different options, which I will explain later.

These are the datarows, where column "Revenue" is null:

```
In [10]: movies_RevenueNaN = movies[movies["Revenue"].isnull()]
         movies_RevenueNaN.head()
```

Out[10]:

	Rank	Title	Year	Score	Metascore	Genre	Vote	Director	Runtime	Revenue	Description
82	83	A Clockwork Orange	1971	8.3	80.0	Crime, Drama, Sci-Fi	662768	Stanley Kubrick	136	NaN	In the future, a sadistic gang leader is impri...
513	514	To Kill a Mockingbird	1962	8.3	87.0	Crime, Drama	262064	Robert Mulligan	129	NaN	Atticus Finch, a lawyer in the Depression-era ...
581	582	Death Proof	2007	7.0	NaN	Action, Thriller	236539	Quentin Tarantino	113	NaN	Two separate sets of voluptuous women are stal...
620	621	My Neighbour Totoro	1988	8.2	86.0	Animation, Family, Fantasy	226126	Hayao Miyazaki	86	NaN	When two girls move to the country to be near ...
685	686	Hachi: A Dog's Tale	2009	8.1	NaN	Drama, Family	212349	Lasse Hallström	93	NaN	A college professor's bond with the abandoned ...

```
In [11]: len_movies_RevenueNaN = len(movies_RevenueNaN)
         len_movies_RevenueNaN
```

```
Out[11]: 2527
```

Figure 4.1: Movies Database - NULL Values

In column "Revenue" there are 7473 "non-NULL" values, and 2527 "NULL" values. I separated the rows

with NULL-values in column “Revenue”. These are the 2527 datarows, where column “Revenue” is null (see fig. 4.1).

```
movies_RevenueNaN = movies[movies["Revenue"].isnull()]
len_movies_RevenueNaN = len(movies_RevenueNaN)
len_movies_RevenueNaN
```

And these are the 7473 columns where “Revenue” is not null:

```
movies_NotNull = movies[movies["Revenue"].notnull()]
len_movies_NotNull = len(movies_NotNull)
len_movies_NotNull
```

I want to see this in a plot:

```
fig, axs = plt.subplots()
axs.pie([len_movies_RevenueNaN, len_movies_NotNull],
        labels=['Revenue=NaN', 'Revenue NotNull'],
        colors  = ['green', 'yellow'])
plt.show
```

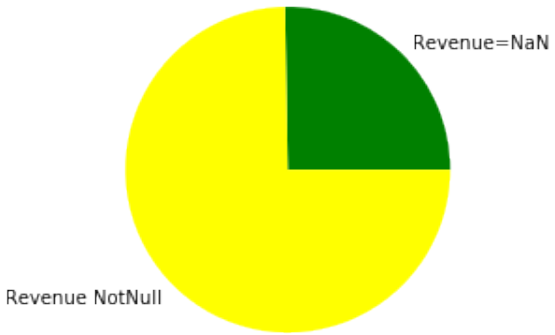


Figure 4.2: Movies Database - Plot Null and NotNull

4.1.3 Visualization of the Data

A first step should always be create some visualization of the data in order to better understand them. For me the histogram is often the very first plot, which I create, because it reveals a lot of information, like min/max-values, symmetry/skew of the distribution.

```
movies_NotNull['Revenue'].hist()
```

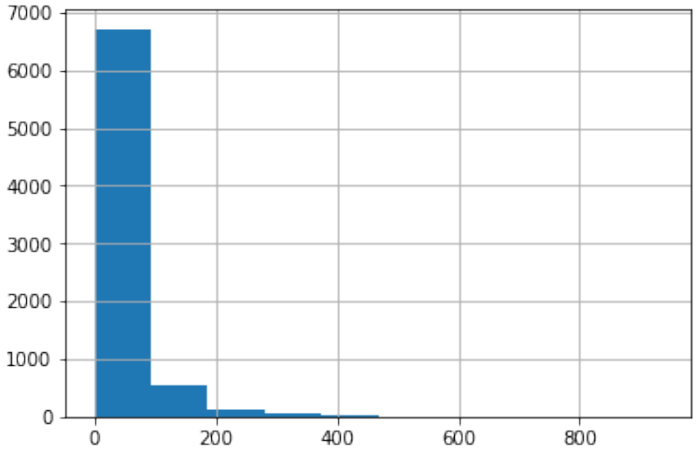


Figure 4.3: Movies Database - Histogram

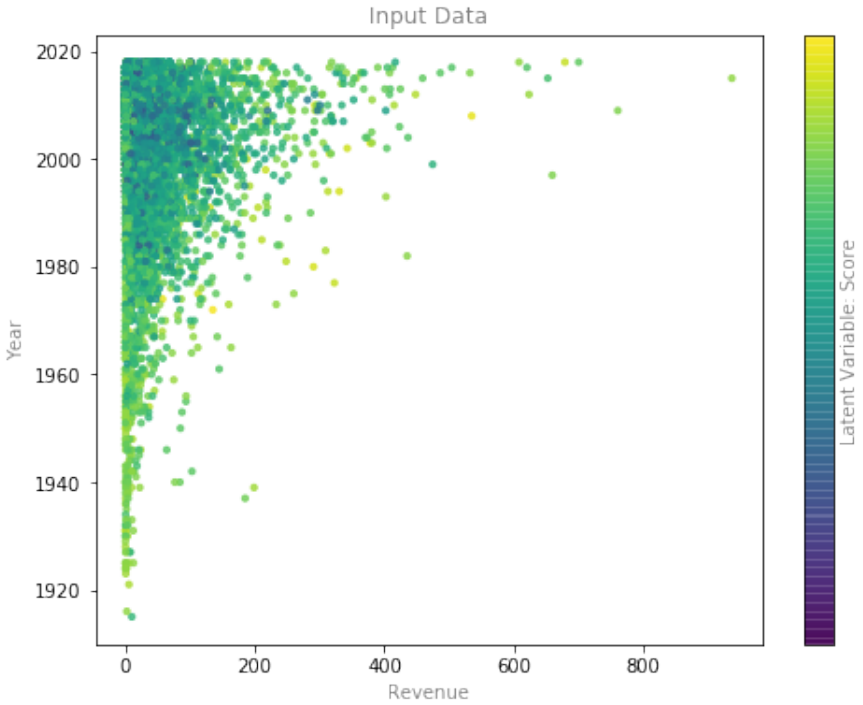


Figure 4.4: Movies Database - Scatterplot Revenue Year

Scatter plots are my second best choice, especially if they represent a third variable with colours apart from the two axes. For example I have chosen the axes `Revenue` and `Year` and displayed the `Score` with a colour scale.

```
fig, ax = plt.subplots(figsize=(8, 6))
point_style = dict(cmap='Paired', s=50)
pts = ax.scatter(movies_NotNull['Revenue'],
                 movies_NotNull['Year'],
                 c=movies_NotNull['Score'],
                 s=10,
                 alpha=0.8)
cb = fig.colorbar(pts, ax=ax)
# format plot
format_plot(ax, 'Input Data', 'Revenue', 'Year')
cb.set_ticks([])
cb.set_label('Latent Variable: Score', c
             olor='gray')
fig.savefig('images/movies/movies_revenue_year_score.png')
```

The result of this code is the plot fig. 4.4.

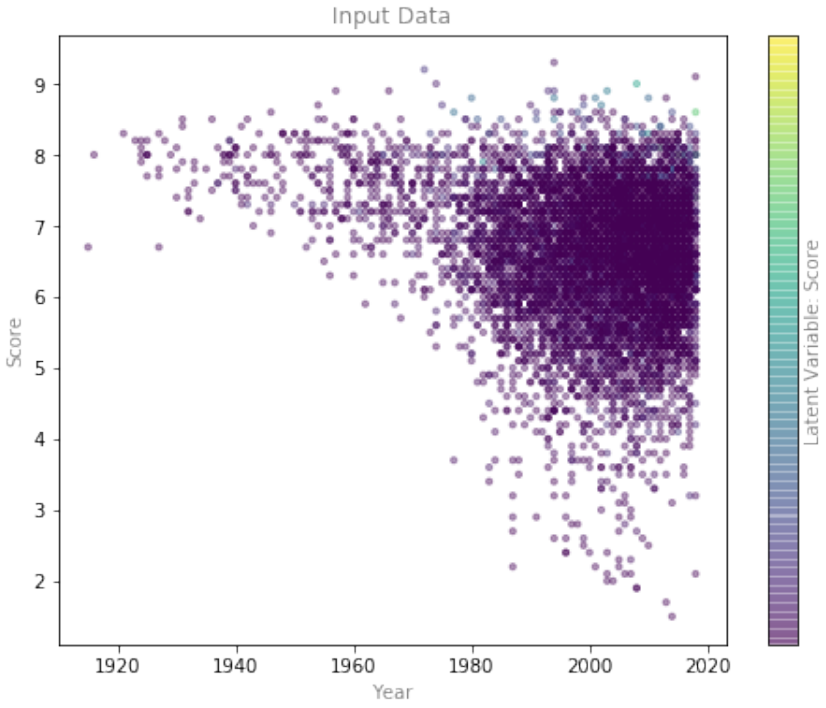


Figure 4.5: Movies Database - Scatterplot Year Score

```
fig, ax = plt.subplots(figsize=(8, 6))
point_style = dict(cmap='Paired', s=50)
pts = ax.scatter(movies_NotNull['Year'],
                 movies_NotNull['Score'],
```

```
                c=movies_NotNull['Revenue'],
                s=10,
                alpha=0.4)
cb = fig.colorbar(pts, ax=ax)
# format plot
format_plot(ax, 'Input Data', 'Year', 'Score')
cb.set_ticks([])
cb.set_label('Latent Variable: Score', color='gray')
fig.savefig('images/movies/movies_year_score_revenue.png')
```

4.1.4 Draw a Stratified Sample

I want to draw a stratified sample (based on "Revenue") on this remaining dataset. First copy the dataset `movies_NotNull`, which we already prepared above:

```
movies_NotNullC = movies_NotNull[:].copy(deep=True)
```

Then let's create bins and count the values being in these bins (which is basically a histogram or the distribution). I can determine the bins myself and experiment a bit which choice of bin values is probably best for me.

```
movies_NotNullC['RevCat']=pd.cut(movies_NotNullC['Revenue'],
                                 bins=[-1,100,200,300,np.inf],
                                 labels=[1, 2, 3, 4])
movies_NotNullC['RevCat'].value_counts()
```

```
1    6753
2     522
3     120
4      78
Name: RevCat, dtype: int64
```

In order to get the proportions, I only have to divide by `len(movies_NotNullC)` which is 7473:

```
movies_NotNullC["RevCat"].value_counts() / len(movies_NotNullC)
```

```
1    0.903653
2    0.069851
3    0.016058
```

```
4    0.010438
Name: RevCat, dtype: float64
```

Now I split the dataset into a training and a testing dataset using stratified sampling in order to have the same distributions in these datasets:

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1,
                               test_size=0.9,
                               random_state=42)
for train_index, test_index in split.split(movies_NotNullC,
                                           movies_NotNullC["RevCat"]):
    strat_train_set = movies_NotNullC.iloc[train_index]
    strat_test_set = movies_NotNullC.iloc[test_index]
```

`Sklearn` is probably the most popular machine learning library in Python. It is a very powerful and comprehensive library that is a must if you want to do machine learning in Python. It comprises classification (identify, which category an object belongs to), regression (predicting a continuous-valued attribute), clustering (grouping of similar objects to sets), dimensionality reduction (reducing the number of variables), model selection (choosing a model and parameters) and preprocessing (extrating features and other preliminary work on the source dataset).

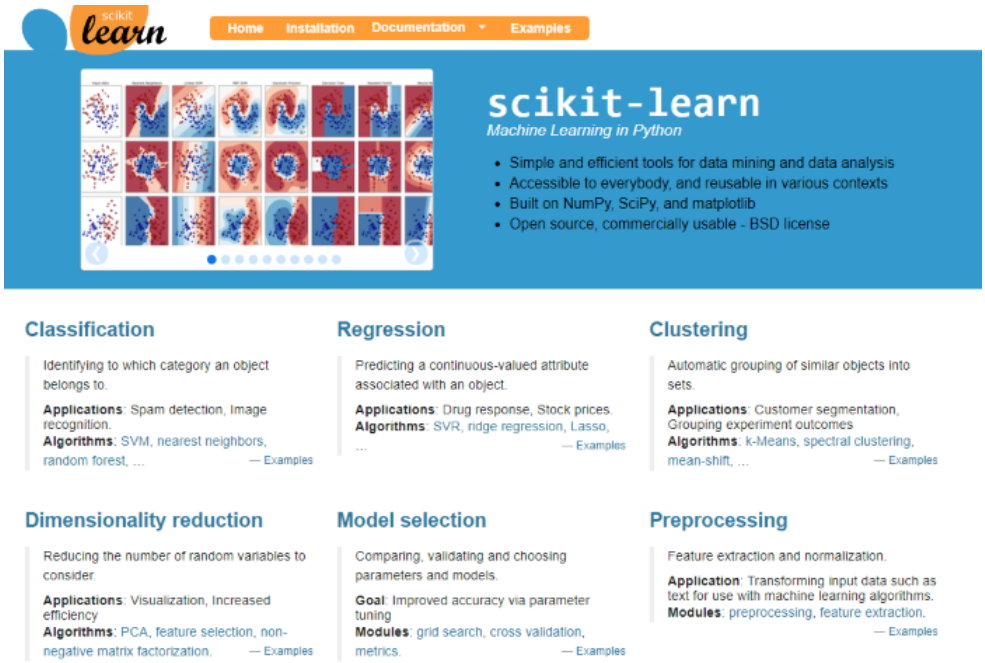


Figure 4.6: Scikit-Learn - The most popular machine learning library in Python

Calculate the proportions of the bins on the stratified sample by dividing `len(strat_train_set)` which is 747. The expectation is, that these proportions are the same as above (otherwise we would have an error somewhere):

```
strat_train_set["RevCat"].value_counts() / len(strat_train_set)
```

```
1    0.903614
2    0.069612
3    0.016064
4    0.010710
Name: RevCat, dtype: float64
```

Looks good so far: the numbers look very similar to the ones above. This means: the proportions in the stratified sample (which has only 1/10-th of the data compare of the whole dataset) correspond to the proportions in the whole dataset. That is what we wanted to have. Let's create a function `revenue_cat_proportions`to better compare these numbers: the overall data, the stratified sample and also a random sample. We can re-use this function later again:

```
def revenue_cat_proportions(data):
    return data["RevCat"].value_counts() / len(data)

train_set, test_set = train_test_split(movies_NotNullC,
                                       test_size=0.9,
                                       random_state=42)
compare_props = pd.DataFrame({
    "Overall": revenue_cat_proportions(movies_NotNullC),
    "Stratified": revenue_cat_proportions(strat_test_set),
    "Random": revenue_cat_proportions(test_set),
}).sort_index()

compare_props["Rand. %error"] = 100 * compare_props["Random"] /
 ↪  compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] /
 ↪  compare_props["Overall"] - 100

print(compare_props)

for set_ in (strat_train_set, strat_test_set):
    set_.drop("RevCat", axis=1, inplace=True)
```

```
    Overall  Stratified    Random  Rand. %error  Strat. %error
1  0.903653    0.903657  0.903063     -0.065336       0.000476
2  0.069851    0.069878  0.069432     -0.600432       0.038109
3  0.016058    0.016057  0.016652      3.699078      -0.004460
4  0.010438    0.010407  0.010853      3.983966      -0.289348
```

4.1.5 Split of Dataset into Training-Data and Test-Data

```
movies_train = strat_train_set.drop('Revenue', axis=1)
movies_train_labels = strat_train_set['Revenue'].copy()
len_movies_train = len(movies_train)

movies_test = strat_test_set.drop('Revenue', axis=1)
movies_test_labels = strat_test_set['Revenue'].copy()
len_movies_test = len(movies_test)
```

The whole dataset of 10000 rows has been split up into

- a training dataset (`movies_train`),
- a testing dataset (`movies_test`) and
- a dataset, where `Revenue` is NULL

Here are the counts:

```
len_movies_train = 747
len_movies_test = 6726

len_movies_train + len_movies_test = 7473
len_movies_NotNull = 7473
len_movies_RevenueNaN = 2527

len_movies_train + len_movies_test + len_movies_RevenueNaN  = 10000
```

Let's visualize these counts:

```
fracs = [len_movies_train, len_movies_test, len_movies_RevenueNaN]
labels = ['training data (movies_train)', 'testing data (movies_test)',
 ↪  'Revenue=NaN']
fig, axs = plt.subplots()
axs.pie(fracs, labels=labels)
plt.show
```

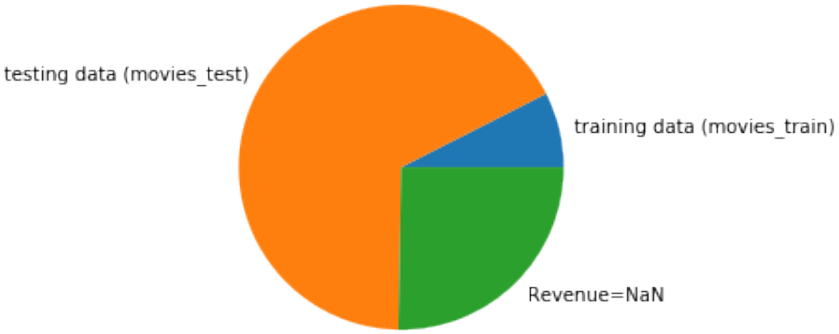


Figure 4.7: Movies Database - Stratified Sample (Train, Test, NaNs)

4.1.6 Create a Pipeline

Now I want to implement a pipeline for doing all the data preparation work quicker when I am testing it later. Only numerical-columns will be taken. The other will be thrown away (for the moment, I might change this later):

Created a pipeline to fill the NULL-value in other columns (e.g. `Metascore`, `Score`).

```
imputer = SimpleImputer(strategy="median")
movies_train_num = movies_train.select_dtypes(include=[np.number])
imputer.fit(movies_train_num)
X = imputer.transform(movies_train_num) # Transform the training set:
movies_tr = pd.DataFrame(X,
                         columns=movies_train_num.columns,
                         index=movies_train.index)
movies_tr.head()
```

	Rank	Year	Score	Metascore	Vote	Runtime
733	734.0	2007.0	6.3	68.0	204005.0	86.0
5851	5852.0	2003.0	5.2	21.0	16061.0	102.0
3816	3817.0	2003.0	5.3	47.0	33688.0	116.0
5384	5385.0	1980.0	6.7	58.0	18517.0	103.0
1058	1059.0	1998.0	6.7	63.0	152346.0	136.0

```
num_pipeline = Pipeline([('imputer', SimpleImputer(strategy='median')),])
num_attribs = ['Rank', 'Year', 'Score', 'Metascore', 'Vote', 'Runtime']
 ↪  #list(movies_train_num)
full_pipeline = ColumnTransformer([('num', num_pipeline, num_attribs)])
```

Apply now the full pipeline on the training-dataset `movies_train`. But before we do this, we have a look into the column `Metascore` and the NULL-values in the training dataset `movies_train`:

```
tmp = movies_train[movies_train["Metascore"].isnull()]
tmp.head(2)
```

Rank	Title	Year	Score	Metascore	Genre	Vote	Director	Runtime	Description
53845385	The Final Countdown	1980	6.7	NULL	Action, Sci-Fi	18517	Don Taylor	103	A modern aircraft carrier is thrown back in ti…
59145915	The Chase	1994	5.8	NULL	Action, Adventure, Comedy	15791	Adam Rifkin	89	Escaped convict Jack Hammond takes a woman hos…

Now apply the Pipeline:

```
movies_train_prepared = full_pipeline.fit_transform(movies_train)
```

Now let's count the NULL values in the new prepared dataset `movies_train_prepared`. I have to transform it back to a Pandas-Dataframe format first:

```
tmp_num = movies_train.select_dtypes(include=[np.number])
tmp_prep = pd.DataFrame(movies_train_prepared,
                        columns=tmp_num.columns,
                        index=movies_train.index)
tmp = tmp_prep[tmp_prep["Metascore"].isnull()]
tmp
```

Rank	Year	Score	Metascore	Vote	Runtime

Zero, as we wanted! All NULL-values in `movies_train_prepared` have been removed by the "median" value (this was how the pipeline was built). Great, this was part of the job, the pipeline should have done. The other part was to eliminate some columns. We now have only 6 remaining instead of 10 columns.

4.1.7 Fit the Model with "DecisionTreeRegressor"

I used the training dataset and fitted it with the "DecisionTreeRegressor" model

```
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(movies_train_prepared, movies_train_labels)
movies_predictions = tree_reg.predict(movies_train_prepared)
```

How big is the error for all training-datasets?

