From Zero to Senior Data Science

A Giude into
Advanced Data Science Standards
and Big Data Approaches

By Andreas Traut

From Zero to Senior Data Science

A Giude 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

Contents

1 Introduction

1.1 Aim

The goal of this book is to guide you to becoming a Senior Data Scientist without too much prior knowledge. My aim is to help you to get started in the Data Science topics and explain the necessary knowledge to become a Senior Data Scientist. 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 Docker, Jupyter-Notebooks,…) on your own computer.

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 to you to get started and my motivation is to enable you to continue working independently yourself with these references to further documentation. 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 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. I hope you enjoy reading and experimenting.

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! 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 area for the future. Even HR managers struggle in correctly evaluating the meaning and requirements of "Data Science".

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

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 person 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. This person denies that fact, that programming skills with machine learning libraries is also very essential in order to be a “Data Scientist”.

Another person writes “Data Scientist” in a job profile and thinks that you have to have many years of profound programming experience and lists programming languages and tools that you can only learn and understand in depth in two lifetimes ignoring that mathematical skills are also required to be a Data Scientist.

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**”[2]. 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.

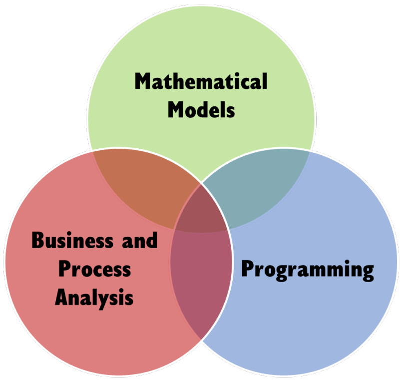


Figure 1.1: Data Science - Adapted version. Creative Commons Legal Code

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

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

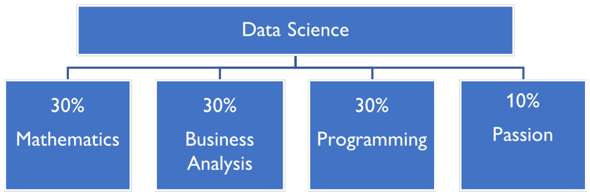


Figure 1.2: Data Science

So "Data Science" is a mix of mathematical knowledge, machine-learning programming experience and business knowledge. The balanced intersection of these areas makes a good Data Scientist.

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!

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**"[3] from"*Fraunhofer-Institut für Intelligente Analyse- und Informationssysteme IAIS*"[4] 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.

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
- a 40 paged expose (similar to a bachelor thesis) approved by Fraunhofer and related to a data science topic

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

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

Besides this I found various articles among them also one on the Online-Magazine "Towards Data Science"[5]. 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.

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 as he wants to have it.

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[6], which is a widespread standard today, but I also use an Integrated Development Environment[7]. 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

In the data scientist environment 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[8] or Seaborn[9], which are available for free. I will show you in Section sec. 2 how to install these tools on your own computer.

[5] "Towards Data Science", https://towardsdatascience.com/becoming-a-level-3-0-data-scientist-52641ff73cb3
[6] Jupyter-Notebook, https://jupyter.org/
[7] Integrated development environment, Wikipedia, https://en.wikipedia.org/wiki/Integrated_development_environment
[8] Matplotlib, https://matplotlib.org/
[9] Seaborn, https://seaborn.pydata.org/

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 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". Data Scientist often forget, that all models, visualizations, which they have built, need to be used by someone, who is probably not as skilled 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 vs 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 to focus on this "comparison" question of "Small Data" coding vs "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. 4.2) is for being used in an IDE (Integrated Developer Environment), like the Spyder-IDE[10] from the Anaconda distribution[11] and applies the *Scikit-Learn Python Machine Learning Library*[12] (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. 4.3) is a "Big Data" example and will use a Docker environment[13] and apply the *Apache Machine Learning Library*[14], a scalable machine learning library. The mind-map from the second part will be extended and aligned to the second example.

[10] Spyder-IDE, https://www.spyder-ide.org/
[11] Anaconda distribution, https://www.anaconda.com/
[12] Scikit-Learn, https://scikit-learn.org/
[13] Docker environment, https://www.docker.com/
[14] 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 areas 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 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,** 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* and a *Certified Advanced Data Scientist*. 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* during my Erasmus studies in France.

I am also holding the *Certificate of ”Data Analysis with Python: Zero to Pandas*[15] (see fig. 1.3) which covers topics like data visualization and exploratory data analysis on the basis of Python[16], Numpy[17],

[15] Certificate of ”Data Analysis with Python: Zero to Pandas, https://jovian.ai/certificate/MFQTGOBQGM
[16] Python, https://www.python.org/
[17] Numpy, https://numpy.org/

Pandas[18], Matplotlib[19] and Seaborn[20]. I can recommend this course and I wish I would have found this course before I wrote this repository, because it was very helpful.