```
trainmse = mean_squared_error(movies_train_labels, movies_predictions)
trainrmse = np.sqrt(trainmse)
```

```
0.0
```

What?! Crazy! Does this make sense? Yes: the decision tree model is very powerful and on a small training dataset like mine is is able to create a perfect fit. I will need to cross-validate this result in order to know, if the model makes sense.

4.1.8 Cross-Validation

Cross-validation is a "rotation estimation" (or out-of sample testing) is often used, when we want to implement a prediction and serves to estimate how accurately the predictive model will perform in a real-word scenario. The prepared training dataset will be cut into pieces (here I choose 10) and rotating pieces will be used for the 10 validations. This results in a 10-fold cross-validation. So here is how I verified with the cross-validation function[4] from scikit-learn, how good this model/parameters are:

```
scores = cross_val_score(tree_reg,
                         movies_train_prepared,
                         movies_train_labels,
```

[4] Scikit-Learn - Cross-Validation, https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html

```
                         scoring="neg_mean_squared_error",
                         cv=10)
rmse_scores = np.sqrt(-scores)
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())
display_scores(rmse_scores)
```

```
Scores: [ 69.66166984  62.59977724  62.6450061  112.98391756  47.99491086
 ↪  52.16787811  42.65104005  83.21059338  41.09199207  63.84158945]
Mean: 63.88483746577791
Standard deviation: 20.447326452253154
```

These numbers look more useful, than what we had before. Now in the next step I will test test model on the testing dataset.

4.1.9 Test the model

We take on arbitrary row in the testing dataset "movies_test". Take for example row number 322:

```
some_data = movies_test.iloc[0:20]
some_data_label = movies_test_labels.iloc[0:20]
some_data.head(2)
```

	Rank	Title	Year	Score	Metascore	Genre	Vote	Director	Runtime
3123	3124	Victor Frankenstein	2015	6.0	36.0	Drama, Horror, Sci-Fi	45259	Paul McGuigan	110
3061	3062	Battleship Potemkin	1925	8.0	NULL	Drama, History	46636	Sergei M. Eisenstein	75

As we didn't apply the pipeline on the testing dataset (we only did on the training dataset `movies_train`), there might still some NULL values in columns `Metascore`.

Compare the true value ("some_data_labels") and the predicted value (`some_data_predicitons`) side-by-side. Left side is the true value from the original movies dataset. Right side is the predicted value based on the tree model:

```
some_movies = movies.iloc[some_data.index[0:len(some_data)]]
some_data_prepared = full_pipeline.fit_transform(some_data)
some_data_predictions = tree_reg.predict(some_data_prepared)
side_by_side = [(true, pred)
                for true, pred in
                zip(list(some_data_label),
                    list(some_data_predictions))]
side_by_side
```

```
[(5.78, 10.91),
 (0.05, 0.44),
 (0.3, 2.68),
...
 (36.0, 40.22),
 (3.61, 19.64),
 (0.59, 0.05),
 (0.99, 5.48)]
```

In this side-by-side comparison I can see, that some predicted values (right side) are already rather close to source values (left side), but there is room for improvements (which I explain later how to do this).

The mean-squared-error is as follows:

```
mse = mean_squared_error(some_data_label, some_data_predictions)
rmse = np.sqrt(mse)
rmse
```

```
51.3429319867886
```

In order to know, if this "rmse" is good enough, I will first taking the whole testing dataset:

```
movies_test_prepared =full_pipeline.fit_transform(movies_test)
movies_test_predictions = tree_reg.predict(movies_test_prepared)
lin_mse = mean_squared_error(movies_test_labels, movies_test_predictions)
lin_rmse = np.sqrt(lin_mse)
```

The result is `lin_rmse = 54.68`. Now I want to compare this to mean and standard deviation: `movies_test_labels.mean()` results in `36.20` and `movies_test_labels.std()` is `60.60`.

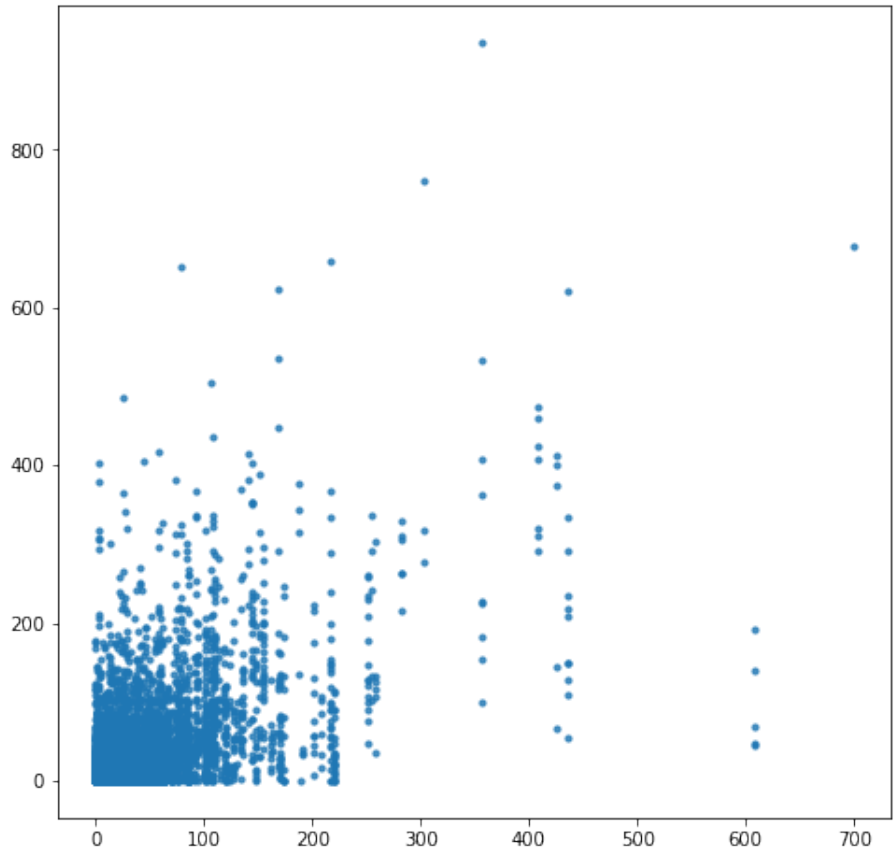


Figure 4.8: Movies Database - Plot True vs Predicted Value Side-by-Side Testdataset

A side-by-side comparison off the testing dataset. Left side is the true value from the original movies dataset. Right side is the predicted value based on the tree model:

```
side_by_side = [(true, pred)
                for true, pred in
                zip(list(movies_test_labels),
                    list(movies_test_predictions))]
```

Plotting the true labels and the predicted labels on the testing dataset:

```
fig, ax = plt.subplots(figsize=(8, 8))
pts = ax.scatter(movies_test_predictions,
                 movies_test_labels,
                 s=10,
                 alpha=0.8)
```

See fig. 4.8: Looks confusing. I would have expected something a bit more similar to the plot fig. 4.9. Plotting the same for the training dataset:

```
movies_train_prepared =full_pipeline.fit_transform(movies_train)
movies_train_predictions = tree_reg.predict(movies_train_prepared)
fig, ax = plt.subplots(figsize=(8, 6))
pts = ax.scatter(movies_train_predictions,
                 movies_train_labels,
                 s=10,
                 alpha=0.8)
```

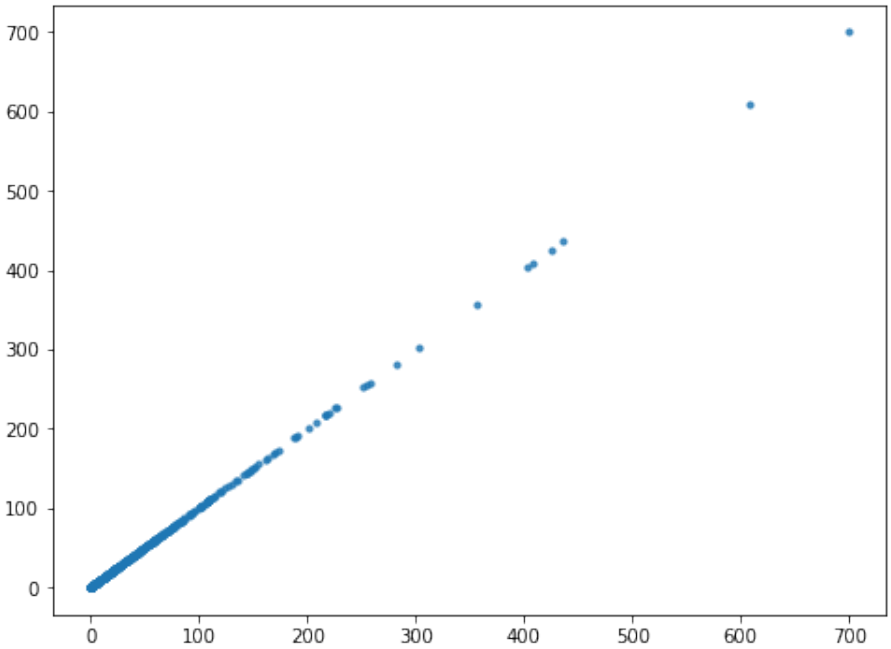


Figure 4.9: Movies Database - Plot True vs Predicted Value Side-by-Side Trainingdataset

Now I calculate the Revenue where it has NULL-values:

```
movies_RevenueNaN_prepared =full_pipeline.fit_transform(movies_RevenueNaN)
movies_RevenueNaN_predictions = tree_reg.predict(movies_RevenueNaN_prepared)
```

These are the predictions:

```
movies_RevenueNaN_predictions
```

```
array([74.1 , 11.99, 83.08, ...,  2.98, 43.49, 47.29])
```

I will insert the prediction into the dataset:

```
movies_RevenueNaN.loc[:,"Revenue"] = movies_RevenueNaN_predictions
```

4.1.10 Conclusion

The conclusion of this machine learning example is obvious: it is rather not possible to predict the "revenue" based on the available numerical information (the most useful numerical features were "year", "score", ...). Lot's of information, which is in the dataset has not yet been used, e.g. "genre", "director". These information could have an positive impact on the correctness of the predictions. But as "genre" has 486 different values it is a bit more complicated to treat them as "categorial" values:

```
movies['Genre'].value_counts()
```

```
Comedy, Drama, Romance     494
Drama                      482
Comedy, Drama              407
Drama, Romance             365
Comedy                     357
                          ...
Action, Fantasy, War         1
War                          1
Musical, Romance, War        1
Comedy, Romance, Family      1
Crime, Drama, Western        1
Name: Genre, Length: 486, dtype: int64
```

As already mentioned right in the beginning of this Jupyter-Notebook the "OneHotEncoder" could be used and I will explain sec. 4.3.5.2 how this works. But before we should work on these 486 categorial values: could we simplify it, e.g. extract "Drama" and use it as a separate criteria? This will a topic in another example.

On my GitHub profile you can find my Jupyter-Notebook for this example[5]

[5] Movies Example Jupyter Notebook, https://github.com/AndreasTraut/Machine-Learning-with-Python/blob/master/Movies%20Machine%20Learning%20-%20Predict%20NaNs.ipynb

4.2 Mind Map: Common Structure

The mind map in fig. 4.10 shows a generally valid structure for solving machine learning tasks. For me, these 10 steps are easy to remember and a useful guide to structure machine learning projects. You can also restructure the mind map as you like and build it differently. I'm not saying it's the best mind map ever, but it's definitely a good and useful start.

I will use apply, adapt and explain this mind map on the "Small Data" domain with Scikit-Learn and on the "Big Data" domain with "Apache Spark ML". This should build a small bridge between the "Small Data" and the "Big Data" world. You can hold the code side by side and compare, if you want.

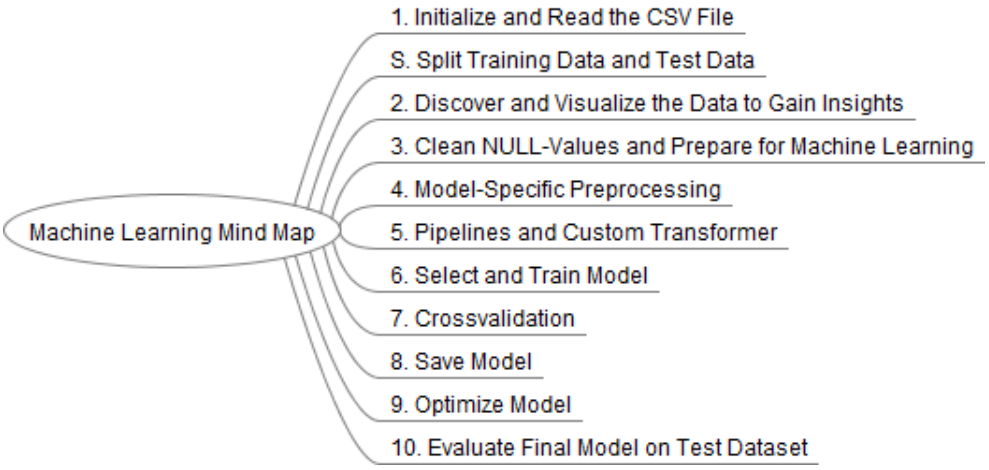


Figure 4.10: Machine Learning Mind Map - "Small Data" and "Big Data"

The 10 steps can be explained in short words as follows:

1. **Initialize and Read the CSV File** This step is understandably slightly different for "Small Data" (Scikit-Learn) and "Big Data" (Apache Spark ML). The installation of the environments is different, but so is the initialization (for Apache Spark ML, a Spark Session and a Spark Context must be initialized).
2. **Discover and Visualize the Data to Gain Insights** In this step you should try to get some first impressions about the dataset and understand the data better: column structure, number of rows, data types but also first visualizations like histograms and scatterplots.
3. **Clean NULL-Values and Prepare for Machine Learning** Only a few dataset will be complete. You need to think about how to deal with gaps (NULL values) and complete initial preparations to make the dataset machine-learning ready.
4. **Model-Specific Preprocessing** In this step, things get a bit more specific: depending on the data set and model you are aiming to implement (e.g. regression), different model-specific

preparatory work is necessary. Sometimes categorical values need to be edited so that they are correctly understood by the model.

5. **Pipelines and Custom Transformer** Since the preliminary work required in the previous step can be quite extensive, there is the very useful concept of pipelines. This allows you to better structure and efficiently organize the model-specific preliminary work.

6. **Select and Train Model** After all the preliminary work, this step is where the fun can finally begin: selecting a model and training it on the training dataset.

7. **Crossvalidation** Cross-validation is used to generate a robust model on a relatively small data set. The reason for this is that the available models are unfortunately so good that they would often produce a 100% perfect fit on a fixed data set. In order not to run into the so-called "overfitting" trap, a small data set is rotated and validated several times during cross-validation. This gives a better indication of the model error.

8. **Save Model** Of course, a model must also be stored and so this step sounds trivial. But it can be of great consequence: as shown in the CRISP-Cycle in fig. 3.1, the "deployment" is an essential component: ask yourself early on how a model, which was built in the development environment will be transferred later to the production environment. Depending on how your setup is, this can get complicated.

9. **Optimize Model** If you have arrived here, you have already achieved a lot. Now, for example, the grid search is used to find even better suitable parameter sets for your model and to optimize it.

10. **Evaluate Final Model on Test Data** After finding the best possible model, you can finally evaluate it and apply it to the test data set. After that you may think about how to transfer the model from your development environment to the production environment.

(S) **Split Training and Test Data** You may have noticed that I have not yet mentioned the step "(S) Split Training and Test Data". I have seen several examples in which this step (S) was performed in quite different places. For example, it could be directly between step 1 and 2. However, then all the preliminary work (e.g. cleaning up the NULL values, etc.) would not be included in the test data set. The step (S) could also be done between step 5 and 6 (after the pipeline where the transformations have been performed). Then the test data set would contain all the preliminary work. I think both approaches are possible and reasonable. So in my opinion you can include the step (S) where you think it makes sense, but should be clear about what the implications are.

4.3 "Small Data": Machine Learning using Scikit-Learn

The second example is a `.py` file for being used in an IDE (integrated developer environment), like the Spyder-IDE from the Anaconda distribution (see sec. 2 for hints on installing) and apply the *Scikit-Learn Python Machine Learning Library*[6] (you may call this example a "Small Data" example if you want).

I will apply the generally valid structure from sec. 4.2, adapt it to the "Small Data" example and put it into the mind-map fig. 4.11. This structure is also aligned to the "Big Data" Mind Map (Apache Spark ML, see fig. 4.18) in order to compare these two approaches. So let's start with the "Scikit-Learn": the Mind-Map fig. 4.11 shows you what we need to do. You will find the same structure in the `.py` file (which you can download from my repository) and it should be a guide to work out your own problems with the same structure.

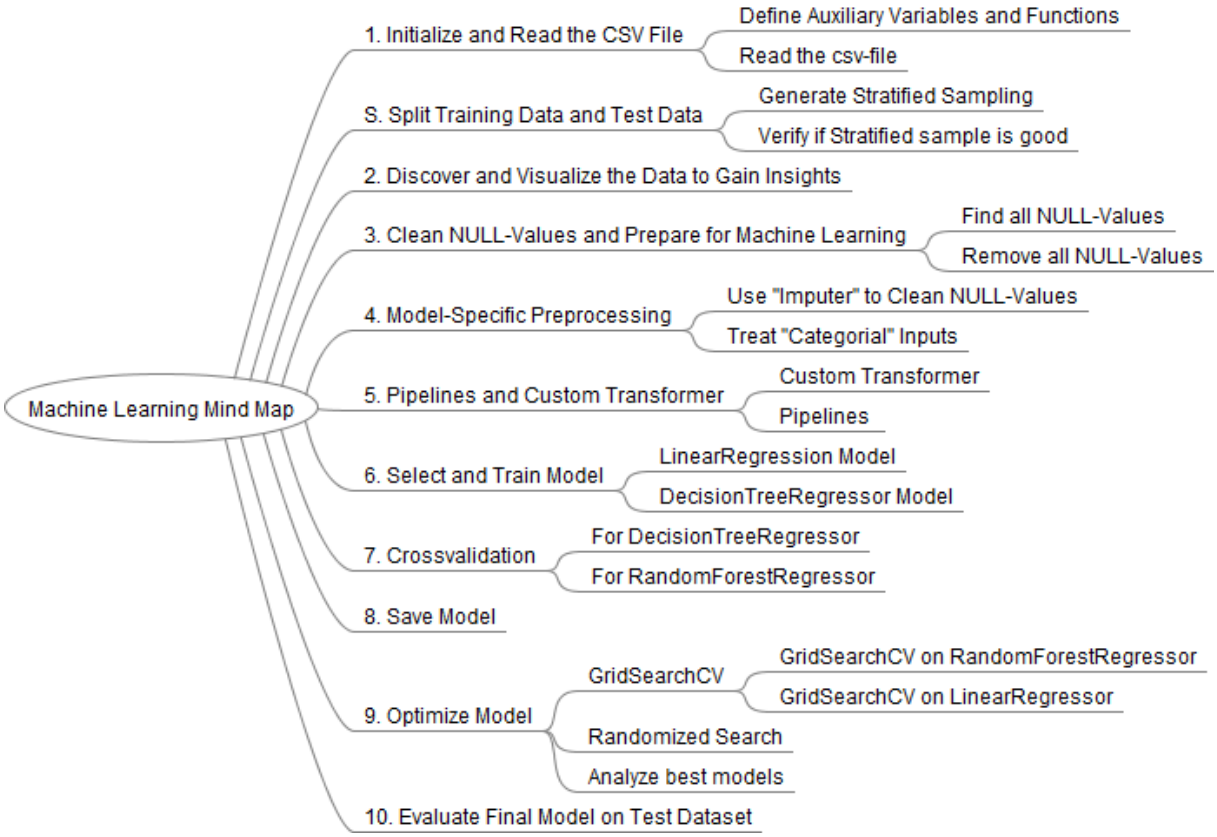


Figure 4.11: Machine Learning Mind Map - "Small Data" with Scikit-Learn

A quick note on my documentation, which follows not: I will refer to the official Scikit-Learn documentation more often. You need to get familiar with the official documentation anyway and learn how to quickly find, what you need for your specific problem. The official documentation for Scikit-Learn is very well structured and once you understood, how to navigate through it you won't need my book

[6] Scikit-Learn Python Machine Learning Library, https://scikit-learn.org/stable/

or Google or Stackoverflow[7] to solve your problems (Stackoverflow is a question and answer forum for programmers). Thus I apologize already once for the fact that I will not describe here everything in the last detail again, which is already much better described elsewhere.

4.3.1 Initialize and Read the CSV File (1)

4.3.1.1 Define Auxiliary Variables and Functions

These auxiliary variables and auxiliary functions are intended to make your work easier:

```
PROJECT_ROOT_DIR = "."
myDataset_NAME = "AirBnB"
IMAGES_PATH = os.path.join(PROJECT_ROOT_DIR, "media")
myDataset_PATH = os.path.join("datasets", "AirBnB")

def save_fig(fig_id, prefix=myDataset_NAME,
             tight_layout=True, fig_extension="png",
             resolution=300):
    path = os.path.join(IMAGES_PATH,
                        prefix + "_" + fig_id + "." +
                        fig_extension)
    print("Saving figure", prefix + "_" + fig_id)
    if tight_layout:
        plt.tight_layout()
    plt.savefig(path, format=fig_extension, dpi=resolution)
```

4.3.1.2 Read the csv-file

Define a function for reading the CSV file:

```
def load_myDataset_data(myDataset_path=myDataset_PATH):
    csv_path = os.path.join(myDataset_path, "listings.csv")
    return pd.read_csv(csv_path)

myDataset = load_myDataset_data()
print(myDataset.head())
```

The dataset has the following format (use `myDataset.info()`):

[7] Stackoverflow, https://stackoverflow.com

```
RangeIndex: 25164 entries, 0 to 25163
Data columns (total 15 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   name                            25114 non-null  object
 1   host_id                         25164 non-null  int64
 2   host_name                       25142 non-null  object
 3   neighbourhood_group             25164 non-null  object
 4   neighbourhood                   25164 non-null  object
 5   latitude                        25164 non-null  float64
 6   longitude                       25164 non-null  float64
 7   room_type                       25164 non-null  object
 8   price                           25164 non-null  int64
 9   minimum_nights                  25164 non-null  int64
 10  number_of_reviews               25164 non-null  int64
 11  last_review                     20636 non-null  object
 12  reviews_per_month               20636 non-null  float64
 13  calculated_host_listings_count  25164 non-null  int64
 14  availability_365                25164 non-null  int64
dtypes: float64(3), int64(6), object(6)
```

From this overview I already know a lot: there are 15 columns and most of them have 25164 non-null entries where the datatype is already something useful (e.g. int64 or float64). But some columns have NULL values (because the count is smaller) and some columns have an "object" datatype, which might need to be converted to something "useful". We will keep this in mind and continue the work.

4.3.2 Split Training Data and Test Data (S)

It is very important to know and define what the aim of the machine-learning problem is. Here the aim is to predict the "price" and use the other columns (or some of them) as features.

4.3.2.1 Generate Stratified Sampling

I want to create the training dataset and the test dataset by applying stratified sampling:

```
myDataset["price"].hist()
myDataset["price_cat"] = pd.cut(myDataset["price"],
                                bins=[-1, 50, 100, 200, 400, np.inf],
                                labels=[50, 100, 200, 400, 500])
print("\nvalue_counts\n", myDataset["price_cat"].value_counts())
myDataset["price_cat"].hist()
```

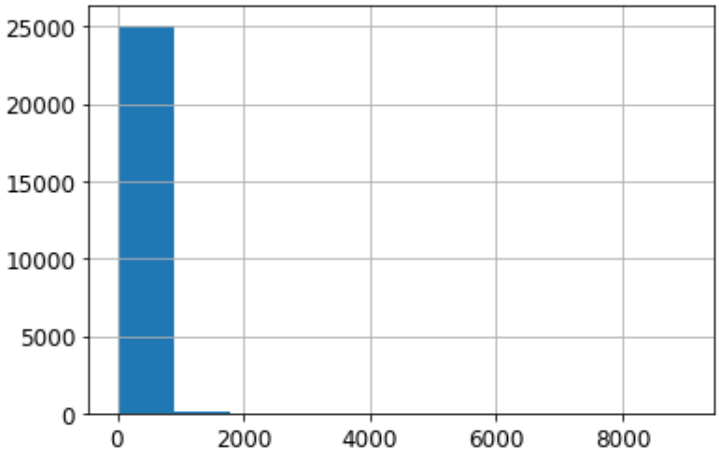


Figure 4.12: Small Data - AirBnB Histogram

There is a very long tail in the price (going to about 9000). Therefore I decided to have several bins for the lower values (50, 100, 200, 400).

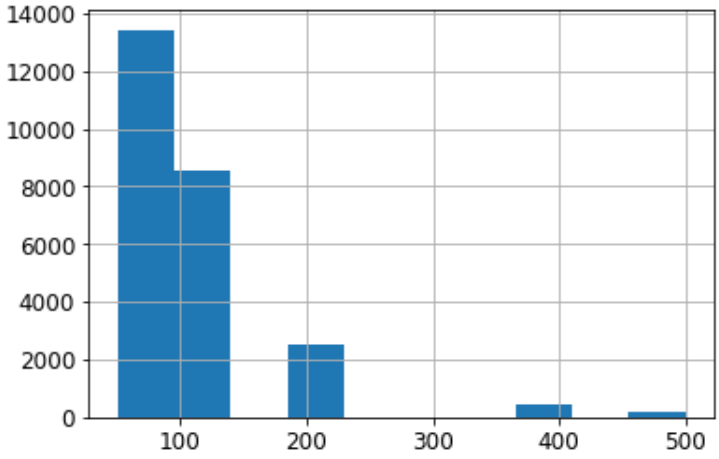


Figure 4.13: Small Data - AirBnB Histogram Stratified Sample

```
split = StratifiedShuffleSplit(n_splits=1,
                               test_size=0.2,
                               random_state=42)
for train_index, test_index in split.split(myDataset,
                                           myDataset["price_cat"]):
    strat_train_set = myDataset.loc[train_index]
    strat_test_set = myDataset.loc[test_index]
```

4.3.2.2 Verify if Stratified sample is good

Next step is to verify if the stratified sample is good and compare it to a random sampling

```
def price_cat_proportions(data):
    return data["price_cat"].value_counts() / len(data)

train_set, test_set = train_test_split(myDataset, test_size=0.2,
 ↪  random_state=42)

compare_props = pd.DataFrame({
    "Overall": price_cat_proportions(myDataset),
    "Stratified": price_cat_proportions(strat_test_set),
    "Random": price_cat_proportions(test_set),
}).sort_index()

compare_props["Rand. %error"] = 100 * compare_props["Random"] /
                                compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] /
                                compare_props["Overall"] - 100
```

From the `%error` columns I can see, that the stratified sample is always better, than the random sampling.