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

1.5 Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Figure 1.4: Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

[18] Pandas, https://pandas.pydata.org/
[19] Matplotlib, https://matplotlib.org/
[20] Seaborn, https://seaborn.pydata.org/

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.

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.

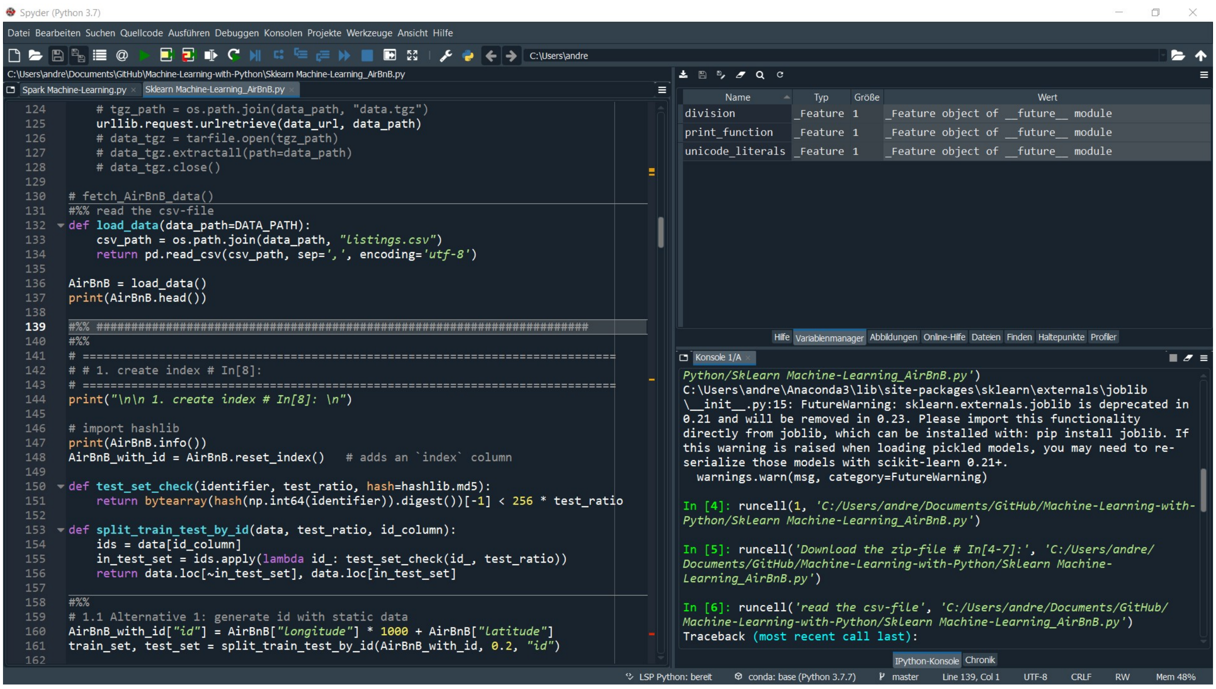


Figure 2.1: Spyder-IDE: An integrated development environment with debugger.

A Jupyter-Notebook runs in a browser (like Chrome, Edge, Firefox) and combines code and documen-

[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

tation. The advantage is, that it looks beautiful, is easy to share with other people. The disadvantage is, that there is no code-debugger. It look 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 Notebooks as well as the Spyder-IDE and a lot more Data Science applications, which might be useful for you later.

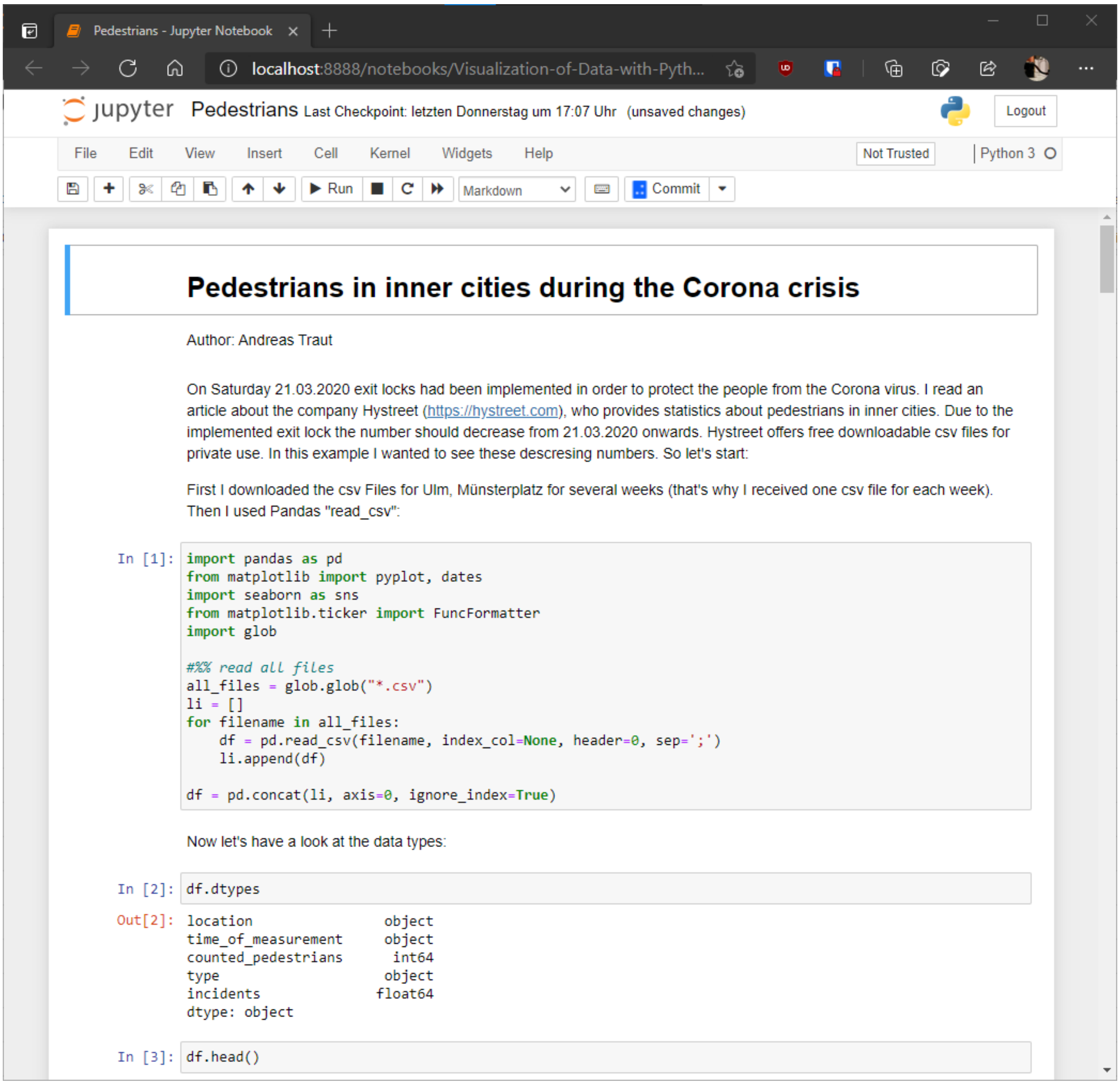


Figure 2.2: Jupyter-Notebook: Code and Documentation in one place.

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

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. :-)

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*

[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
[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

virtualization on Linux.”

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[12] 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.

[12] Docker-Curriculum, https://docker-curriculum.com/

3 Visualization of Different Datasets

3.1 Examples

In the first part of this repository I will work on examples. For the visualization tasks, which I wanted to do here, I exemplary used these different datasets:

1. A public dataset of the "Consumer Price Index" from the official statistics website of the "Bayerisches Landesamt für Statistik"[1].
2. A dataset of my own songs, which I listened to (66'955 songs since 2016, downloaded from LastFM[2].
3. An artificially treated dataset of "Marathon run-times", where I showed how systematics in the data can be found.
4. The number of pedestrians in inner cities when the Corona-exit-lock had been implemented.
5. The data from the Deutsche Bahn API to monitor status of their station elevators.
6. A very brief introduction into visualization of Big Data.

The examples are available:

- as `.py` files for being used for example in an Spyder-IDE[3] and
- as `.ipynb` files, which are Jupyter-Notebooks[4].