```
      Overall  Stratified    Random  Rand. %error  Strat. %error
50   0.533659    0.533678  0.536062      0.450249       0.003473
100  0.340168    0.340155  0.343731      1.047386      -0.003974
200  0.101256    0.101331  0.097755     -3.457526       0.074516
400  0.017326    0.017286  0.015498    -10.554013      -0.233322
500  0.007590    0.007550  0.006954     -8.380604      -0.527513
```

4.3.3 Discover and Visualize the Data to Gain Insights (2)

Now it it time to create some visualizations as I described in sec. 3. The first plot it a bad example, because the big blue dots don't reveal any interesting information about the `price` (which was our prediction variable).

```
myDataset.plot(kind="scatter",
               x="longitude", y="latitude",
               title="bad_visualization_plot")
save_fig("bad_visualization_plot")
```

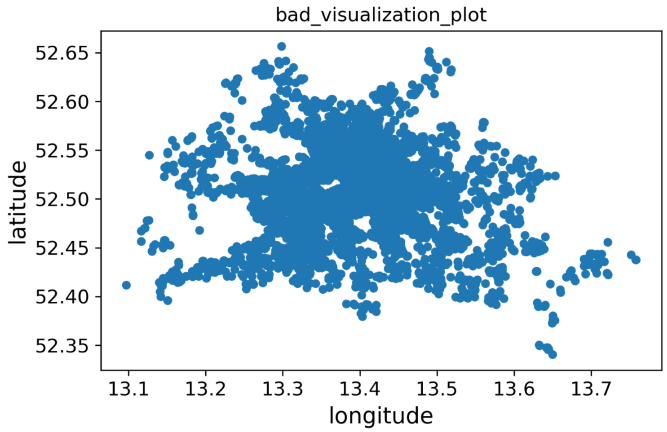


Figure 4.14: Small Data - AirBnB Bad Visualization

Better would be to include the price by a heat color.

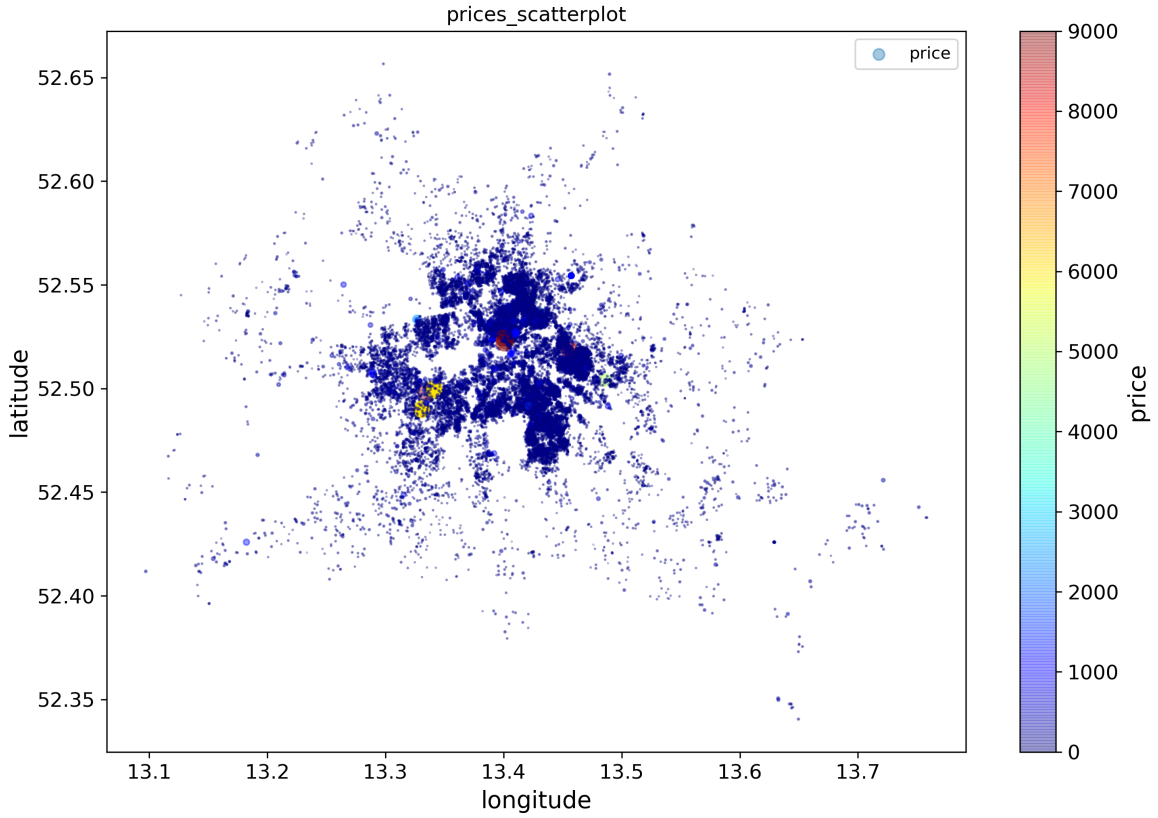


Figure 4.15: Small Data - AirBnB Better Visualization

I think it is interesting, that there are some AirBnB offers, where the price is very high (more than 5000).
I wanted to zoom in and used `myDataset = myDataset[(myDataset['longitude']>=13.32)`

`& (myDataset['longitude']<=13.35)]` for this reason:

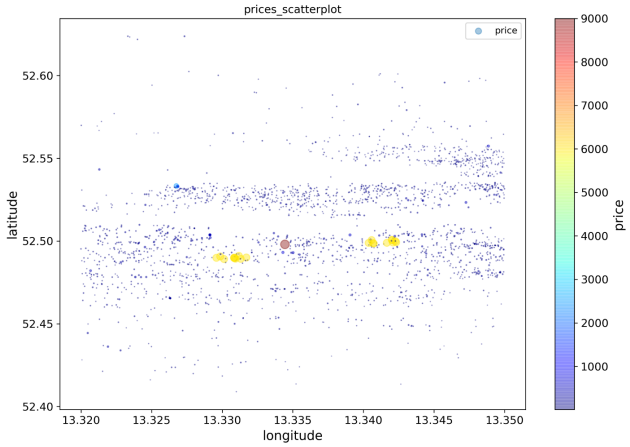


Figure 4.16: Small Data - AirBnB Better Visualization Zoomed

```
myDataset.plot(kind="scatter",
               x="longitude", y="latitude",
               alpha=0.4, s=myDataset["price"]/100,
               label="price", figsize=(10,7),
               c="price", cmap=plt.get_cmap("jet"),
               colorbar=True, sharex=False,
               title="prices_scatterplot")
plt.legend()
save_fig("prices_scatterplot")
```

```
attributes = ["number_of_reviews", "host_id",
              "availability_365", "reviews_per_month", "minimum_nights"]
scatter_matrix(myDataset[attributes],
               figsize=(12, 8))
plt.suptitle("scatter_matrix_plot")
save_fig("scatter_matrix_plot")
```

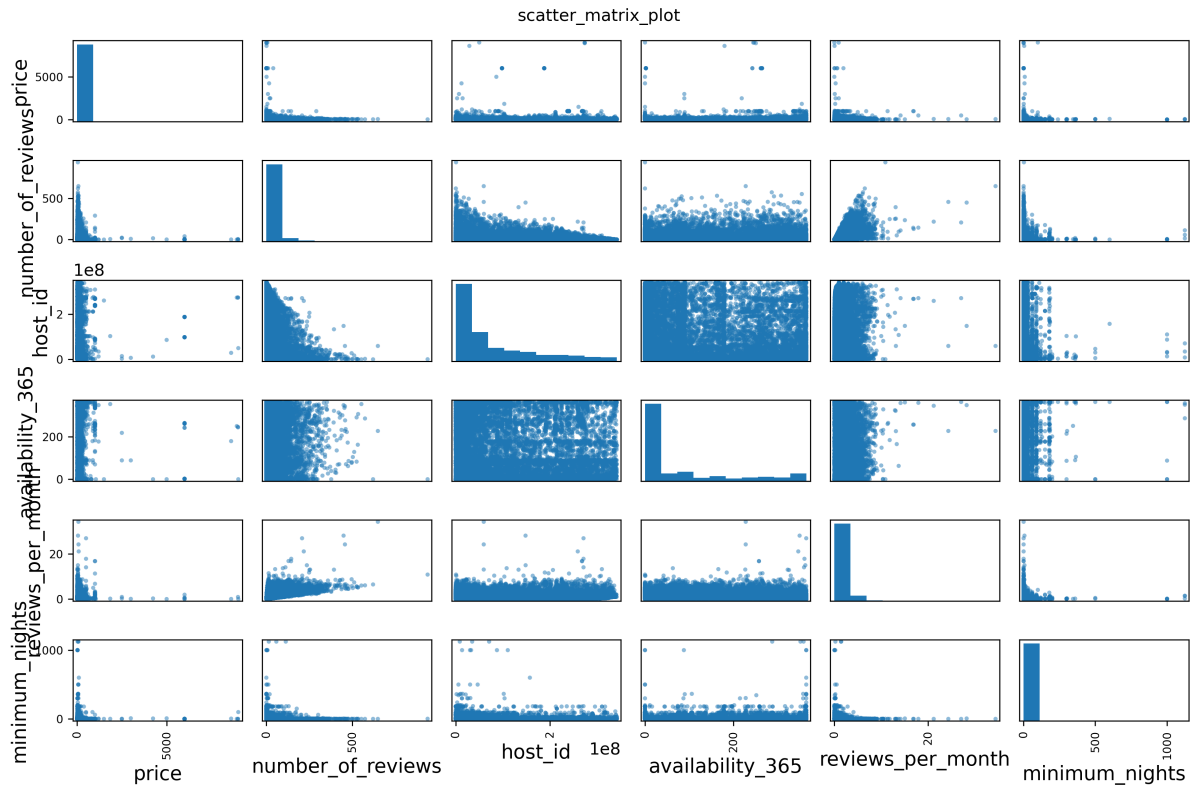


Figure 4.17: Small Data - AirBnB Scatter Matrix Plot

As I am interested in predicting the `price` I will calculate the correlation matrix in order to see, which column is most important:

```
corr_matrix = myDataset.corr()
print("correlation:\n", corr_matrix["price"].sort_values(ascending=False))
```

```
correlation:
price                             1.000000
availability_365                  0.096979
calculated_host_listings_count    0.077545
host_id                           0.045434
reviews_per_month                 0.034332
latitude                          0.007836
number_of_reviews                 0.000611
minimum_nights                   -0.006361
longitude                        -0.036490
Name: price, dtype: float64
```

4.3.4 Clean NULL-Values and Prepare for Machine Learning (3)

4.3.4.1 Find all NULL-Values

How many Non-Null rows are there? Use `print(myDataset.info())`

```
Data columns (total 14 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   name                            20088 non-null  object
 1   host_id                         20131 non-null  int64
 2   host_name                       20114 non-null  object
 3   neighbourhood_group             20131 non-null  object
 4   neighbourhood                   20131 non-null  object
 5   latitude                        20131 non-null  float64
 6   longitude                       20131 non-null  float64
 7   room_type                       20131 non-null  object
 8   minimum_nights                  20131 non-null  int64
 9   number_of_reviews               20131 non-null  int64
 10  last_review                     16501 non-null  object
 11  reviews_per_month               16501 non-null  float64
 12  calculated_host_listings_count  20131 non-null  int64
 13  availability_365                20131 non-null  int64
dtypes: float64(3), int64(5), object(6)
```

Are there NULL values in the columns? Use: `print(myDataset.isnull().any())`

```
name                             True
host_id                         False
host_name                        True
neighbourhood_group             False
neighbourhood                   False
latitude                        False
longitude                       False
room_type                       False
minimum_nights                  False
number_of_reviews               False
last_review                      True
reviews_per_month                True
calculated_host_listings_count  False
availability_365                False
```

And if you want to see these columns, then use:

```
myDataset[myDataset["reviews_per_month"].isnull()].head()
```

4.3.4.2 Remove all NULL-Values

There are different options for handling NULL values, which occur in a column:

Option 1: you can delete the entire row.

```
sample_incomplete_rows.dropna(subset=["total_bedrooms"])
```

Option 2: you can delete the entire column: Use

```
sample_incomplete_rows.drop("total_bedrooms", axis=1)
```

Option 3: you can fill these with an assumption, like for example the median (which is a often a good assumption for filling NULL values, but not always, as we have seen in the movies database example).

```
sample_incomplete_rows = myDataset[myDataset.isnull().any(axis=1)]
median = myDataset["reviews_per_month"].median()
sample_incomplete_rows["reviews_per_month"].fillna(median, inplace=True)
print("sample_incomplete_rows\n",
      sample_incomplete_rows['reviews_per_month'].head())
```

```
sample_incomplete_rows
24411    0.43
15700    0.43
9311     0.43
21882    0.43
24259    0.43
```

Here I can see, that the column “reviews_per_month” in my sample has been filled with the median value 0.43.

4.3.5 Model-Specific Preprocessing (4)

4.3.5.1 Use “Imputer” to Clean NULL-Values

Remove all text attributes because median can only be calculated on numerical attributes:

```
imputer = SimpleImputer(strategy="median")
myDataset_num = myDataset.select_dtypes(include=[np.number])
imputer.fit(myDataset_num)
print("\nimputer.strategy:", imputer.strategy)
print("\nimputer.statistics_\n", imputer.statistics_)
print("\nmyDataset_num.median\n", myDataset_num.median().values)
print("\nmyDataset_num.mean\n", myDataset_num.mean().values)
```

```
imputer.strategy: median
imputer.statistics_
 [3.9804571e+07 5.2509580e+01 1.3416280e+01 3.0000000e+00 5.0000000e+00
 4.3000000e-01 1.0000000e+00 0.0000000e+00]
myDataset_num.median
 [3.9804571e+07 5.2509580e+01 1.3416280e+01 3.0000000e+00 5.0000000e+00
 4.3000000e-01 1.0000000e+00 0.0000000e+00]
myDataset_num.mean
 [7.74925763e+07 5.25100259e+01 1.34059193e+01 7.20863345e+00
 2.17043863e+01 1.02009696e+00 2.43967016e+00 7.33118573e+01]
```

Transform the training set:

```
X = imputer.transform(myDataset_num)
myDataset_tr = pd.DataFrame(X,
                            columns=myDataset_num.columns,
                            index=myDataset.index)
myDataset_tr.loc[sample_incomplete_rows.index.values]
```

4.3.5.2 Treat "Categorial" Inputs

Often features are not given as numbers, but categorial. For example the column "room_type". We have different types of rooms, like for example "Private room", "Entire home/apt" and others. I want to use the OneHotEncoder for these categorial values. The OneHotEncoder[8] from the scikit-learn library can determine all different categories (here we have four different categories as shown below) and the OneHotEncoder can also convert these categorial values into numbers. For example: having the four categories

[8] Scikit-Learn - OneHotEncoder, https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

```
'Entire home/apt', 'Hotel room', 'Private room', 'Shared room'
```

a dataset with “Private room” in column “room_type” will become [0,0,1,0]. Another dataset where the “room_type” is “Entire home/apt” will become [1,0,0,0]. This is very easy to understand and interpret for machine.

```
myDataset_cat = myDataset[['room_type']]
print("myDataset_cat.head\n", myDataset_cat.head(10), "\n")
cat_encoder = OneHotEncoder()
myDataset_cat_1hot = cat_encoder.fit_transform(myDataset_cat)
print("cat_encoder.categories_:\n", cat_encoder.categories_)
print("myDataset_cat_1hot.toarray():\n", myDataset_cat_1hot.toarray())
print("myDataset_cat_1hot:\n", myDataset_cat_1hot)
```

```
            room_type
5177      Private room
19616  Entire home/apt
10275     Private room
...
8872      Private room

cat_encoder.categories_:
 [array(['Entire home/apt', 'Hotel room', 'Private room', 'Shared room'],
      dtype=object)]

myDataset_cat_1hot.toarray():
 [[0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]
 ...
 [0. 0. 1. 0.]]
```

4.3.6 Pipelines and Custom Transformer (5)

4.3.6.1 Custom Transformer

The function `add_extra_feature` serves (as the name already suggests) to add new features to the dataset. For example: if your dataset contains the length of a trip (measured in kilometers) and the time for this trip (measured in hours) then it would be obvious that a new feature where trip-length divided by time would be helpful. Or if your dataset contains the length and width and height, then it

would make sense to add an extra feature: the volume (which is the product length * width * height). Here I used `number_of_reviews` and `reviews_per_month` and created a new feature:

```
print("myDataset.columns\n", myDataset_num.columns)
number_of_reviews_ix, availability_365_ix,
calculated_host_listings_count_ix,
reviews_per_month_ix = [
    list(myDataset_num.columns).index(col)
    for col in ("number_of_reviews", "availability_365",
                "calculated_host_listings_count",
                "reviews_per_month")]

def add_extra_features(X):
    number_reviews_dot_revievs_per_month =
    X[:, number_of_reviews_ix] * X[:, reviews_per_month_ix]
    return np.c_[X, number_reviews_dot_revievs_per_month]

attr_adder = FunctionTransformer(add_extra_features,
                                 validate=False)
myDataset_extra_attribs = attr_adder.fit_transform(myDataset_num.values)

myDataset_extra_attribs = pd.DataFrame(
    myDataset_extra_attribs,
    columns=list(myDataset_num.columns)+ \
            ["number_reviews_dot_revievs_per_month"],
            index=myDataset_num.index)
print("myDataset_extra_attribs.head()\n",
      myDataset_extra_attribs.head())
```

4.3.6.2 Pipelines

Pipelines are used, when different preparations on a dataset have to be aligned. Defining these preparations as a pipeline will be useful to better structure all these processes and for more clarity in your coding. Here is how a pipeline on numerical attributes (columns, which contain numbers and categorial attributes (columns, which contain categorial values) can be defined:

```
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', FunctionTransformer(add_extra_features,
                                              validate=False)),
        ('std_scaler', StandardScaler())])
myDataset_num_tr = num_pipeline.fit_transform(myDataset_num)
```

```
print("myDataset_num_tr\n", myDataset_num_tr)

num_attribs = list(myDataset_num)
cat_attribs = ["room_type"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])
myDataset_prepared = full_pipeline.fit_transform(myDataset)
print("myDataset_prepared\n", myDataset_prepared)
```

4.3.7 Select and Train Model (6)

4.3.7.1 LinearRegression Model

Now we have a prepared dataset, `myDataset_prepared` and we are ready to select and train models. The first is a linear regression model. The quality / correctness of the model is measured with the root-mean-square error "rmse".

```
lin_reg = LinearRegression()
lin_reg.fit(myDataset_prepared, myDataset_labels)
some_data = myDataset.iloc[:10]
some_labels = myDataset_labels.iloc[:10]
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions:\n", lin_reg.predict(some_data_prepared))
print("Labels:\n", list(some_labels))

myDataset_predictions = lin_reg.predict(myDataset_prepared)
lin_mse = mean_squared_error(myDataset_labels, myDataset_predictions)
lin_rmse = np.sqrt(lin_mse)
print("lin_rmse\n", lin_rmse)
```

```
Predictions:
 [ 40.29259797  89.54082186  45.36959079 101.64252692  82.68120391
  89.57892837  76.33903757  76.42930129  38.55979179 861.9719368 ]
Labels:
 [41, 190, 35, 50, 100, 60, 69, 80, 32, 140]
lin_rmse
 213.51224401460684
```

These ten predictions are from the linear model and the labels are the real values. Some predictions are really close, but some are not satisfying. The root-mean-square-error on the whole testing dataset. In order to better understand, if this "rmse" acceptable or not, I will calculate the mean and the standard deviation of the testing dataset:

```
print("mean of labels:\n", myDataset_labels.mean())
print("std deviation of labels:\n", myDataset_labels.std())
```

```
mean of labels:
 74.19313496597287
std deviation of labels:
 227.66240520718222
```

I think we should try to find a better model.

4.3.7.2 DecisionTreeRegressor Model

Another model is the decision tree.

```
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(myDataset_prepared, myDataset_labels)
myDataset_predictions = tree_reg.predict(myDataset_prepared)

tree_mse = mean_squared_error(myDataset_labels, myDataset_predictions)
tree_rmse = np.sqrt(tree_mse)
print("tree_rmse\n", tree_rmse)
```

```
tree_rmse
 2.5907022436676304
```

The "rmse" for the decision tree regression is very low compared to the "rmse" for the linear regression. This looks great, but I want to cross-validate in a next step, to know, if this value is reliable or not.

4.3.8 Crossvalidation (7)

4.3.8.1 For DecisionTreeRegressor

Again I will apply the cross-validation as I already explained in sec. 4.1.8.

```
scores = cross_val_score(tree_reg, myDataset_prepared,
                         myDataset_labels,
                         scoring="neg_mean_squared_error",
                         cv=10)
tree_rmse_scores = np.sqrt(-scores)
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

```
Scores: [137.8762754  312.21141325 218.18522714 237.35937087  72.91732588
  65.62821113 304.05961303  86.65346214 178.337086   207.20918128]
Mean: 182.0437166129294
Standard deviation: 85.6479894222897
```

4.3.8.2 For LinearRegression

```
lin_scores = cross_val_score(lin_reg, myDataset_prepared,
                             myDataset_labels,
                             scoring="neg_mean_squared_error",
                             cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

```
Scores: [216.69779596 285.34262945 264.35619589 299.03183929 241.53524149
  78.73412537 208.49809202 186.28646055 147.62364622  91.09183577]
Mean: 201.91978620154234
Standard deviation: 72.64261021086485
```

We can see, that the linear regression model is not as good, as the decision tree model.

4.3.8.3 For RandomForestRegressor

```
forest_reg = RandomForestRegressor(n_estimators=10, random_state=42)
forest_reg.fit(myDataset_prepared, myDataset_labels)
```

```
myDataset_predictions = forest_reg.predict(myDataset_prepared)
forest_mse = mean_squared_error(myDataset_labels, myDataset_predictions)
forest_rmse = np.sqrt(forest_mse)
print("forest_rmse\n", forest_rmse)

forest_scores = cross_val_score(forest_reg, myDataset_prepared,
                                myDataset_labels,
                                scoring="neg_mean_squared_error",
                                cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
display_scores(forest_rmse_scores)
```

```
forest_rmse
 59.973965336421536
Scores: [157.89997398 130.24045184 208.33415063 235.2803762   93.2680475
  51.26602041 138.75095351 194.70999772 114.94817824 152.59500511]
Mean: 147.72931551364474
Standard deviation: 52.34950554724271
```

The random forest model is even better than the decision tree model.

4.3.9 Save Model (8)

I already explained, that saving a model can sounds trivial. But it can be of great consequence in the "deployment" (see CRISP-Cycle in fig. 3.1). Transferring a model and corresponding data (including the preliminary data cleaning and feature extraction work) from the development environment to the production environment can be topic, which you should think about carefully. Here is how you can save the model:

```
joblib.dump(forest_reg, "forest_reg.pkl")
# and later...
my_model_loaded = joblib.load("forest_reg.pkl")
```

4.3.10 Optimize Model (9)

4.3.10.1 GridSearchCV

In this step I want to know, if there better parameters for the models. Therefore I define a parameter grid and do a cross-validation on this grid. At the end I will know the best parameters, which is for the

random forest model `max_features=6` and `n_estimators=50`.

```
param_grid = [
    {'n_estimators': [30, 40, 50], 'max_features': [2, 4, 6, 8, 10]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3,
 ↪  4]}
]
forest_reg = RandomForestRegressor(random_state=42)
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(myDataset_prepared, myDataset_labels)

print("Best Params: ", grid_search.best_params_)
print("Best Estimator: ", grid_search.best_estimator_)
print("\nResults (mean_test_score and params):")
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
Best Params:  {'max_features': 6, 'n_estimators': 50}
Best Estimator:  RandomForestRegressor(max_features=6, n_estimators=50,
 ↪  random_state=42)

Results (mean_test_score and params):
141.45328245705397 {'max_features': 2, 'n_estimators': 30}
142.78195035217828 {'max_features': 2, 'n_estimators': 40}
142.37123716346338 {'max_features': 2, 'n_estimators': 50}
135.775474892488 {'max_features': 4, 'n_estimators': 30}
134.4190304245193 {'max_features': 4, 'n_estimators': 40}
135.55354286489987 {'max_features': 4, 'n_estimators': 50}
134.16110647203996 {'max_features': 6, 'n_estimators': 30}
134.1926275989663 {'max_features': 6, 'n_estimators': 40}
132.63913672411098 {'max_features': 6, 'n_estimators': 50}    <--- best
 ↪  parameters
136.15995027100854 {'max_features': 8, 'n_estimators': 30}
134.6124097344303 {'max_features': 8, 'n_estimators': 40}
133.2709123855618 {'max_features': 8, 'n_estimators': 50}
137.99385113493992 {'max_features': 10, 'n_estimators': 30}
137.62914189800856 {'max_features': 10, 'n_estimators': 40}
137.05278896563013 {'max_features': 10, 'n_estimators': 50}
```

```
163.96946102421438 {'bootstrap': False, 'max_features': 2,
                    'n_estimators': 3}
144.03648391473084 {'bootstrap': False, 'max_features': 2,
                    'n_estimators': 10}
149.43499632921868 {'bootstrap': False, 'max_features': 3,
                    'n_estimators': 3}
139.80926358472138 {'bootstrap': False, 'max_features': 3,
                    'n_estimators': 10}
143.39742710695754 {'bootstrap': False, 'max_features': 4,
                    'n_estimators': 3}
137.45096056556272 {'bootstrap': False, 'max_features': 4,
                    'n_estimators': 10}
```

4.3.10.1.1 GridSearchCV on RandomForestRegressor

```
param_grid = [
    {'fit_intercept': [True], 'n_jobs': [2, 4, 6, 8, 10]},
    {'normalize': [False], 'n_jobs': [3, 10]},
  ]
lin_reg = LinearRegression()
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
lin_grid_search = GridSearchCV(lin_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
lin_grid_search.fit(myDataset_prepared, myDataset_labels)

print("Best Params: ", lin_grid_search.best_params_)
print("Best Estimator: ", lin_grid_search.best_estimator_)
print("\nResults (mean_test_score and params):")
cvres = lin_grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
Best Params:  {'fit_intercept': True, 'n_jobs': 2}
Best Estimator:  LinearRegression(n_jobs=2)

Results (mean_test_score and params):
214.25154731008828 {'fit_intercept': True, 'n_jobs': 2}
214.25154731008828 {'fit_intercept': True, 'n_jobs': 4}
214.25154731008828 {'fit_intercept': True, 'n_jobs': 6}
```

```
214.25154731008828 {'fit_intercept': True, 'n_jobs': 8}
214.25154731008828 {'fit_intercept': True, 'n_jobs': 10}
214.25154731008828 {'n_jobs': 3, 'normalize': False}
214.25154731008828 {'n_jobs': 10, 'normalize': False}
```

4.3.10.1.2 GridSearchCV on LinearRegressor

4.3.10.2 Randomized Search

```
param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }
forest_reg = RandomForestRegressor(random_state=42)
rnd_search = RandomizedSearchCV(forest_reg,
                                param_distributions=param_distribs,
                                n_iter=10, cv=5,
                                scoring='neg_mean_squared_error',
                                random_state=42)
rnd_search.fit(myDataset_prepared, myDataset_labels)

print("Best Params: ", rnd_search.best_params_)
print("Best Estimator: ", rnd_search.best_estimator_)
print("\nResults (mean_test_score and params):")
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

```
Best Params:  {'max_features': 7, 'n_estimators': 180}
Best Estimator:  RandomForestRegressor(max_features=7, n_estimators=180,
 ↪  random_state=42)

Results (mean_test_score and params):
133.8480285550475 {'max_features': 7, 'n_estimators': 180}
136.57169310969803 {'max_features': 5, 'n_estimators': 15}
139.51632044685252 {'max_features': 3, 'n_estimators': 72}
137.1771768583531 {'max_features': 5, 'n_estimators': 21}
134.3830342778431 {'max_features': 7, 'n_estimators': 122}
139.33370530867487 {'max_features': 3, 'n_estimators': 75}
138.9071602851763 {'max_features': 3, 'n_estimators': 88}
```

```
135.5081501949792 {'max_features': 5, 'n_estimators': 100}
138.3278914625561 {'max_features': 3, 'n_estimators': 150}
184.05585159780387 {'max_features': 5, 'n_estimators': 2}
```

4.3.10.3 Analyze best models

```
feature_importances = grid_search.best_estimator_.feature_importances_
print("feature_importances:\n", feature_importances)
extra_attribs = ["number_reviews_dot_revievs_per_month"]
cat_encoder = full_pipeline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
print("\nattributes:\n", attributes)
my_list = sorted(zip(feature_importances, attributes), reverse=True)
print("\nMost important features (think about removing features):")
print("\n".join('{}' for _ in range(len(my_list))).format(*my_list))
```

```
feature_importances:
 [0.16069378 0.07746518 0.16413731 0.04363706 0.02335222 0.05394379
 0.10185993 0.17949253 0.04558588 0.00626952 0.13746286 0.00526594
 0.00083401]

attributes:
 ['host_id', 'latitude', 'longitude', 'minimum_nights', 'number_of_reviews',
 ↪  'reviews_per_month', 'calculated_host_listings_count',
 ↪  'availability_365', 'number_reviews_dot_revievs_per_month', 'Entire
 ↪  home/apt', 'Hotel room', 'Private room', 'Shared room']

Most important features (think about removing features):
(0.17949252977627858, 'availability_365')
(0.1641373056263463, 'longitude')
(0.16069378145971047, 'host_id')
(0.13746285698834157, 'Hotel room')
(0.10185992760999112, 'calculated_host_listings_count')
(0.07746518330470566, 'latitude')
(0.05394378731977807, 'reviews_per_month')
(0.04558587773390613, 'number_reviews_dot_revievs_per_month')
(0.043637064748633575, 'minimum_nights')
(0.023352218877592826, 'number_of_reviews')
(0.0062695183764503, 'Entire home/apt')
```

```
(0.0052659368947157795, 'Private room')
(0.0008340112835496431, 'Shared room')
```

4.3.11 Evaluate final model on test dataset (10)

```
final_model = grid_search.best_estimator_
print("final_model:\n", final_model)

X_test = strat_test_set.drop("price", axis=1)
y_test = strat_test_set["price"].copy()

X_test_prepared = full_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_prepared)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)

print ("final_predictions:\n", final_predictions )
print ("final_rmse:\n", final_rmse )

confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
mean = squared_errors.mean()
m = len(squared_errors)

# from scipy import stats
print("95% confidence interval: ",
      np.sqrt(stats.t.interval(confidence, m - 1,
                          loc=np.mean(squared_errors),
                          scale=stats.sem(squared_errors))))
```

```
final_model:
 RandomForestRegressor(max_features=6, n_estimators=50, random_state=42)
final_predictions:
 [132.24  78.2   96.3  ...  75.66  51.2   47.04]
final_rmse:
 112.12588420446276
95% confidence interval:  [ 53.7497262  149.18242105]
```

4.4 "Big Data": Machine Learning using Spark ML Library

This will be an example for a "Big-Data"[9] environment and uses the *Apache Spark MLib*[10] scalable machine learning library.

Various tutorials, documentation, "code-fragments" and guidelines can be found in the internet **for free** (at least for your private and non-commercial use). The best is in my opinion the official documentation [11]. A few more helpful sources are the following GitHub repositories:

- tirthajyoti/Spark-with-Python[12] (MIT Licence)
- Apress/learn-pyspark[13] (Freeware License)
- mahmoudparsian/pyspark-tutorial[14] (Apache License v2.0)

Concerning the "Big Data" topic I want to add the following: I passed a certification as *"Data Scientist Specialized in Big Data Analytics"* at Fraunhofer Institute. I must say: Understanding the concept of "Big-Data" and how to differentiate "standard" machine learning from a "scalable" Big Data environment is not easy. I recommend a separate training!

Some steps are a bit similar to "Scikit-Learn" (e.g. data-cleaning, preprocessing), but the technical environment for running the code is different and also the code itself is different. For this reason I added a separate chapter, see sec. 5 which covers the topics *"Map-Reduce"* [15] (one of the powerful programming models for Big Data), *"K-Means-Clustering in Spark"* and a Min-Hash Example[16].

Let's start with the "Big Data" structure, which I put into a mind map, see fig. 4.18. This structure is aligned to the "Small Data" Mind Map (Scikit-Learn, see fig. 4.11) in order to compare these two approaches. You will find the same structure in the Jupyter-Notebook on my Docker repository, which you can download and it should be a guide to work out your own problems with the same structure.

[9] Big-Data, Wikipedia, https://en.wikipedia.org/wiki/Big_Data

[10] Apache Spark MLib, https://spark.apache.org/mllib/

[11] Official Apache Spark ML documentation, https://spark.apache.org/docs/latest/ml-guide.html

[12] GitHub repository "tirthajyoti/Spark-with-Python", https://github.com/tirthajyoti/Spark-with-Python (MIT Licence)

[13] GitHub repository "Apress/learn-pyspark", https://github.com/Apress/learn-pyspark (Freeware License)

[14] GitHub repository "mahmoudparsian/pyspark-tutorial", https://github.com/mahmoudparsian/pyspark-tutorial (Apache License v2.0)

[15] Map Reduce, Wikipedia, https://en.wikipedia.org/wiki/MapReduce

[16] Minhashing and Local-Sensitive Hashing Example, https://github.com/AndreasTraut/Deep_learning_explorations

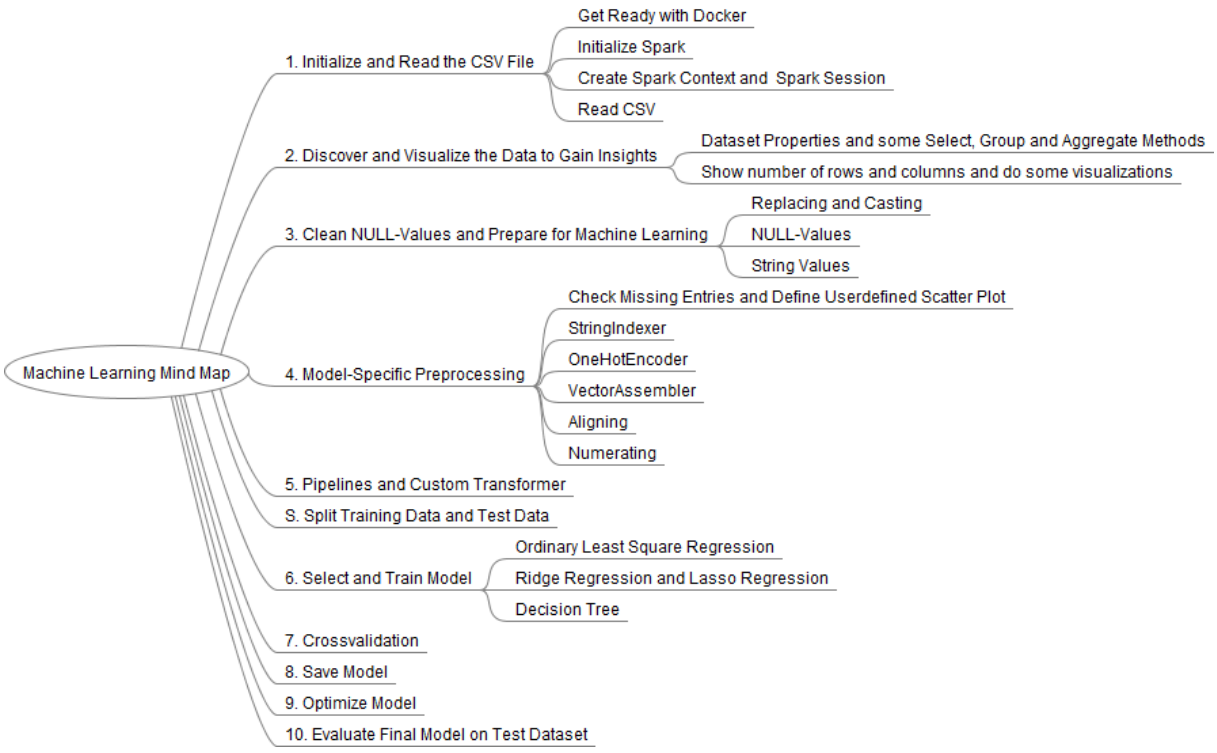


Figure 4.18: Machine Learning Mind Map - "Big Data" with Apache Spark ML

4.4.1 Initialize and Read the CSV File (1)

4.4.1.1 Get Ready with Docker

There are different ways to get ready and started with the Apache Spark and Hadoop environment:

- I guess, that you can install it on your own computer (which I found very difficult because of lack of user-friendly and easy understandable documentation).
- Or you can dive into a Cloud environment, like e.g. Microsoft Azure or Amazon EWS or Google Cloud and try to get a virtual machine up and running for your purposes. Have a look at my documentation[17], where I shared my experiences, which I had with Microsoft Azure: getting started with a Cloud service is not easy!
- Or you can use Docker[18], which I did. What is Docker? Docker is *"an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux."* I recommend learning from the Docker-Curriculum[19] what docker container and docker images are and how it works. Despite the difficult subject matter it is a nice tutorial.

[17] My experiences with Microsoft Azure, https://github.com/AndreasTraut/Experiences-with-MicrosoftAzure
[18] Docker, https://www.docker.com/
[19] Docker-Curriculum, https://docker-curriculum.com/

For the following explanation I decided to use Docker. I found a container, which had *Apache Spark Version 3.0.0* and *Hadoop 3.2* installed and built my machine-learning code (using pyspark) on top of this container. I shared my code and developments on Docker-Hub in the my docker machine-learning repository[20]. After having installed the Docker application you will need to pull my "machine-learning-pyspark" image to your computer. Open a command shell (type `cmd` on a Windows computer) and enter the following command:

```
docker pull andreastraut/machine-learning-pyspark
```

Then type the following:

```
docker run -dp 8888:8888 andreastraut/machine-learning-pyspark:latest
```

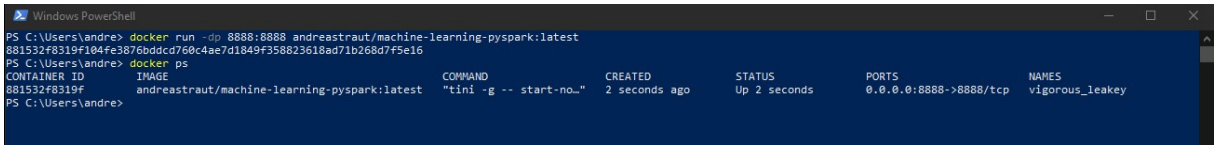


Figure 4.19: Big Data - Run Docker

You will see in your Docker Dashboard that a container is running:

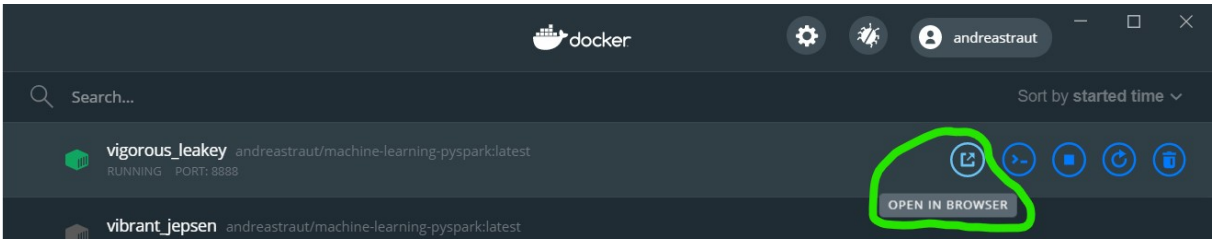


Figure 4.20: Big Data - Docker Dashbord

After having opened your browser (e.g. Chrome, Edge or Firefox Browser), navigate to `localhost:8888` (8888 is the port, which will be opened).

[20] My docker machine-learning repository, https://hub.docker.com/repository/docker/andreastraut/machine-learning-pyspark

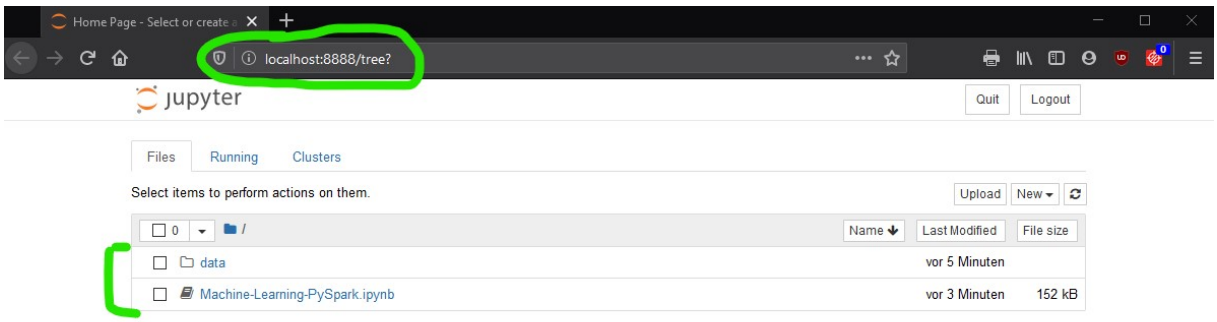


Figure 4.21: Big Data - Docker Localhost

The folder "data" contains the datasets. If you would like to do further analysis or produce alternate visualizations of the Airbnb-data, you can download them from Inside-AirBnB[21]. It is available below under consideration of the *Creative Commons CC0 1.0 Universal (CC0 1.0) Public Domain Dedication Licence*[22]. The data for the Vermont-Vendor-Payments[23] is available under condieration of the Open Data Commons Open Database License[24].

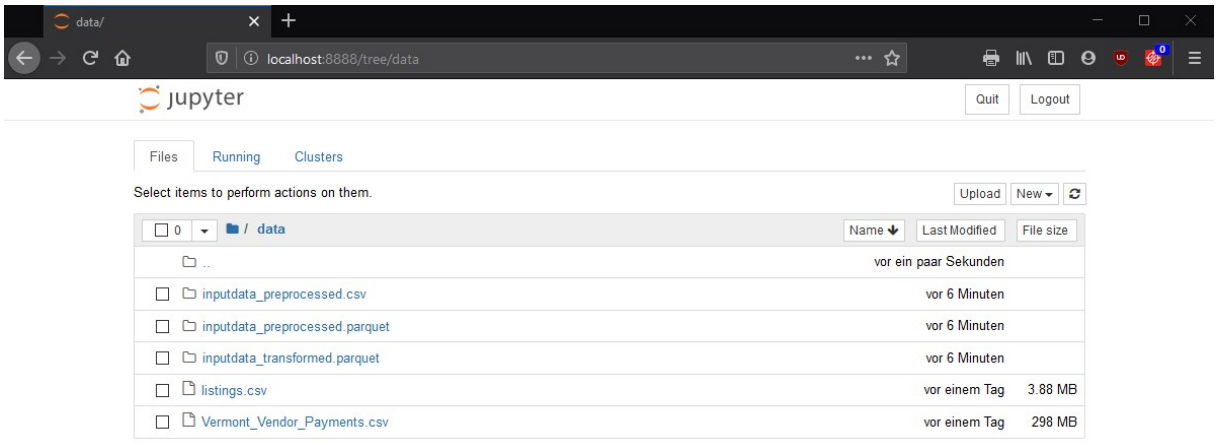


Figure 4.22: Big Data - Docker Localhost Datafiles

When you open the Jupyter-Notebook, you will see, that Apache Spark Version 3.0.0 and Hadoop Version 3.2 is installed:

[21] Inside-AirBnB, http://insideairbnb.com/get-the-data.html

[22] Creative Commons 1.0 Universal Public Domain Dedication Licence, http://creativecommons.org/publicdomain/zero/1.0/

[23] "Vermont_Vendor_Payments.csv", https://data.vermont.gov/Finance/Vermont-Vendor-Payments/786x-sbp3

[24] Open Data Commons Open Database License , http://opendatacommons.org/licenses/odbl/1.0/

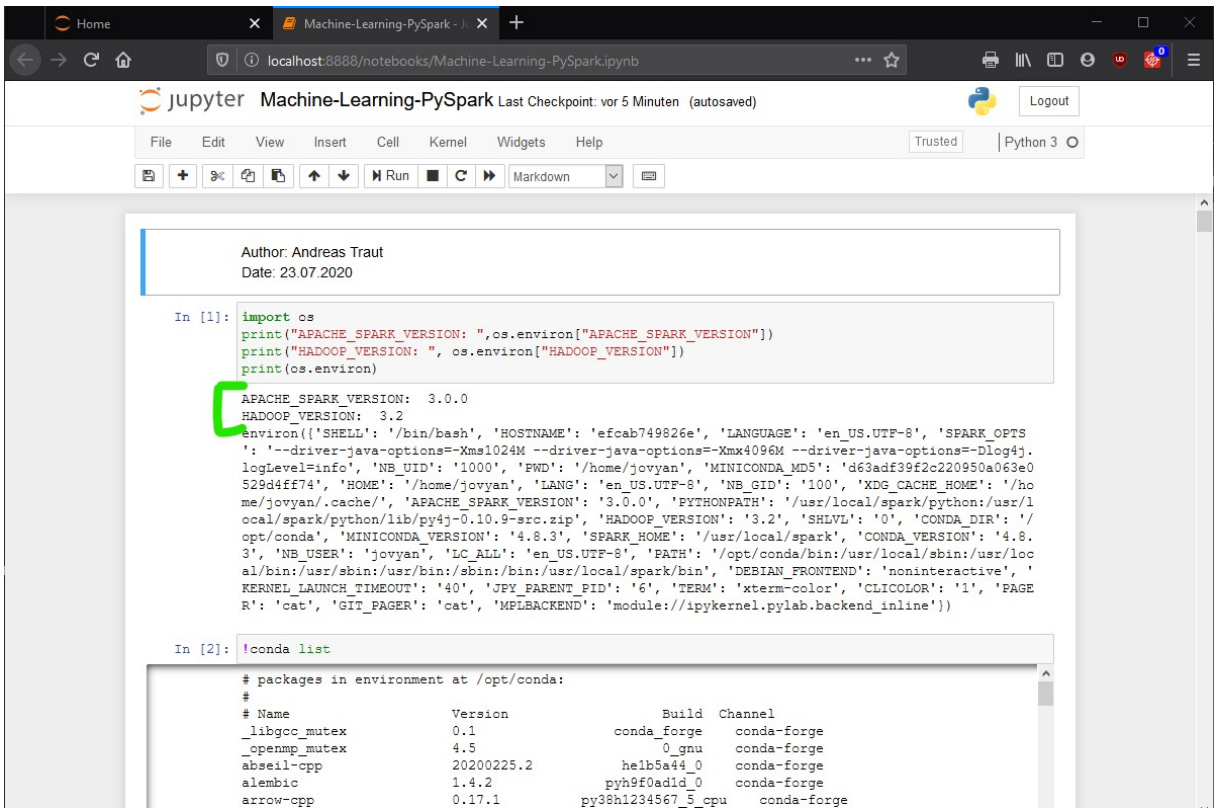


Figure 4.23: Big Data - Docker Jupyter Notebook

4.4.1.2 Initialize Spark

I recommend to read the Apache Spark "Get Started Guide"[25]. Initializing a Spark sessions and read a CSV file (see also the official Spark documentation[26]).

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
```

4.4.1.3 Create Spark Context and Spark Session

```
sc = pyspark.SparkContext(appName='Spark Modelling Context')
spark = SparkSession.builder \
```

[25] Apache Spark "Get Started Guide", http://spark.apache.org/docs/latest/quick-start.html

[26] Official Spark documentation, https://spark.apache.org/docs/latest/sql-getting-started.html#starting-point-sparksession

```
        .appName('Spark Modelling Session') \
        .config('spark.executor.memory','5g') \
        .config('spark.executor.cores','4') \
        .getOrCreate()
```

4.4.1.4 Read CSV

```
datapath = os.environ['PWD']
filename = datapath + "/data/listings.csv"
data = spark.read.csv(path=filename,
                      sep=',', encoding='utf-8',
                      header=True, inferSchema=True)
data.cache()
```

```
DataFrame[id: string, name: string, host_id: string, host_name: string,
 ↪  neighbourhood_group: string, neighbourhood: string, latitude: string,
 ↪  longitude: string, room_type: string, price: string, minimum_nights:
 ↪  int, number_of_reviews: string, last_review: string, reviews_per_month:
 ↪  string, calculated_host_listings_count: double, availability_365: int]
```

4.4.2 Discover and Visualize the Data to Gain Insights (2)

4.4.2.1 Dataset Properties and some Select, Group and Aggregate Methods

Before we start cleaning the data, we want to have a first glance at the columns and rows in order to learn, what is special in the dataset. With the following command you will get some properties (like count, mean, stddev, min, max).

```
data.describe().show(truncate=10)
```

Similarly to the following command you can also get the distinct count value of other columns:

```
data.select(F.countDistinct('host_name')).show()
```

```
+------------------------+
|count(DISTINCT host_name)|
+------------------------+
```

```
|                      6483|
+--------------------------+
```

Grouping works as follows:

```
data.groupBy('host_name').count().sort('count', ascending=False).show(n=5,
 ↪  truncate=False)
```

```
+---------+-----+
|host_name|count|
+---------+-----+
|Anna     |207  |
|Julia    |180  |
|Daniel   |166  |
|Michael  |161  |
|David    |145  |
+---------+-----+
only showing top 5 rows
```

4.4.2.2 Show number of rows and columns and do some visualizations

```
data.printSchema()
```

```
root
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- host_id: string (nullable = true)
 |-- host_name: string (nullable = true)
 |-- neighbourhood_group: string (nullable = true)
 |-- neighbourhood: string (nullable = true)
 |-- latitude: string (nullable = true)
 |-- longitude: string (nullable = true)
 |-- room_type: string (nullable = true)
 |-- price: string (nullable = true)
 |-- minimum_nights: integer (nullable = true)
 |-- number_of_reviews: string (nullable = true)
 |-- last_review: string (nullable = true)
 |-- reviews_per_month: string (nullable = true)
 |-- calculated_host_listings_count: double (nullable = true)
```

```
data.describe().show(truncate=False, n=1, vertical=True)
```

```
-RECORD 0------------------------------
 summary                           | count
 id                                | 25206
 name                              | 25146
 host_id                           | 25163
 host_name                         | 25141
 neighbourhood_group               | 25163
 neighbourhood                     | 25163
 latitude                          | 25163
 longitude                         | 25163
 room_type                         | 25163
 price                             | 25163
 minimum_nights                    | 25163
 number_of_reviews                 | 25158
 last_review                       | 20690
 reviews_per_month                 | 20641
 calculated_host_listings_count    | 25109
only showing top 1 row
```

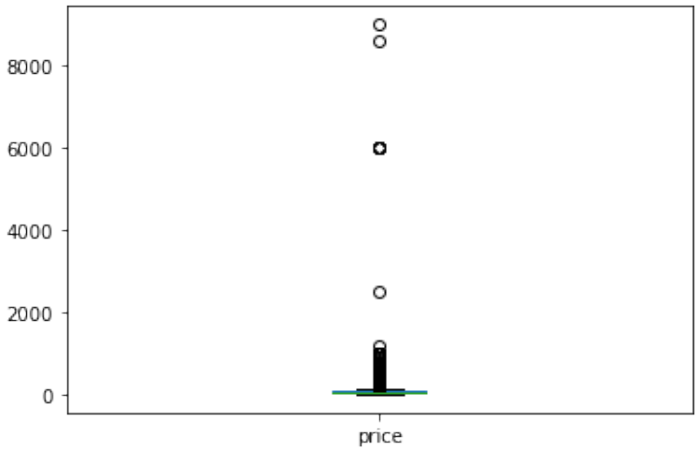


Figure 4.24: Big Data - Price Box Plot

```
data = data.withColumn('price',
                       data['price'].cast('double'))
data.sample(fraction=0.3,
            seed=42,
            withReplacement=False) \
```

```
    .sort('price', ascending=False) \
    .select('price').toPandas().plot.box();
```

4.4.3 Clean NULL-Values and Prepare for Machine Learning (3)

4.4.3.1 Replacing and Casting

First I will cast the price into a `double` value:

```
data = data.withColumn('price', data['price'].cast('double'))
```

There is a very useful function available for replacing characters: `regexp_replace`. This is how it works:

```
data = data.withColumn('price', F.regexp_replace('price','\$',''))
data = data.withColumn('price', F.regexp_replace('price',',',''))
```

4.4.3.2 NULL-Values

Replacing NULL values is a very common problem. As I want to predict the `price` I will have a look into these NULL values first:

```
print("{} missing values for price"
      .format(data
              .filter(F.isnull(data['price']))
              .count()))
```

```
104 missing values for price
```

As mentioned in previous examples, there are different possibilities to replace NULL values: you could eliminate the row or fill the NULL values with an approximate value, like the median. In this case I decide to fill the NULL values with 0.

```
data = data.fillna(0, subset='price')
```

```
print("{} missing values for
 ↪  price".format(data.filter(F.isnull(data['price'])).count()))
```

```
0 missing values for price
```

4.4.3.3 String Values

```
string_types = [x[0]
                for x in data.dtypes
                if x[1] == 'string']
data.select(string_types).describe().show()
```

4.4.4 Model-Specific Preprocessing (4)

4.4.4.1 Check Missing Entries and Define User-Defined Scatter Plot

It is always a good idea to define functions for procedures, which you need more often. This user-defined scatter plot is very helpful:

```
data.select(*(F.sum(F.col(c).isNull() \
                    .cast("int")).alias(c) for c in data.columns)) \
    .toPandas() \
    .transpose()

import numpy as np
import matplotlib.pyplot as plt

def scatter_predicted_vs_actual_price(model, data):
    prediction = np.array([p[0]
                           for p in model.transform(data). \
                           select('prediction').collect()]).flatten()
    truth      = np.array([p[0] for p in model.transform(data).\
                           select('label').collect()]).flatten()
    fig, ax = plt.subplots(figsize=(5, 5))
    ax.scatter(prediction, truth,
               facecolor='steelblue',
               s=30, alpha=0.6)
    ax.set_xlabel('Predicted price', fontsize=12)
    ax.set_ylabel('Actual price', fontsize=12)
```

```
    m = max(max(prediction), max(truth))
    ax.plot([0,m], [0,m], color='grey')
    ax.set_xlim([0,m])
    ax.set_ylim([0,m])
    plt.show()
```

4.4.4.2 StringIndexer

Similarly to what I did in the "Small Data" example in sec. 4.3.5.2 I want to encode the column "room_type", which contains strings, to numbers. But in order to explain something new, I will not use the "OneHotEncoder" but another interesting possibility, which is useful to treat the "convert string to numbers" problem. I will use the function "StringIndexer"[27] from the Apache Spark Machine-Learning Library which can also convert strings to numbers, but a bit differently.

```
from pyspark.ml.feature import StringIndexer
room_indexer = StringIndexer(inputCol='room_type',
                             outputCol='room_index')
room_indexer_model = room_indexer.fit(data)
data = room_indexer_model.transform(data)

data.groupby('room_type'). \
        agg(F.collect_set('room_index'). \
        alias('room_index')). \
        sort('room_type', ascending=False).show(4)
```

```
+---------------+----------+
|      room_type|room_index|
+---------------+----------+
|    Shared room|     [2.0]|
|   Private room|     [1.0]|
|     Hotel room|     [3.0]|
|Entire home/apt|     [0.0]|
+---------------+----------+
only showing top 4 rows
```

As you can see I have only one new column "room_index" and not one column per category, which would have been four new columns in this case. The new column "room_index" contains one number, which corresponds to the type of room.

[27] "StringIndexer", https://spark.apache.org/docs/latest/ml-features.html#stringindexer

4.4.4.3 OneHotEncoder

As already explained in the "Small Data" example in sec. 4.3.5.2 I will apply the "OneHotEncoder"[28] to the column "neighbourhood_group_index" (I applied the "StringIndexer" on the string column "neighbourhood_group" and obtained the "neighbourhood_group_index" column with numbers). Similarly to what I showed in sec. 4.3.5.2 the result are several new columns for each category and the corresponding 1 and 0 to encode the original string.

```
from pyspark.ml.feature import OneHotEncoder
one_hot_encoder = OneHotEncoder(inputCol='neighbourhour_group_index',
                                outputCol='one_hot_neighbourhood_group',
                                dropLast = False)
one_hot_encoder_model = one_hot_encoder.fit(data)
data = one_hot_encoder_model.transform(data)
data.select('neighbourhood_group_index',
 ↪  'one_hot_neighbourhood_group').show(5)
```

```
+------------------------+--------------------------+
|neighbourhood_group_index|one_hot_neighbourhood_group|
+------------------------+--------------------------+
|                     2.0|             (42,[2],[1.0])|
|                     5.0|             (42,[5],[1.0])|
|                     0.0|             (42,[0],[1.0])|
|                     2.0|             (42,[2],[1.0])|
|                     2.0|             (42,[2],[1.0])|
+------------------------+--------------------------+
only showing top 5 rows
```

4.4.4.4 VectorAssembler

The "VectorAssembler"[29] combines the transformations from above. Given a list of columns it will create a single feature vector, which contains all the information from the previous transformations.

```
from pyspark.ml.feature import VectorAssembler
data = data.withColumn('number_of_reviews',
                       data['number_of_reviews'].cast('double'))
numeric_attributes = ['number_of_reviews']
vec_num = VectorAssembler(inputCols=numeric_attributes,
                          outputCol='num_features')
```

[28] "OneHotEncoder", https://spark.apache.org/docs/latest/ml-features.html#onehotencoder

[29] "VectorAssembler", https://spark.apache.org/docs/latest/ml-features.html#vectorassembler

```
data = vec_num.transform(data)
vec_label = VectorAssembler(inputCols = ['price'],
                            outputCol = 'vec_label')
data = vec_label.transform(data)
```

4.4.4.5 Aligning

Aligning in this context means, that you use the variable `label` for the labeled column (here it is the "price" column) and the variable `feature_cols` for the features.

```
label = 'price'
feature_cols = ['num_features',
                'vec_label']
cols = feature_cols + ['label']
data_feat = data.withColumnRenamed(label,'label').select(cols)
```

```
+------------+---------+-----+
|num_features|vec_label|label|
+------------+---------+-----+
|     [145.0]|   [90.0]| 90.0|
|      [27.0]|   [28.0]| 28.0|
|     [133.0]|  [125.0]|125.0|
|     [292.0]|   [33.0]| 33.0|
|       [8.0]|  [180.0]|180.0|
+------------+---------+-----+
only showing top 5 rows
```

4.4.4.6 Numerating

```
feat_len = len(numeric_attributes) + \
        data.select('room_type','room_index').distinct().count()
features = numeric_attributes  + \
        [r[0] for r in
         data.select('room_type','room_index').distinct().collect()]
feature_dict = dict(zip(range(0, feat_len), features))
```

4.4.5 Pipelines and Custom Transformer (5)

```
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=[vec_num,
                            vec_label])
pipeline_model = pipeline.fit(data)
```

If you want to work with another pipeline, then all you need is to replace the second line. For example into this:

```
pipeline = Pipeline(stages=[room_indexer,
                            vec_num,
                            vec_label])
```

4.4.6 S. Split Training Data and Test Data (S)

```
data_train, data_test = data_feat.randomSplit([0.9,0.1],
                                              seed=42)
```

This creates a random split into a training dataset `data_train` and the testing dataset `data_test`. Remember, that in the movies data example (see sec. 4.1) we used the stratified sample.

4.4.7 Select and Train Model (6)

4.4.7.1 Ordinary Least Square Regression

After having extracted, transformed and selected features we will apply some models, for example the "OLS Regression"[30]. Please read the official documentation for learning more about OLS.

```
from pyspark.ml.regression import LinearRegression
lr = LinearRegression(featuresCol='num_features',
                      labelCol='label',
                      maxIter=1000,
                      fitIntercept=True)
lr_model = lr.fit(data_train)
```

[30] "OLS Regression", https://spark.apache.org/docs/latest/ml-classification-regression.html#linear-regression

```
lr_model.coefficients
pred = lr_model.transform(data_test)
pred.select('label','prediction').show(5)
```

Evaluate the results:

```
from pyspark.ml.evaluation import RegressionEvaluator
re = RegressionEvaluator(metricName='rmse')
rmse = re.evaluate(pred)
print(rmse)
scatter_predicted_vs_actual_price(lr_model,
                                  data_test)
```

4.4.7.2 Ridge Regression and Lasso Regression

The coding syntax for the "Ridge Regression" and the "Lasso Regression" are pretty similar (there are also the `fit` and the `transform` methods). Please read the official documentation in order to understanding the "Ridge Regression and the Lasso Regression"[31] in detail.

4.4.7.3 Decision Tree

The coding syntax for the "Decision Tree" is pretty similar (there is also the `fit` and the `transform` method). Please read the official documentation in order to understanding the "Decision Tree"[32] in detail.

4.4.8 Save Model

If you want to save your intermediate you can do it as follows:

```
data.select(*data.columns[:-1]).write. \
    format("parquet").save("data/inputdata_preprocessed.parquet",
                           mode='overwrite')
data.select(*data.columns[:-1]).write. \
    csv('data/inputdata_preprocessed.csv',
        mode='overwrite', header=True)
```

[31] Ridge Regression and the Lasso Regression, https://spark.apache.org/docs/latest/mllib-linear-methods.html#linear-least-squares-lasso-and-ridge-regression

[32] Decision Tree, https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-trees

Reading works as follows:

```
filename = "data/inputdata_preprocessed.parquet"
data = spark.read.parquet(filename)
```

4.5 Summary Mind Map

In fig. 4.25 you can see my common Machine Learning Mind Map, which (who would have thought it) works for "Small Data" (Scikit-Learn) as well for "Big Data" (Apache Spark ML). I recommend to use it (or another of your own choice), in order to structure your work efficiently. The more you develop your code by following a common structure link mine, the easier it will become for you to adapt already existing code to new problems.

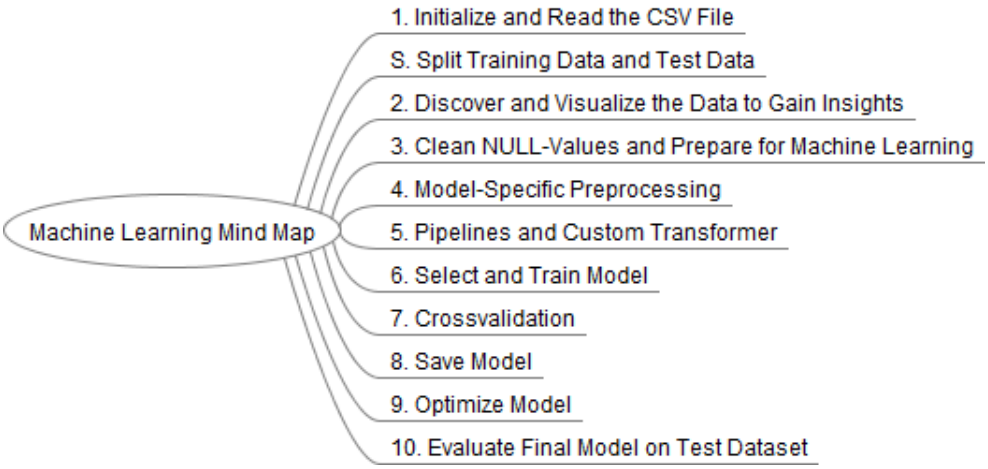


Figure 4.25: Machine Learning Mind Map - "Small Data" and "Big Data"

4.6 Future learnings and coding & data sources

For all of these topics various tutorials, documentation, coding examples and guidelines can be found in the internet **for free**! The Open Source Community is an incredible treasure trove and enrichment that positively drives many digital developments: Scikit-Learn, Apache Spark, Spyder, GitHub, Tensorflow and also Firefox[33], Signal-Messenger[34], Threema-Messenger[35], Corona-Warnapp[36], … to be mentioned. There are many positive examples of sharing code and data "for free".

[33] Firefox, https://github.com/mozilla
[34] Signal-Messenger, https://github.com/signalapp
[35] Threema-Messenger, https://github.com/threema-ch
[36] Corona-Warnapp, https://github.com/corona-warn-app

Coding:

If you Google for example *"how to prepare and clean the data with spark"*, you will find tons of documents around *"removing null values"* or *"encoders"* (like the "OneHotEncoder" for treating categorical inputs) or "Pipelines" (for putting all the steps in an efficient, customizable order) so on. You will be overwhelmed of all this. Some resources to mention are the official Apache Spark ML documentation and a few more Github repositories like the ones, which I mentioned already (tirthajyoti/Spark-with-Python (MIT Licence), Apress/learn-pyspark (Freeware License), mahmoudparsian/pyspark-tutorial (Apache License v2.0).

Data:

If you would like to do further analysis or produce alternate visualizations of the Airbnb-data, you can download them under consideration of the *Creative Commons 1.0 Universal Public Domain Dedication Licence*[37]. The data for the Vermont-Vendor-Payments can be downloaded under consideration of the *Open Data Commons Open Database License* [38]. The movies database doesn't even mention a license. There you find a lot of more datasets and also coding examples for your studies.

[37] Creative Commons 1.0 Universal Public Domain Dedication Licence, http://creativecommons.org/publicdomain/zero/1.0/

[38] Open Data Commons Open Database License , http://opendatacommons.org/licenses/odbl/1.0/

5 Big Data: Map-Reduce and K-Means Clustering

5.1 Map-Reduce Example

Map-Reduce is a programming model for generating big data sets with parallel distributed algorithm on a cluster. This is very important for Big Data and therefore I added some examples in order to explain how Map-Reduce works: the first is the "Word Count Example" (see sec. 5.1.1) for learning the basics of Map-Reduce and the second is the "TF-IDF Example" (sec. 5.1.2) for applying Map-Reduce on an interesting text-mining problem.

Please learn the basis of the Map-Reduce programming model for example from Wikipedia[1] before continuing reading.

5.1.1 Word Count Example

The "Word Count" example is one of the easiest for explaining Map-Reduce: given a text as input the aim is to return a list with words and the number of occurrences of each of these words. I used the "Moby Dick" as input text:

First we start a spark session and open the text file:

```
from pyspark import SparkContext
sc=SparkContext(master="local[4]")
text_file = sc.textFile("data/Moby-Dick.txt")
type(text_file)
```

```
pyspark.rdd.RDD
```

The text is now available in `text_file`, which is a resilient distributed dataset (RDD)[2]. RDD is a fault-tolerant collection of elements (here the lines of words), that can be operated on in parallel.

[1] Map Reduce, Wikipedia, https://en.wikipedia.org/wiki/MapReduce

[2] Resilient Distributed Dataset (RDD), https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds

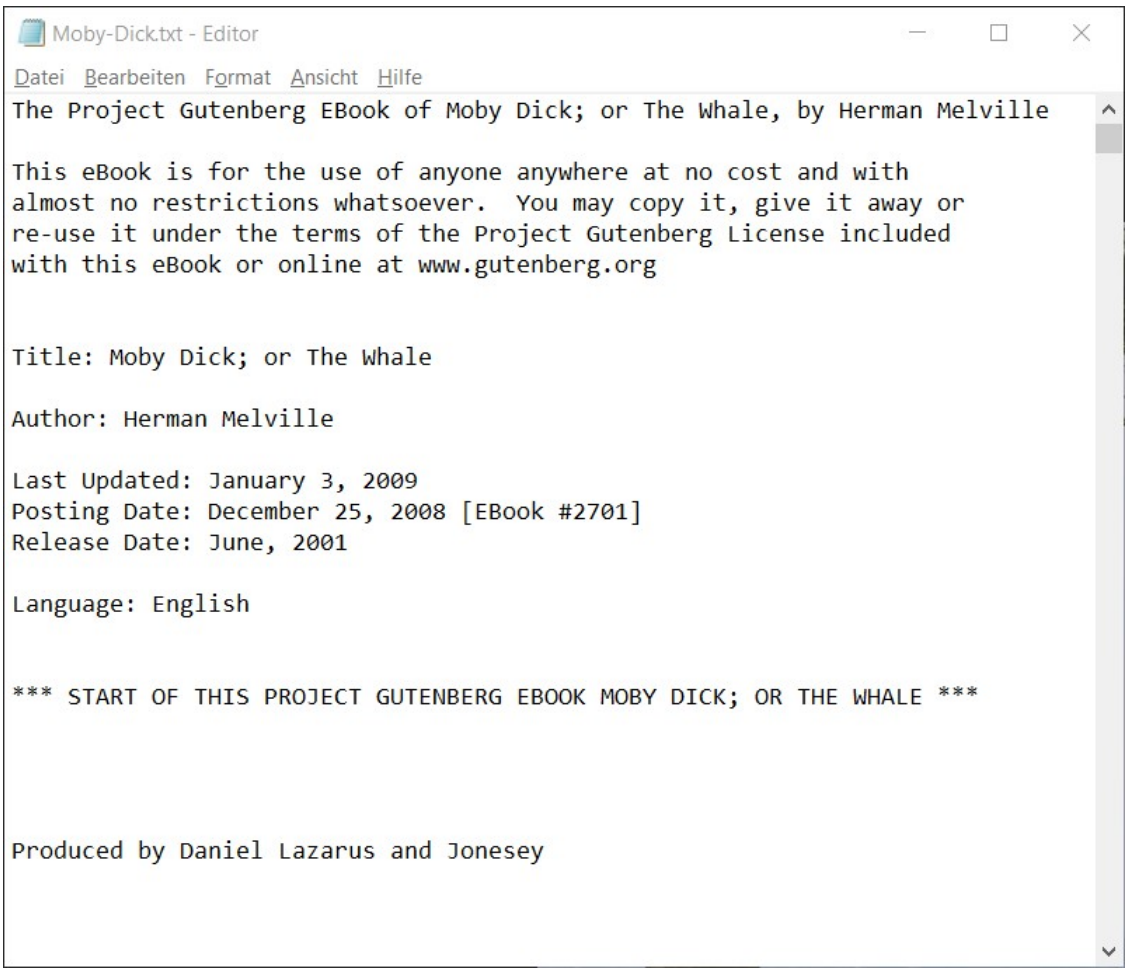


Figure 5.1: Map-Reduce - Word Count Example "Moby Dick"

Let's have a look into the `text_file`:

```
print(text_file.collect())
```

```
['The Project Gutenberg EBook of Moby Dick; or The Whale, by Herman
 ↪  Melville', '', 'This eBook is for the use of anyone anywhere at no cost
 ↪  and with', 'almost no restrictions whatsoever.  You may copy it, give it
 ↪  away or', 're-use it under the terms of the Project Gutenberg License
 ↪  included', 'with this eBook or online at www.gutenberg.org', '', '',
 ↪  'Title: Moby Dick; or The Whale', '', 'Author: Herman Melville', '',
 ↪  'Last Updated: January 3, 2009', 'Posting Date: December 25, 2008 [EBook
 ↪  #2701]', 'Release Date: June, 2001', '', 'Language: English', '', ''
```

As we can see each line is separated with a comma, as follows: 'line1', 'line2', ' line3', …

In the next steps we will

1. split the lines by using the spaces as separators
2. eliminate the empty elements
3. **map** the words to a tuple `(word, 1)`
4. **reduce** by key in order to count the number of occurances of each word (which is `a, b` will be transformed to `a + b`)

The python code is as follows:

```
#1:
words = text_file.flatMap(lambda line: line.split(" "))

#2:
not_empty = words.filter(lambda x: x!='')

#3:
key_values = not_empty.map(lambda word: (word, 1))

#4:
counts = key_values.reduceByKey(lambda a, b: a + b)
```

I will explain each of these four steps in more detail now:

5.1.1.1 Split the lines by using the spaces as separators (#1)

With the following line of code each line is split up into the words:

```
words = text_file.flatMap(lambda line: line.split(" "))
```

As a word is separated from another word with a blank, we have to use " " as a separator.

```
print(words.take(35))
```

```
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Moby', 'Dick;', 'or', 'The',
 ↪  'Whale,', 'by', 'Herman', 'Melville', '', 'This', 'eBook', 'is', 'for',
 ↪  'the', 'use', 'of', 'anyone', 'anywhere', 'at', 'no', 'cost', 'and',
 ↪  'with', 'almost', 'no', 'restrictions', 'whatsoever.', '', 'You']
```

We can find some empty entries `...,'','',...` which we will filter out in the next step. But before we do this we want to understand why `flatMap` has been used here instead of `map`:

The reason is that the operation `line.split(" ")` generates a **list** of strings, so had we used `map` the result would be an RDD of lists of words. Not an RDD of words.

The difference between `map` and `flatMap` is that the second expects to get a list as the result from the map and it **concatenates** the lists to form the RDD.

5.1.1.2 Eliminate the empty elements (#2)

The next line of code is the following:

```
not_empty = words.filter(lambda x: x!='')
```

This will filter in order to get rid of all empty entries.

```
print(not_empty.take(35))
```

```
['The', 'Project', 'Gutenberg', 'EBook', 'of', 'Moby', 'Dick;', 'or', 'The',
 ↪  'Whale,', 'by', 'Herman', 'Melville', 'This', 'eBook', 'is', 'for',
 ↪  'the', 'use', 'of', 'anyone', 'anywhere', 'at', 'no', 'cost', 'and',
 ↪  'with', 'almost', 'no', 'restrictions', 'whatsoever.', 'You', 'may',
 ↪  'copy']
```

As we can see the empty entries are gone (we had two empty entries in the `words.take(35)` above).

5.1.1.3 Map the words to a tuple (#3)

In the next line of code we will apply the `map`:

```
key_values = not_empty.map(lambda word: (word, 1))
```

This means, that each `word` will be mapped to `(word, 1)`. We need this in order to be able to count the words.

```
print(key_values.take(35))
```

```
[('The', 1), ('Project', 1), ('Gutenberg', 1), ('EBook', 1), ('of', 1),
 ↪  ('Moby', 1), ('Dick;', 1), ('or', 1), ('The', 1), ('Whale,', 1), ('by',
 ↪  1), ('Herman', 1), ('Melville', 1), ('This', 1), ('eBook', 1), ('is',
 ↪  1), ('for', 1), ('the', 1), ('use', 1), ('of', 1), ('anyone', 1),
 ↪  ('anywhere', 1), ('at', 1), ('no', 1), ('cost', 1), ('and', 1), ('with',
 ↪  1), ('almost', 1), ('no', 1), ('restrictions', 1), ('whatsoever.', 1),
 ↪  ('You', 1), ('may', 1), ('copy', 1)]
```

5.1.1.4 Reduce by key (#4)

Reduce by key in order to count the number of occurances of each word (which is a , b will be transformed to a + b).

```
counts = key_values.reduceByKey(lambda a, b: a + b)
```

Have a look at the **#3** above and try to find the word 'The':

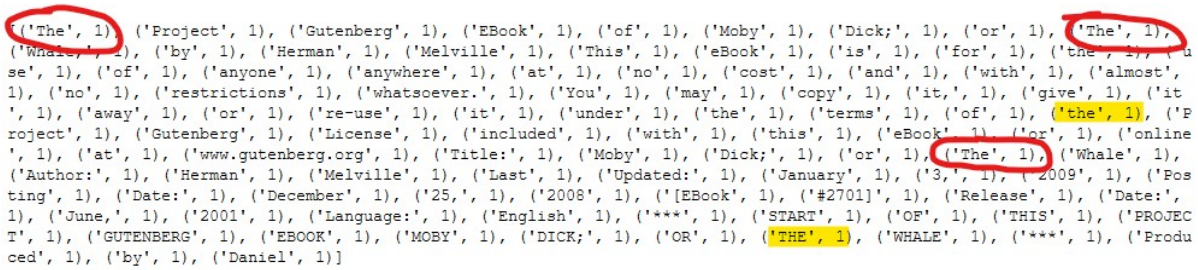


Figure 5.2: Map-Reduce - Word Count Example Reduce by Key

The aim is to count these. Please note that the word 'the' and 'THE' will not be part of this count as they are different. If we wanted to have them aggregated all together we would have needed another map step, which changes every word into its capital cases.

```
[('The', 549), ('Project', 79), ('EBook', 1), ('of', 6587), ('Moby', 79),
 ↪  ('is', 1586), ('use', 35), ('anyone', 5), ('anywhere', 11), ('at',
 ↪  1227), ('no', 447), ('restrictions', 2), ('whatsoever.', 5), ('may',
 ↪  223), ('it,', 237), ('give', 68), ('away', 117), ('re-use', 2), ('this',
 ↪  1169), ('online', 4), ('www.gutenberg.org', 2), ('Author:', 1), ('Last',
 ↪  1), ('January', 1), ('3,', 2), ('Posting', 1), ('Date:', 2), ('#2701]',
 ↪  1), ('June,', 3), ('Language:', 1), ('English', 42), ('***', 6), ('OF',
 ↪  59), ('THIS', 12), ('GUTENBERG', 3), ('MOBY', 3), ('DICK;', 3)]
```

5.1.1.5 Sort by keys and examine top 10 words

Interesting to have would be the top 10 words:

```
countsSort = counts.sortBy(lambda a: a[1], ascending=False)
countsSort.take(10)
```

```
[('the', 13766),
 ('of', 6587),
 ('and', 5951),
 ('a', 4533),
 ('to', 4510),
 ('in', 3878),
 ('that', 2693),
 ('his', 2415),
 ('I', 1724),
 ('with', 1692)]
```

Having words like 'the', 'of', 'and',.. in the top 10 of a text is not surprising. Here is how we can look up the word 'The' and 'THE': `countsSort.lookup('The')` results in 549 and `countsSort.lookup('THE')` results in 98. It would make sense to aggregate the counts for 'the' and 'The' and 'THE'. In this case we would expect have 14413:

```
countsSort.lookup('the')[0]+ countsSort.lookup('The')[0] +
 ↪  countsSort.lookup('THE')[0]
```

After having learnt this, we might think to restart the whole counting steps and include the UPPERCASE. We will do this in **#3**.

5.1.1.6 Steps for counting the words using UPPERCASE

```
#1:
words = text_file.flatMap(lambda line: line.split(" "))

#2:
not_empty = words.filter(lambda x: x!='')

#3:
tmp = not_empty.map(lambda word: word.upper())      # UPPERCASE
```

```
key_values_upper = tmp.map(lambda word: (word, 1))

#4:
counts_upper = key_values_upper.reduceByKey(lambda a, b: a + b)
```

Have a look into the results:

```
counts_upperSort = counts_upper.sortBy(lambda a: a[1], ascending=False)
counts_upperSort.take(20)
```

```
[('THE', 14413),
 ('OF', 6668),
 ('AND', 6309),
 ('A', 4658),
 ('TO', 4595),
 ('IN', 4115),
 ('THAT', 2759),
 ('HIS', 2485),
 ('IT', 1776),
 ('WITH', 1750),
 ('I', 1724),
 ('AS', 1713),
 ('HE', 1683),
 ('BUT', 1672),
 ('IS', 1605),
 ('WAS', 1577),
 ('FOR', 1557),
 ('ALL', 1359),
 ('AT', 1312),
 ('THIS', 1283)]
```

On my Docker machine-learning repository[3] you will find the Jupyter-Notebook, with this example.

5.1.2 Term Frequency–Inverse Document Frequency (TF-idf) Example

In this example I will expand the Map-Reduce from sec. 5.1. The "Term Frequency–Inverse Document Frequency (TF-idf)"[4] is a very interesting measure. It shows, how important a term is to a document.

[3] My docker machine-learning repository, https://hub.docker.com/repository/docker/andreastraut/machine-learning-pyspark

[4] Term frequency - Inverse Document Frequency, https://en.wikipedia.org/wiki/Tf%E2%80%93idf

The TF-idf is a numerical statistic, which is often used in text-based recommender systems and for information retrieval[5] or text mining or to classify texts.

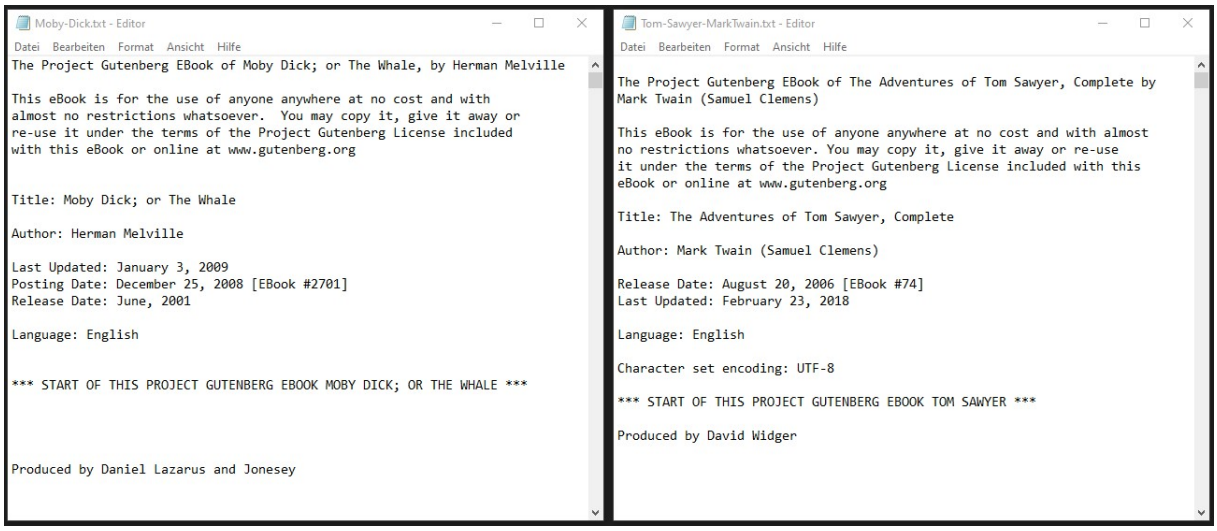


Figure 5.3: TF-idf example - Moby Dick and T om Sawyer

In my example I used the texts of “Moby Dick” and “Tom Sawyer” as input files. By applying the TF-idf measure, the result are two lists of most important words for each of these documents. This is what the TF-idf found for each of the two documents:

```
Moby Dick:
    WHALE, AHAB, WHALES, SPERM, STUBB, QUEEQUEG, STRARBUCK, AYE

Tom Sawyer:
    HUCK, TOMS, BECKY, SID, INJUN, POLLY, POTTER, THATCHER
```

These list of words are (according to the TF-idf measure) the most important ones for each of the texts and they are also the words, which are the least important ones for the other document. If you have read “Moby Dick”, then you would agree, that WHALE, AHAB, WHALES, SPERM, STUBB, QUEEQUEG, STRARBUCK, AYE are very important words for the “Moby Dick” text. And if you have read “Tom Saywer”, then you would also agree, that HUCK, TOMS, BECKY, SID, INJUN, POLLY, POTTER, THATCHER are very important words for the “Tom Sawyer” text. And if you have read both of these texts, then you would agree, that neither of these words are relevant for the opposite text: e.g. in the “Moby Dick” text the name HUCK does not appear and in the “Tom Saywer” text there is no WHALE.

I will explain now, how this works by applying the “Map-Reduce” approach as described in sec. 5.1.

[5] Information Retrieval, https://en.wikipedia.org/wiki/Information_retrieval

5.1.2.1 Count the words

First I start a Spark Session and read the text:

```
document1 = "data/Moby-Dick.txt"
text_file = sc.textFile(document1)
```

Next is to clean the data by eliminating quotation marks, commas and so on (“,.”) which is done in #3:

```
#1:
words = text_file.flatMap(lambda line: line.split(" "))

#2:
not_empty = words.filter(lambda x: x!='')

#3:
tmpA = not_empty.map(lambda word: word.upper())  # UPPERCASE
tmpB = tmpA.map(lambda word:
                word.replace("“", ""))  # eliminate the “
tmpC = tmpB.map(lambda word:
                word.replace("”", ""))  # eliminate the ”
tmpD = tmpC.map(lambda word:
                word.replace(",", ""))  # eliminate the ,
tmpE = tmpD.map(lambda word:
                word.replace(".", ""))  # eliminate the .
tmpF = tmpE.map(lambda word:
                word.replace("\"", "")) # eliminate the "
tmpG = tmpF.map(lambda word:
                word.replace("?", ""))  # eliminate the ?
tmp_char = tmpG.map(lambda word:
                    word.replace("'", "")) # eliminate the '
key_values_upper = tmp_char.map(lambda word: (word, 1))

#4:
counts_upper = key_values_upper.reduceByKey(lambda a, b: a + b)
```

I know that the `replace(...)` in #3 is not very elegant. Better would have been to work with a regular expression. But I didn’t want to over-complicate the code here. I think, what I did here is easy to understand. I do the same as above for the other text:

```
document2 = "data/Tom-Sawyer-MarkTwain.txt"
text_file2 = sc.textFile(document2)
```

At the end I have `counts_upper2` for the second text.

5.1.2.2 Calculate TF

If we want to calculate the **TF-idf** measure (see Wikipedia) we would need to do calculate **TF** (Term-Frequency) and **idf** (inverse-document-frequency) sperately and then divide them: **TF / idf**. So first I need to calcualte **TF**:

```
#Number of words for each documents:
n1 = key_values_upper.count()
n2 = key_values_upper2.count()
print("Document 1: {}".format(document1))
print("Total number of words: {}\n".format(n1))
print("Document 2: {}".format(document2))
print("Total number of words: {}".format(n2))
```

```
Document 1: data/Moby-Dick.txt
Total number of words: 215133

Document 2: data/Tom-Sawyer-MarkTwain.txt
Total number of words: 73840
```

Therefore the **TF** for terms of document 1 are as follows:

```
TF1 = counts_upperSort.mapValues(lambda value: value/n1)
TF1.take(7)
```

```
[('THE', 0.06733973867328583),
 ('OF', 0.031120283731459145),
 ('AND', 0.029888487586748662),
 ('A', 0.021772577893675076),
 ('TO', 0.02159594297481093),
 ('IN', 0.019443785937071485),
 ('THAT', 0.013586943890523536)]
```

And the **TF** for terms of document 2 are as follows:

```
TF2 = counts_upperSort2.mapValues(lambda value: value/n2)
TF2.take(7)
```

```
[('THE', 0.05310130010834236),
 ('AND', 0.04156283856988082),
 ('A', 0.025338569880823402),
 ('TO', 0.02421451787648971),
 ('OF', 0.021424702058504875),
 ('HE', 0.015912784398699892),
 ('WAS', 0.015723185265438786)]
```

5.1.2.3 Calculate idf

As we have N=2 documents the **idf** is defined as `log(N/n)=log(2/n)`. For calculating the **idf** we need to know if a term (e.g. the word 'THE') occurs in:

- both documents: then n=2 and the **idf**=`log(1)=0`
- only one of the documents: then n=1 and the **idf**=`log(2)`
- in none of the documents: then n=0 and the **idf**=`log(2/0)` which is not defined (division by 0).

We conclude that when having N=2documents we are only interested in the case, where a term occurs in only **one** document (n=1). I define a idf-funktion for this:

```
import math
def idf(term):
    N = 0
    if (counts_upperSort.lookup(term)):
        N += 1
    if (counts_upperSort2.lookup(term)):
        N += 1
    if (N==2):
        print("Term: {0}\nIn both documents.\nidf('{0}') = {1:0.2f}"
              .format(term, math.log10(2/N)))
    elif (N==1):
        print("Term: {0}\nOnly in one of the documents.\nidf('{0}') =
         ↪  {1:0.2f}"
              .format(term, math.log10(2/N)))
    elif (N==0):
        print("Term: {0}\nDoes not appear.\nidf('{0}') = not defined"
              .format(term))
```

Now I want to test the idf-function on the word “THE”

```
idf('THE')
```

```
Term: THE
In both documents.
idf('THE') = 0.00
```

The idf for ‘THE’ is 0.00 which means, that it is not very informative (it appears in both documents).

```
idf('MOBY')
```

```
Term: MOBY
Only in one of the documents.
idf('MOBY') = 0.30
```

And a final test on an arbitrary word: `idf('BNBWKDMGJ')`

```
Term: BNBWKDMGJ
Does not appear.
idf('BNBWKDMGJ') = not defined
```

5.1.2.4 Calculate TF-idf

Now I want to put it together and calculate the TF-idf measure. First I will create the tuple `(x[0], x[1])`, where `x[0]` are the **TF** for each of the terms of document 1 and `x[1]` are the **TF** for each of the terms of document 2.

If `(x[0], x[1])` has values for `x[0]` and `x[1]` then the term occurs in both documents.
If `(x[0], None)` then the term occurs only in document 1.
If `(None, x[0])` then the term occurs only in document 2.

```
both_tf = TF1.fullOuterJoin(TF2)
both_tf.take(7)
```

```
[('TO', (0.02159594297481093, 0.02421451787648971)),
 ('THAT', (0.013586943890523536, 0.012080173347778982)),
 ('AS', (0.008013647371625925, 0.005444203683640303)),
```

```
 ('ONE', (0.004062603133875324, 0.002397074756229686)),
 ('WERE', (0.00313759395350783, 0.0039003250270855903)),
 ('THEY', (0.003058573068752818, 0.007367280606717226)),
 ('THEIR', (0.0028772898625501436, 0.002627302275189599))]
```

Now calculate the TF-idf for both documents (I will leave out some words, because the result would be too long to show here):

```
both_tfidf = both_tf.mapValues(lambda x:
                               (x[0]*math.log10(2/1),"doc 1")
                               if x[1] is None
                               else (x[1]*math.log10(2/1), "doc 2")
                               if x[0] is None
                               else 0)
both_tfidf.take(40)
```

```
[('TO', 0),
 ('THAT', 0),
 ('AS', 0),
 ...
 ('COULD', 0),
 ('SAME', 0),
 ('STUBB', (0.00029244824872879597, 'doc 1')),
 ('NEVER', 0),
 ('CAN', 0),
 ('HAND', 0),
 ('TOO', 0),
 ('FAR', 0),
 ('SHIPS', 0),
 ('PART', 0),
 ('STARBUCK', (0.00020569326585231104, 'doc 1'))]
```

5.1.2.5 Information retrieval: get most important words

If a term occurs in both documents we already know from above, that the **TF-idf** is 0. It means, that this term is not very interesting and therefore we will get rid of them with the `filter` function:

```
both_tfidf_not0 = both_tfidf.filter(lambda x: x[1]!=0)
both_tfidf_not0.take(7)
```

```
[('STUBB', (0.00029244824872879597, 'doc 1')),
 ('STARBUCK', (0.00020569326585231104, 'doc 1')),
 ('TOWARDS', (0.00015951722657934328, 'doc 1')),
 ('WHALING', (0.00015112158307516732, 'doc 1')),
 ('NANTUCKET', (0.00012173683081055145, 'doc 1')),
 ('LEVIATHAN', (0.00011753900905846347, 'doc 1')),
 ('OIL', (0.00010354626988483686, 'doc 1'))]
```

Now let's sort the **TF-idf** and see, which terms are most interesting for terms of both documents (the `'doc 1'` and `'doc 2'` will let you know, in which document this term occurred:

```
both_tfidf_not0Sort = both_tfidf_not0.sortBy(lambda a: a[1],
 ↪  ascending=False)
both_tfidf_not0Sort.take(7)
```

```
[('WHALE', (0.0012075733906839758, 'doc 1')),
 ('HUCK', (0.0008561253939522759, 'doc 2')),
 ('AHAB', (0.0005401197321019868, 'doc 1')),
 ('WHALES', (0.0005177313494241844, 'doc 1')),
 ('TOMS', (0.0003832180334833997, 'doc 2')),
 ('BECKY', (0.0003791412458931507, 'doc 2')),
 ('SPERM', (0.0003204337270760492, 'doc 1'))]
```

But more interesting would be to have the top 30 words of documents 1. Here they are:

```
tmp = both_tfidf_not0Sort.mapValues(lambda x: x[0] if x[1]=='doc 1' else 0)
doc1_tfidf_not0Sort = tmp.filter(lambda x: x[1]!=0)
doc1_tfidf_not0Sort.take(10)
```

```
[('WHALE', 0.0012075733906839758),
 ('AHAB', 0.0005401197321019868),
 ('WHALES', 0.0005177313494241844),
 ('SPERM', 0.0003204337270760492),
 ('STUBB', 0.00029244824872879597),
 ('QUEEQUEG', 0.00028405260522462),
 ('STARBUCK', 0.00020569326585231104),
 ('AYE', 0.00016651359616615659),
 ('PEQUOD', 0.00016651359616615659),
 ('CREW', 0.00016371504833143124)]
```

You remember: the first document was the "Moby Dick" text. It is not surprising, that 'WHALE' and 'AHAB' (the captain) are in the top 20 words in document 1. As "Moby Dick" is "Sperm Whale" we will find the word 'SPERM' very often. Therefor we are able to extract the most important words of the "Moby Dick" text by using the **TF-idf** measure. This is what we call **information retrieval**: I extracted the top 20 most important words from the "Moby Dick" text.

The second document was "Tom Saywer". Let's have a look into the top 30 words of this document 2:

```
tmp = both_tfidf_not0Sort.mapValues(lambda x: x[0] if x[1]=='doc 2' else 0)
doc2_tfidf_not0Sort = tmp.filter(lambda x: x[1]!=0)
doc2_tfidf_not0Sort.take(10)
```

```
[('HUCK', 0.0008561253939522759),
 ('TOMS', 0.0003832180334833997),
 ('BECKY', 0.0003791412458931507),
 ('SID', 0.0002976054940881721),
 ('INJUN', 0.00028129834372717636),
 ('POLLY', 0.00018345544156120197),
 ('POTTER', 0.00017122507879045517),
 ('THATCHER', 0.00015491792842945945),
 ('SAWYER', 0.0001345339904782148),
 ('HARPER', 0.0001345339904782148)]
```

You can see 'TOM' and his best friend 'HUCK', as well as 'BECKY' (the girl to whom Tom fell in love), which is also not surprising. Again: we found the most important top 20 words for the "Tom Sawyer" text.

If you like to have the result as a dictionary of keys, you can use the following code (I won't show the whole dictionary, it's very long):

```
# Moby Dick
doc1_tfidf_not0Sort.collectAsMap().keys()
```

```
dict_keys(['WHALE', 'AHAB', 'WHALES', 'SPERM', 'STUBB', 'QUEEQUEG',
 ↪  'STARBUCK', 'AYE', 'PEQUOD', 'CREW', 'TOWARDS', 'WHALING', 'FLASK',
 ↪  'NANTUCKET', 'LEVIATHAN', 'MOBY', 'ERE', 'OIL', 'SAILOR', 'AHABS',
 ↪  'HARPOONEER', 'CABIN', ....
```

```
# Tom Sawyer
doc2_tfidf_not0Sort.collectAsMap().keys()
```

```
dict_keys(['HUCK', 'TOMS', 'BECKY', 'SID', 'INJUN', 'POLLY', 'POTTER',
 ↪  'THATCHER', 'SAWYER', 'HARPER', 'TOWARD', 'HUCKLEBERRY', 'HUCKS', 'BEN',
 ↪  'MUFF', 'NOBODY', 'HADNT', 'JIM', 'WARNT', 'TAVERN', 'DOUGLAS',
 ↪  'WELSHMAN', 'HANTED', ...
```

On my Docker machine-learning repository[6] you will find the Jupyter-Notebook, with this example.

5.1.2.6 Conclusion

By using the **TF-idf** measure I was able to extract the most important terms for the two documents "Moby Dick" and "Tom Saywer" as shown below.

We can say two things now:

One: We can say, that the words on each of theses lists **"describe the documents"**. The list has a very "close relationship" to the document and these words are the "information", which we will find in the document. Therefore we have the **information retrieval** part here. For example if you haven't read "Moby Dick" then you would know from this list, that it's about a WHALE, because WHALE is part of the list and the two main characters in "Moby Dick" are AHAB, STUBB and QUEEQUEG because these words are also on the list.

Two: We can also say, that for one docoment the most important words on the lists are the **"most different"** words to all other documents. These words "separate" the documents. Therefore these words could be helpful to **"classify documents"**. For example: image you do the same TF-idf calculation for thousands of documents. Then you will get thousands of lists like shown here. You would expect to have some lists which have a big overlap and others which don't. Now you can say: the smaller the overlap of one list to all the other, the more "different" this document is compared to all the others. And vice-versa: the bigger the overlap of one list to another, the more "similar" the document is to the other. One application for doing this is to "classify documents" or to **"create a better search algorithms"**. For example: if you are working in a law firm, then you probably have thousands of court judgments. It might be interesting for you to find all court judgments, which are the most similar to your current running legal process. Building a "search algorithm" based on these TF-idf lists can very efficient and helpful for finding the **"most similar"** and **"most relevant"** documents.

[6] My docker machine-learning repository, https://hub.docker.com/repository/docker/andreastraut/machine-learning-pyspark

5.2 K-Means Clustering Algorithm

Additionally I worked on this dataset to show how the K-Means Clustering Algorithm[7] can be applied by using the Spark Machine-Learning Library. I will show how the "Vermont Vendor Payments" dataset can be clustered. In the images below every color represents a different cluster:

```
from pyspark.ml.clustering import KMeans
```

5.2.0.1 Three Clusters (K=3)

Now I set the parameter K initially to 3:

```
kmeans = KMeans().setK(3).setSeed(1)
```

Do the fit:

```
model = kmeans.fit(dataset)
model.setPredictionCol('prediction')
```

```
KMeansModel:
    uid=KMeans_00206e60dd5b,
    k=3,
    distanceMeasure=euclidean,
    numFeatures=2
```

Do the transform:

```
transformed = model.transform(dataset) \
                    .select('features', 'prediction')
transformed
print((transformed.count(), len(transformed.columns)))
```

```
DataFrame[features: vector, prediction: int]
(3387, 2)
```

How many clusters do we have and how many points does each of these cluster have?

[7] K-Means Clustering Algorithm, https://spark.apache.org/docs/latest/ml-clustering.html#k-means

```
model.summary.clusterSizes
```

```
[3385, 1, 1]
```

This means, that the first cluster contains 3385 data point, the second only one and the third also only one. These are extreme values.

```
transformed.sort('prediction', ascending=False) \
            .show(n=transformed.count(), truncate=False)
```

```
+-----------------------+----------+
|features               |prediction|
+-----------------------+----------+
|[1.0,2.554683953107E10] |2         |
|[0.0,2.1964140145164E11]|1         |
|[49.0,5948.0]          |0         |
|[49.0,3736.0]          |0         |
|[49.0,471766.88]       |0         |
|[49.0,3916.0]          |0         |
|[49.0,501.25]          |0         |
|[49.0,2421.4]          |0         |
...
```

As expected the two extreme values form two separate clusters (each containing only one single data point). The third cluster is the remaining rest.

5.2.0.2 Five Cluster (K=5)

Let's have a look at the result for K=5 cluster:

```
kmeans = KMeans().setK(5).setSeed(1)
model = kmeans.fit(dataset)
model.setPredictionCol('prediction')
model.summary.clusterSizes
transformed = model.transform(dataset) \
                   .select('features', 'prediction')
transformed.sort('prediction', ascending=False) \
            .show(n=transformed.count(),
                  truncate=False)
```

```
[3373, 1, 1, 12, 0]
+-----------------------+----------+
|features               |prediction|
+-----------------------+----------+
|[7.0,2.32484304242E9]  |3         |
|[2.0,2.33823407921E9]  |3         |
|[9.0,2.32676185131E9]  |3         |
...
|[0.0,1.49675535347E9]  |3         |
|[0.0,1.66099306197E9]  |3         |
|[1.0,2.554683953107E10]|2         |
|[0.0,2.1964140145164E11]|1        |
|[64.0,1915533.81]      |0         |
|[64.0,40981.07]        |0         |
...
```

Hide the cluster “prediction=0” and only show the other clusters:

```
transformed.filter(F.col('prediction') != '0') \
                    .sort('features',
                          ascending=True) \
                    .show(n=transformed.count(),
                          truncate=False)
```

```
+-----------------------+----------+
|features               |prediction|
+-----------------------+----------+
|[0.0,1.49603759082E9]  |3         |
|[0.0,1.49675535347E9]  |3         |
|[0.0,1.66099306197E9]  |3         |
|[0.0,2.17201705172E9]  |3         |
|[0.0,3.12255843328E9]  |3         |
|[0.0,4.58524599324E9]  |3         |
|[0.0,5.05357651587E9]  |3         |
|[0.0,6.11474963014E9]  |3         |
|[0.0,2.1964140145164E11]|1        |
|[1.0,2.554683953107E10]|2         |
|[2.0,2.33823407921E9]  |3         |
|[7.0,2.32484304242E9]  |3         |
|[9.0,2.32676185131E9]  |3         |
|[22.0,1.83834865364E9] |3         |
+-----------------------+----------+
```

As expected we have a **cluster number 1** (see “prediction=1”), with the extreme value 2 * 10e11.

And another **cluster number 2** (see “prediction=2”) with the other extreme value 2.5 * 10e10.

And the **cluster number 3** (see “prediction=3”) has values around 1 * 10e9 and 6 * 10e9.

Therefore the K-Means algorithm found exactly the clusters, which we expected. Great!

5.2.0.3 Ten Clusters (K=10)

Let’s have a look at K=10 clusters:

```
kmeans = KMeans().setK(10).setSeed(1)
model = kmeans.fit(dataset)
model.setPredictionCol('prediction')
model.summary.clusterSizes
transformed = model.transform(dataset) \
                   .select('features', 'prediction')
transformed.filter(F.col('prediction') != '0') \
                   .sort('features', ascending=True) \
                   .show(n=transformed.count(),
                         truncate=False)
```

```
[3363, 1, 1, 1, 18, 3, 0, 0, 0, 0]
+-----------------------+----------+
|features               |prediction|
+-----------------------+----------+
|[0.0,7.826641758E8]    |4         |
|[0.0,8.3250279165E8]   |4         |
|[0.0,8.7528370872E8]   |4         |
|[0.0,9.3917221933E8]   |4         |
|[0.0,9.9943549988E8]   |4         |
|[0.0,1.04293246515E9]  |4         |
|[0.0,1.2885636375E9]   |4         |
|[0.0,1.35549635179E9]  |4         |
|[0.0,1.49603759082E9]  |4         |
|[0.0,1.49675535347E9]  |4         |
|[0.0,1.66099306197E9]  |4         |
|[0.0,2.17201705172E9]  |4         |
|[0.0,3.12255843328E9]  |5         |
|[0.0,4.58524599324E9]  |5         |
|[0.0,5.05357651587E9]  |5         |
|[0.0,6.11474963014E9]  |3         |
|[0.0,2.1964140145164E11]|1         |
```

```
|[1.0,2.554683953107E10] |2         |
|[2.0,2.33823407921E9]   |4         |
|[4.0,1.16802981964E9]   |4         |
|[5.0,1.17368104165E9]   |4         |
|[7.0,2.32484304242E9]   |4         |
|[9.0,2.32676185131E9]   |4         |
|[22.0,1.83834865364E9]  |4         |
+------------------------+----------+
```

This shows that even if I take K=10 the K-Means algorithm will only find 6 clusters (named with number 0,…,5). This is also what we expected, so the K-Means algorithm works perfectly fine. Great.

The first cluster hast 3363 data point, the second cluster has one data point and so on: `[3363, 1, 1, 1, 18, 3, 0, 0, 0, 0]`

I want to transform the result in to vectors and plot it with colors: each cluster should have a separate color. Therefore I create the function “ith_”:

```
from pyspark.ml.linalg import Vectors
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import lit, udf

def ith_(v, i):
    try:
        return float(v[i])
    except ValueError:
        return None

ith = udf(ith_, DoubleType())
```

```
x = transformed.select(ith("features", lit(0))) \
               .toPandas()
x_name = features[0]
y = transformed.select(ith("features", lit(1))) \
               .toPandas()
y_name = features[1]
z = transformed.select("prediction") \
               .toPandas()
```

I will plot the same data as above but as we know from the K-Means Algorithm for each datapoint (State_index, sumofAmount) to which clutster it belogs, I can plot them using different colors:

- violett

- yellow
- blue
- ...

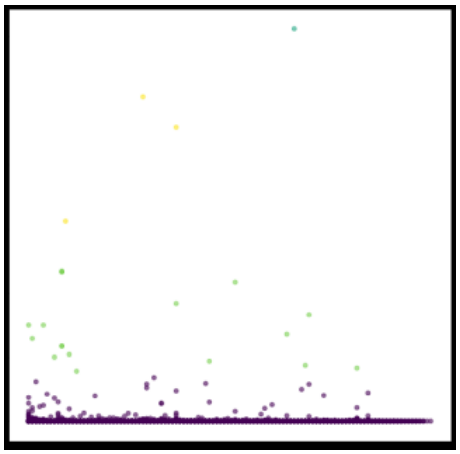
```
my_plotter(x, x_name, y, y_name, z['prediction'])
```

And again: as the extreme values (2 * 10e11,...) create separate cluster (number 1 and number 2), which are disturbing a bit, I filter them out in order to plot only the remaining clusters:

```
transformed_betterVis = transformed.filter((F.col('prediction') >= '3') |
                                           (F.col('prediction') == '0')) \
                                   .sort('features', ascending=True)
transformed_betterVis.show(n=transformed.count(), truncate=False)
```

Now I can plot the remaining data points:

```
x = transformed_betterVis.select(ith("features", lit(0))) \
                         .toPandas()
x_name = features[0]
y = transformed_betterVis.select(ith("features", lit(1))) \
                         .toPandas()
y_name = features[1]
z = transformed_betterVis.select("prediction").toPandas()
my_plotter(x, x_name,
           y, y_name,
           z['prediction'])
```



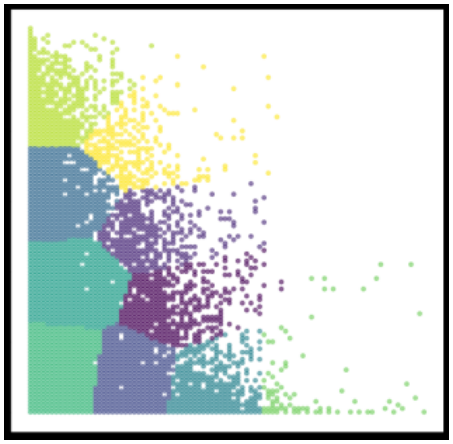
Each color represents a separate cluster and the plot proves, that these clusters make sense:

The violet cluster is **cluster number 3** and contains a data point at about6 * 10e9.

The blue cluster is the the **cluster number 4** and contains values between 1 * 10e9 and 2.5 * 10e9.

The yellow cluster is **cluster number 5** and contains values between 3 * 10e9 and 5 * 10e9.

Of course you can also use other features for the clustering. Another clustering (based on State and Department) results in the following image:



On my Docker machine-learning repository[8] you will find the Jupyter-Notebook, with this example.

[8] My docker machine-learning repository, https://hub.docker.com/repository/docker/andreastraut/machine-learning-pyspark

6 Use Cases of Artificial Intelligence in the Industry

In the following text I explain in an easily understandable way what is meant by ”artificial intelligence (AI) in the industry”, describe areas of application and illustrate with practical examples and programming applications how AI can be implemented in concrete terms.

In the **first part** (see sec. 6.1), I will explain the basic concepts of AI and describe some areas where AI is already being successfully applied. I will also describe the peculiarities of big data, deep learning and process mining.

In the **second part** (see sec. 6.2), I show an example of how the car manufacturer BMW has benefited from AI techniques.

In the **third part** (see sec. 6.3), I show how AI techniques can be implemented in the Python programming language when dealing with the topic of image recognition and provide my programming code in the process.

In the **fourth part** (see sec. 6.4), I give some recommendations on what to look for when introducing AI techniques in a company.

I think the choice of further articles to delve into the topic ”Use Cases of Artificial Intelligence in the Industry” is huge and I hope this short introduction is helpful to get started.

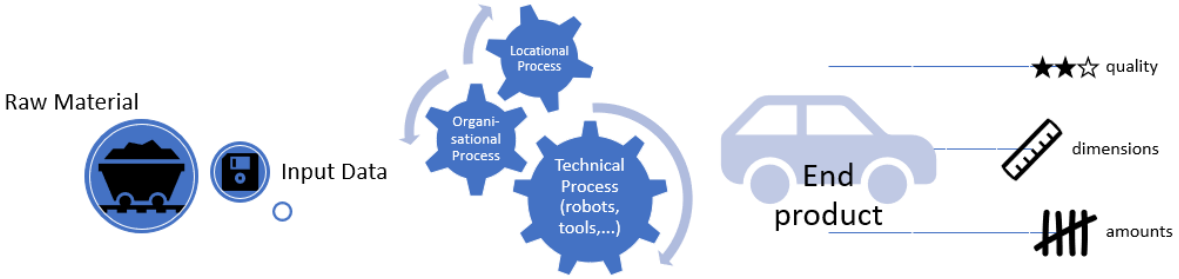


Figure 6.1: Use Cases of AI - Chart

6.1 AI Explained

My diagram fig. 6.1 shows **raw material / input data** on the left and the **end product** on the right. In between, **different processes**. During the runtime of these processes, the intermediate results are usually logged and stored by means of log files (which is also data).

6.1.1 What kind of processes are meant here?

For example, machines could process raw materials that are exposed to a certain temperature and pressure during their processing. Temperature and pressure are determined by sensors, histories with a time stamp and stored in log files. These processes can take place regionally at different locations or organizationally in different company units. There might also be technically differences in these processes (e.g. robots, tools,…).

I would like to mention a few example (there are a lot more) where AI is successfully applied in practice:

- **Sales forecasts**: Artificial intelligence calculates the expected sales of products based on a large number of input data (e.g. stock market data, weather, commodity prices, customs restrictions, price development on the sales markets, inflation, interest rates or social media trends). This makes it possible to better determine expected sales and optimally control production. [1]
- **Automatic orders**: The order quantities and order times for raw materials are automatically determined and optimized by artificial intelligence. This is to prevent storage capacities from being exceeded or delivery bottlenecks from occurring. In addition, as many supplier discount offers as possible are to be optimally utilized. [2]
- **Product development for series production**: Automated tests are carried out on the products and validated by the artificial intelligence so that it can point out where adjustments still need to be made to the products so that they can be produced in series cost-effectively and without errors. [3]
- **Quality control**: Images of the products are generated using sensors, X-rays or high-resolution cameras. Artificial intelligence can then use image recognition algorithms to detect defects in the products and sort them out. [4][5]

[1] Big Data Insider, This is the role of machine learning in sales forecasting, https://www.bigdata-insider.de/diese-rolle-spielt-machine-learning-bei-absatzprognosen-a-625751/

[2] Procurement up-to-date, Optimal raw material inventory thanks to precise AI-based demand forecasts, https://beschaffung-aktuell.industrie.de/logistik/optimaler-rohstoffbestand-dank-praeziser-bedarfsprognosen-auf-ki-basis/

[3] Intel, Artificial intelligence reduces costs and accelerates time to market, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/artificial-intelligence-reduces-costs-and-accelerates-time-to-market-paper.pdf

[4] Fraunhofer Institute, AI-based visual quality control, https://www.iais.fraunhofer.de/de/geschaeftsfelder/Computer-Vision/visuelle-qualitaetskontrolle.html

[5] Elektronik Praxis, AI in quality control: when no detail should be overlooked, https://www.elektronikpraxis.vogel.de/ki-

I have linked further articles in the footnotes. The list could go on, but this selection should cover the most important areas.

6.1.2 What data does artificial intelligence have access to?

Artificial intelligence now has access to all data:

- all input data: also includes all data describing the material properties (length, width, weight...)
- All log data: also includes all data resulting from the processing steps, e.g. temperature or pressure with which materials are processed.
- all output data: includes data that measures the quality of the product but as well the dimension (length,...) and amounts. For example, a quality assurance staff might rate the product as `not ok` because it is defective or because an important KPI metric is not satisfactory.

The artificial intelligence knows everything and can thus establish a **connection** between ”input data” and ”output data” (or ”raw material” and ”end product”) at any time and always has the process (the log files) in view. For example: as soon as a human evaluates the ”output data” or the ”end product” as *”not ok”*, the artificial intelligence can draw a conclusion as to which input parameter or which process step was most relevant for the anomaly and can make a suggestion as to what should be changed.

6.1.3 Is Artificial Intelligence Really Intelligent?

Artificial intelligence is not ”intelligent” as we humans commonly understand it: AI is only an algorithm that can represent these relationships with models. There are different approaches, depending on what is relevant at the time:

- When the amount of input data is huge, we speak of Big Data[6]. This is the case, for example, with sensor data, i.e. when temperature, pressure or travel distances of machines are continuously collected. Each data point in itself is often only a simple number, but over time it adds up to a huge amount of data that can no longer be processed using conventional data processing methods. Big Data approaches are sometimes quite different from conventional approaches to data processing: other systems are used, such as Apache Spark[7] to calculate with networked computers) or Hadoop[8] (to process very large data sets distributed across several computers). I have been working in my Machine-Learning Repository[9] to describe the difference when working with a Big Data system compared to conventional data processing.

in-der-qualitaetskontrolle-wenn-kein-detail-uebersehen-werden-darf-a-860403/

[6] Big Data https://de.wikipedia.org/wiki/Big_Data

[7] Apache Spark, https://spark.apache.org/

[8] Hadoop, https://hadoop.apache.org/

[9] Machine-Learning Repository, https://github.com/AndreasTraut/Machine-Learning-with-Python

- If, on the other hand, the input data takes a back seat (i.e. no Big Data), but the processes come to the fore, this is called *Process mining*[10]. Here, the log files that record the processes are often transformed into models and then evaluated.
- One speaks of a *Deep Learning*[11] approach when neural networks are used. This is often the case when the input data are not just simple data points, as is the case with sensor data, but have a complex structure, such as images or documents. An image consists of several thousand pixels in each of the two image axes and for each of these pixels there are many possible shades of color. A document has sentences, words and letters that follow grammatical and orthographic rules. In the Deep Learning approach, groups of pixels or letters are linked to so-called ”neurons”, which together form a layer. A neuron layer then passes on data to the next neuron layer above it according to a given arithmetic operation, and when hundreds of such layers are stacked on top of each other, you get a neural network (hence the name ”deep”). I have also dealt with this and documented my experiences[^repodeeplearning].

AI approaches can therefore be used to penetrate the **interrelationships of** the input data, the log files and the output data. I will explain in the next section how profit can be generated with this.

6.2 Example: Benefits of AI for BMW

The ”Capgemini Research Institute” published in December 2019 an interesting study[12]: 300 companies from the industrial manufacturing, automotive, consumer goods, aerospace and defense sectors were examined and it was found that companies in Germany already use a great deal of artificial intelligence in their value chains and production processes compared to other countries, but should deepen this.

I would like to briefly pick out a concrete example from the CBR Exclusive Article “How BMW optimised supply chain big data with Teradata”[13], on which this study is based:

The car manufacturer BMW works at 31 production sites where very complex processes take place. In the picture above I have tried to show the countless different raw materials in red. The process from the raw materials to the final product (the car) is very long, complicated and confusing and is sometimes scattered over different continents. Inventory is stored temporarily at many points in the process. A major challenge for BMW was to store the large amount of data that is generated in a meaningful form (keyword: Data Warehouse[14] and Data Lake[15]).

[10] Process mining, Wikipedia, https://en.wikipedia.org/wiki/Process-Mining

[11] Deep-Learning, Wikipedia, https://en.wikipedia.org/wiki/Deep_learning

[12] ”Capgemini Research Institute” published an interesting study, https://www.capgemini.com/de-de/news/ki-in-der-industrie/

[13] CBR-Online: “How BWM optimised supply chain big data with Teradata”, https://www.cbronline.com/big-data/analytics/bmw-optimise

[14] Data Warehouse, https://de.wikipedia.org/wiki/Data_Warehouse

[15] Data Lake, https://de.wikipedia.org/wiki/Data-Lake

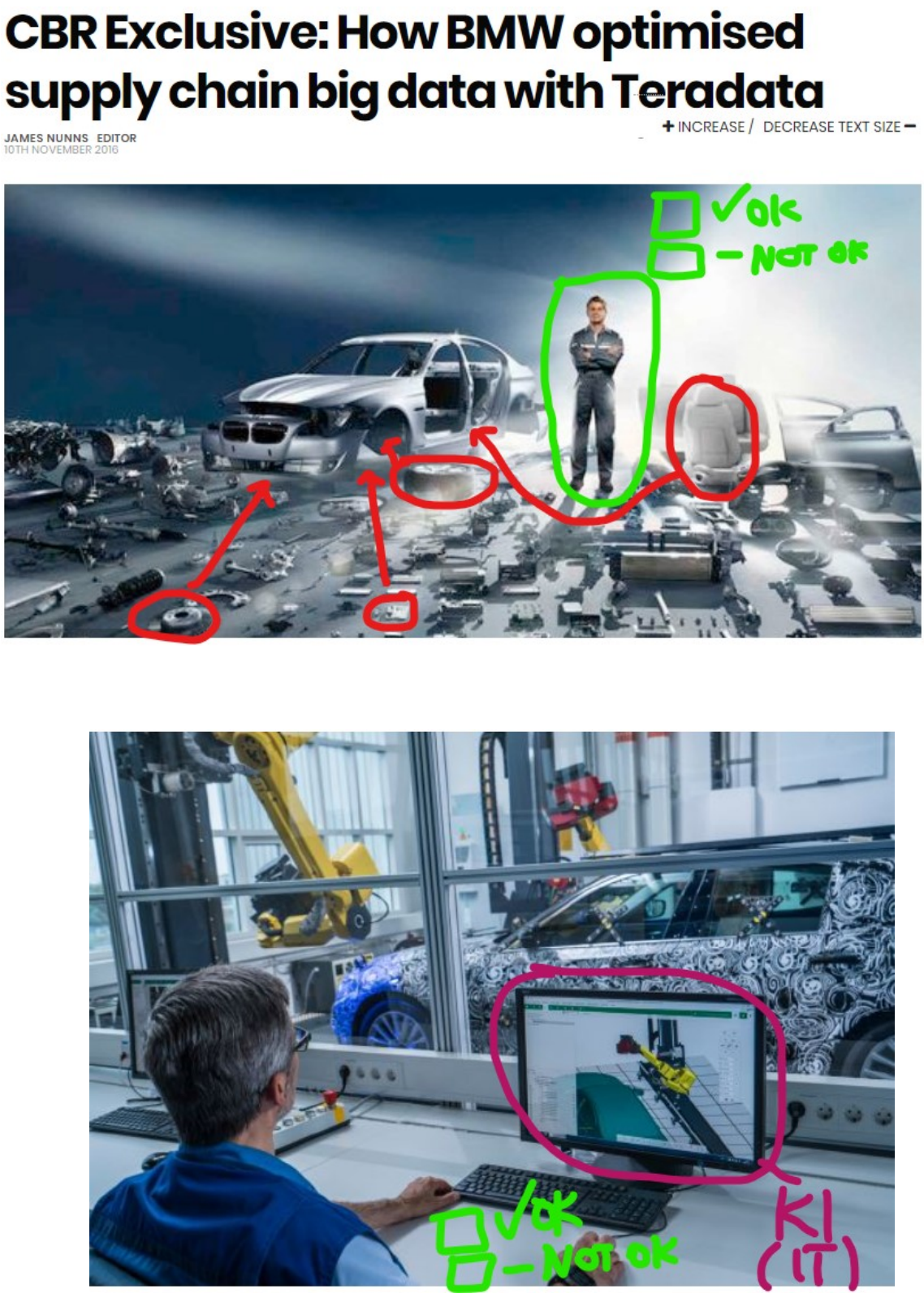


Figure 6.2: CBR-Online: "How BMW optimised supply chain big data with Teradata", Source: BMW Group

BMW celebrated a success in 2016 when it was able to gain valuable insights from analyzing its inventory: the teams were able to reduce inventory costs by 70% by gaining more transparency across their many production sites and optimizing processes.

In 2016, Klaus Straub, CIO at BMW, described in an article[16] his ideas for the digital transformation of the company. Even then in 2016, he saw the great potential that artificial intelligence would create, for example, to **improve quality** or **make processes more efficient,** although linking IT with real production processes would be a major challenge.

But how can this be implemented in concrete terms? I would like to give an insight into this in the following section.

6.3 Implementation of AI Techniques

There are many free and freely available (Open Source[17]) tools that you only have to adapt to your own needs depending on the question. How exactly this is done is shown below in my program code.

6.3.1 What can be seen in the picture?

For example, if there is a picture of a component and the question is what is in that picture, a human could quickly find out by just looking at it. The human could also see whether the component is defective or not. The AI can do that too, and for this question I choose a deep-learning approach and use the *ResNet50*[18] network, which is already trained on the *ImageNet*[19] image data set. This saves me a lot of programming effort and so I only need 10 lines of code for the model to tell me that in fig. 6.3 the program sees a ”cup” with 86% probability and a ”coffee cup (coffeepot)” with 6.8%. In fig. 6.5 [20] the model sees a “switch component” with 76% probability. You can download my program code from my GitHub repository [21].

[16] Global Intelligence for Digital Leaders, Klaus Straub, CIO BWM, “Powering Digital Disruption at BMW”, https://www.i-cio.com/profession/cio-profiles/item/powering-digital-disruption-at-bmwhttps://www.i-cio.com/profession/cio-profiles/item/powering-digital-disruption-at-bmw

[17] Open Source, Wikipedia https://en.wikipedia.org/wiki/Open_Source

[18] ResNet50, Tensorflow https://www.tensorflow.org/api_docs/python/tf/keras/applications/ResNet50

[19] “ImageNet”, http://www.image-net.org/

[20] Pixabay, network map, https://pixabay.com/de/photos/netzwerkkarte-bauteil-schaltkreis-550544/

[21] Jupyter-Notebook ”Espresso / Network Card Example, https://github.com/AndreasTraut/Deep-Learning/blob/master/Image_classification/Image_classifier_example_2_transfer_learning_ResNet52-German.ipynb

Figure 6.3: Use Case - ResNet50: Espressocup

```
resnet = ResNet50(weights='imagenet')
x = preprocess_input(np.expand_dims(img.copy(), axis=0))
preds = resnet.predict(x)
decode_predictions(preds, top=5)
```

```
[[('n07930864', 'cup', 0.86062807),
  ('n03063689', 'coffeepot', 0.06782799),
  ('n03063599', 'coffee_mug', 0.05167182),
  ('n03950228', 'pitcher', 0.0064888997),
  ('n04398044', 'teapot', 0.005076931)]]
```

Figure 6.4: Use Case - ResNet50: Code for Prediction of Espressocup

Figure 6.5: Use Case - ResNet50: Network Card, Pixabay[22]

```
[[('n04372370', 'switch', 0.76144683),
  ('n03777754', 'modem', 0.08100146),
  ('n03272010', 'electric_guitar', 0.032640774),
  ('n03492542', 'hard_disc', 0.015652671),
  ('n04070727', 'refrigerator', 0.008717526)]]
```

Figure 6.6: Use Case - ResNet50: Code for Prediction Network Card

[22] Pixabay, network map, https://pixabay.com/de/photos/netzwerkkarte-bauteil-schaltkreis-550544/

6.3.2 Which groups can be formed?

Let's say we have a picture that we don't know much about and which we want to classify in a grouping that we know. For example, an X-ray picture where we ask ourselves whether and which disease is to be seen on it[23]. Or a picture of a plant for which we want to know the name and care instructions. Texts can also be grouped. In the case of a document or contract, we may be looking for similar texts. Grouping similar things is a frequently discussed problem. We all know the useful Google function to show similar images. You enter a term (e.g. espresso cup) in the search bar and similar images are displayed:

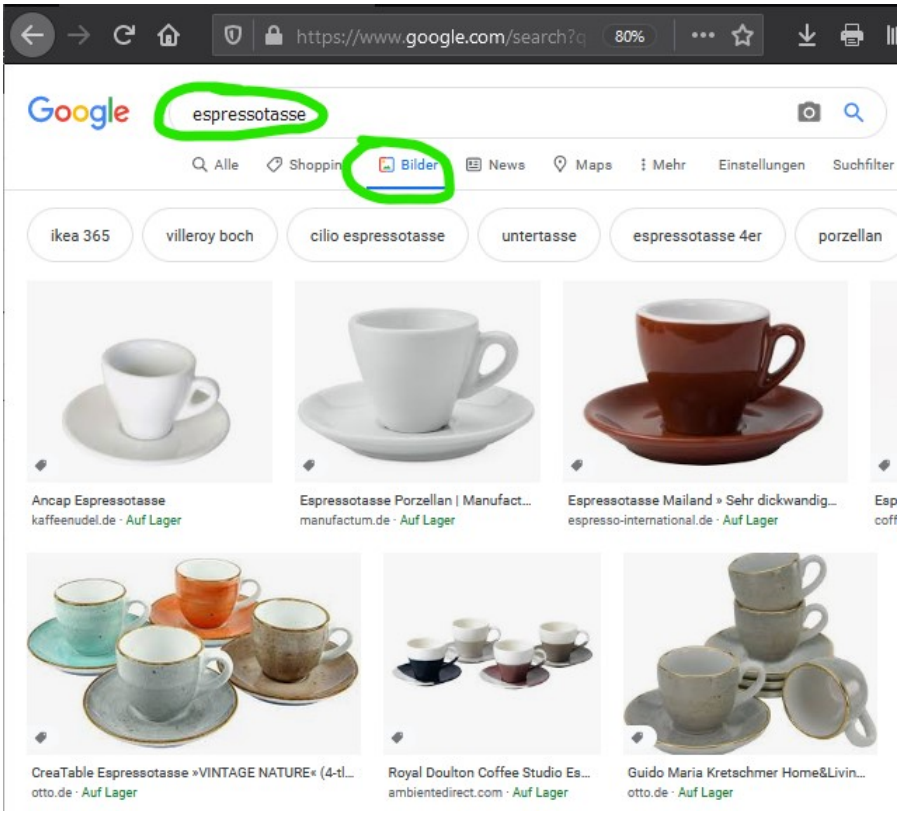


Figure 6.7: Use Case - Google: Similar Pictures

Two questions arise when implementing the program code.

The first question is: How do you compare two pictures? Or two texts? It's not easy, but this problem has been analyzed many times. So there are procedures that you just have to copy, and I will show you how in a moment.

The second question is: How do you go about comparing all the things (pictures or texts) in pairs? Let's take 1 million things that we compare with each other in pairs in order to be able to form the groups.

[23] Die Zeit, Artificial Intelligence: When Computers Evaluate X-Ray Images, https://www.zeit.de/wissen/2019-09/kuenstliche-intelligenz-medizin-diagnose-krankheiten-bilddiagnostik

Then we already have 1 million times 999 999 / 2, that is about 500 billion arithmetic operations. That can take a very long time. Since this question involves a lot of input data, I choose a Big Data approach, namely "Local-Sensitive-Hashing" (LSH).[24] LSH is a technique to classify similar things into groups with a high probability. In other words, one does without absolutely exact results and accepts a small probability of error. This probability can be set with control parameters (as needed). Once these parameters are set, the AI algorithm can very quickly classify new images into groups. The advantage of doing without an absolutely exact 100% grouping is obvious: LSH runs much faster than 100% exact algorithms.

The result of my work was: I applied the LSH algorithm to over 9000 images (each about 300*300 pixels), including my collection of church windows and espresso cups (I like to drink espresso and started photographing the cups at some point). On my own computer, this grouping took a few minutes and was only necessary once. After that, I downloaded a completely new picture of an espresso cup from the internet:

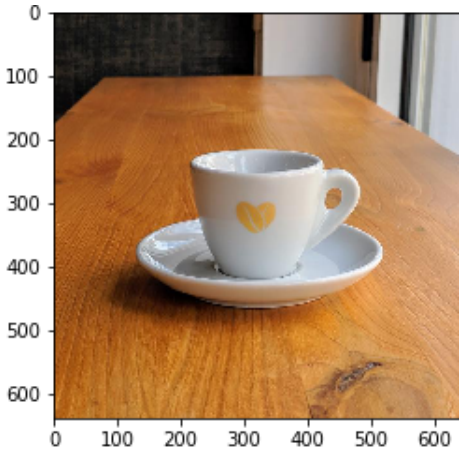


Figure 6.8: Use Case - Espresso Cup

I gave this picture to the program and then used it to search for similar pictures from my picture collection. After a few seconds, the program showed me these 15 pictures:

Figure 6.9: Use Case - Espresso Cup: Similar Pictures

[24] Towards Datascience, Locality Sensitive HashingAn effective way of reducing the dimensionality of your data, https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

I then tested the same for my church window images with this test image:

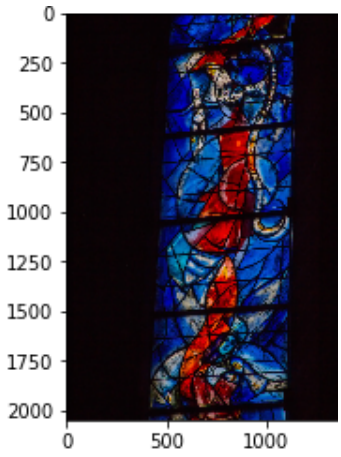


Figure 6.10: Use Case - Church Window

Result:

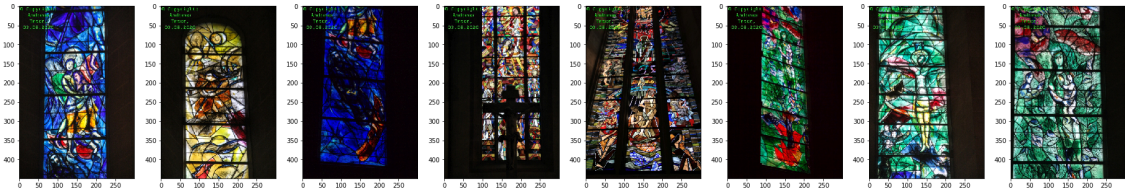


Figure 6.11: Use Case - Church Window: Similar Pictures

If you want to have a quick look at the lines of code to make sure that only a few lines of code are needed for this problem, you can download my program code from my GitHub repository[25] and read my further explanations[26]. I have also explained the deep-learning topic in more depth in my Deep-Learning repository[27].

In the next section, I will give you a first concept to introduce AI techniques in your company.

6.4 Recommendations For Implementing AI

If you are now interested in using artificial intelligence in your company as well, the recommendation is that you consider the following:

[25] Jupyter-Notebook "Similar Pictures" Example, https://github.com/AndreasTraut/Deep_learning_explorations/blob/master/8_Image_similarity_search/Beispiel_aehnliche_Bilder_finden.ipynb

[26] "Similar Pictures" Example: further explanations, https://github.com/AndreasTraut/Deep_learning_explorations/blob/master/README.md

[27] Deep-Learning repository, https://github.com/AndreasTraut/Deep-Learning

What concrete benefits do you expect the analysis of your data to bring you? First collect and structure background knowledge about your company: Which company units are affected, who are the key people, who is the ”sponsor” of the project?

- Describe the problem and the motivation for the data analysis project. Also think about the current situation, its advantages and disadvantages. You will need this to compare with the new data analysis project.
- Describe what a successfully implemented data analysis project would look like: Are there success metrics (objective goals) or subjective goals that you can define or describe to measure the ”success” of the data analysis project for your company? It is important to define measurable business objectives so that further measurable objectives can be derived from them for the further implementation of the data analysis project: What kind of data analysis should be targeted for the problem? What data is specifically needed for these models and what technical and organizational steps are needed to extract, transform, model and evaluate this data from different sources?
- Early on, also ask yourself how the ”deployment” is to proceed, i.e. how the programs that were developed in a test environment are to be made to run in daily productive operation. Should you use your own computers or the cloud? If your company has high data protection requirements, a cloud solution may not be the first choice and should be questioned. An expensive investment in your own hardware could then be the next step for you. If, on the other hand, you want to try out different things first and are not yet ready to invest massive capital in new hardware, then a cloud solution could be ideal for you. You can read about my experiences with the Microsoft Azure Cloud Platform[28].

There are methodological approaches that can be applied when transforming into such a data analysis project. Since the costs for Big Data systems are enormous (financial costs but also the time your employees are tied up) and usually many company areas are affected, it is advisable to take a structured approach.

I hope that my brief introduction to the topic of *”artificial intelligence in industry”* was helpful for getting started and I think that the selection of further articles for delving into the topic is huge. I wish you much fun and success in your further research and implementation.

[28] My Documented Experiences with the Microsoft Azure Cloud Platform, https://github.com/AndreasTraut/Experiences-with-MicrosoftAzure

List of Figures