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[5]. The CSV-file has the following format:

---

1 Bayerisches Landesamt für Statistik, www.statistikdaten.bayern.de

2 LastFM, www.last.fm

3 Spyder-IDE, https://www.spyder-ide.org/

4 Jupyter-Notebook, https://jupyter.org/

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

	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 3.1: 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). It 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”) to a number/date.

```
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, '')
```

I lived in Switzerland and bought my computer there. Unfortunately my keyboard and also my language setting is `'German_Switzerland.1252'` and therefore importing the CSV file with the `pandas.read_csv` would not work, because in Switzerland and Germany have different separators (comma versus point): 29,6 (German) should become 29.6 (Switzerland).

```
df=pd.read_csv('61111-202z-bearbeitet.csv',
               encoding="latin-1",
               names=['year', 'month', 'value'])
df.head()
```

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

It is a good exercise to learn how to convert different language settings. We have to define `converters`, which I named `myMonthConverter` (converts Januar to 1, Februar to 2, …) and `myValueConverter` (converts 29,6 to 29.6).

```
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')

#Read the csv. Comma separated. Encoding=latin-1
#Convert 'month' to numbers and 'value' to floats.
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

But the type of the column `value` is not a number, but an object: `df.dtypes`

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

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

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

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.

```
#Create new column 'date' based on 'year' and 'month' and convert to date
#Create new column 'datenum' as float for being used in the plot
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.dtypes
```

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

```
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

Finally I have everything for the regression Plot (regplot). 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:

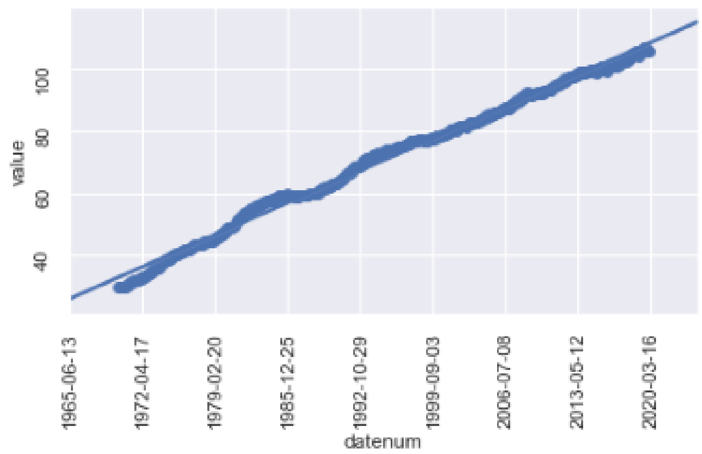


Figure 3.2: Consumer Price Index Figure

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))
```

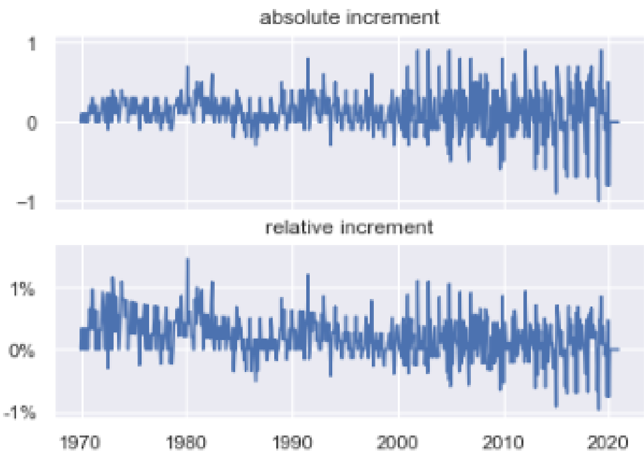


Figure 3.3: Consumer Price Figure - absolute and relative increments

In the picture on the left I would say that there is nothing noticeable (apart from a fairly steady rise in consumer prices over the whole period from 1970 until 2020). A bit disappointing so far.

But now having a look at the increments on the right side (absolute and relative increments) I found: perhaps one can say that the consumer prices grew more evenly between 1970 and 1995 and that the growth was almost entirely positive (above 0=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. 3.4, 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).

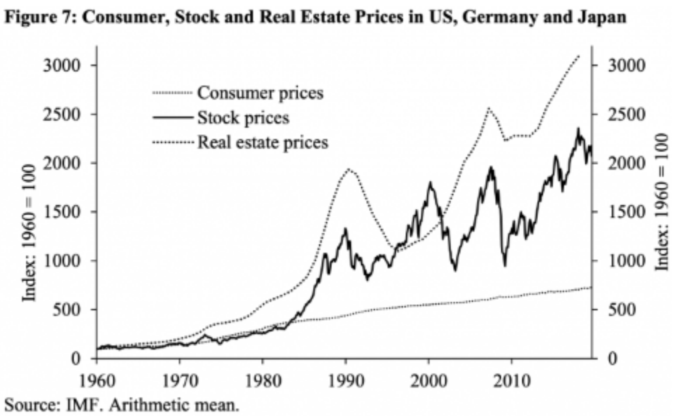


Figure 3.4: Consumer price index versus Stockprices, Real estate prices

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[6].

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[7] to upload my music statistics (so called "scrobbling").

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

[7] LastFM, www.last.fm

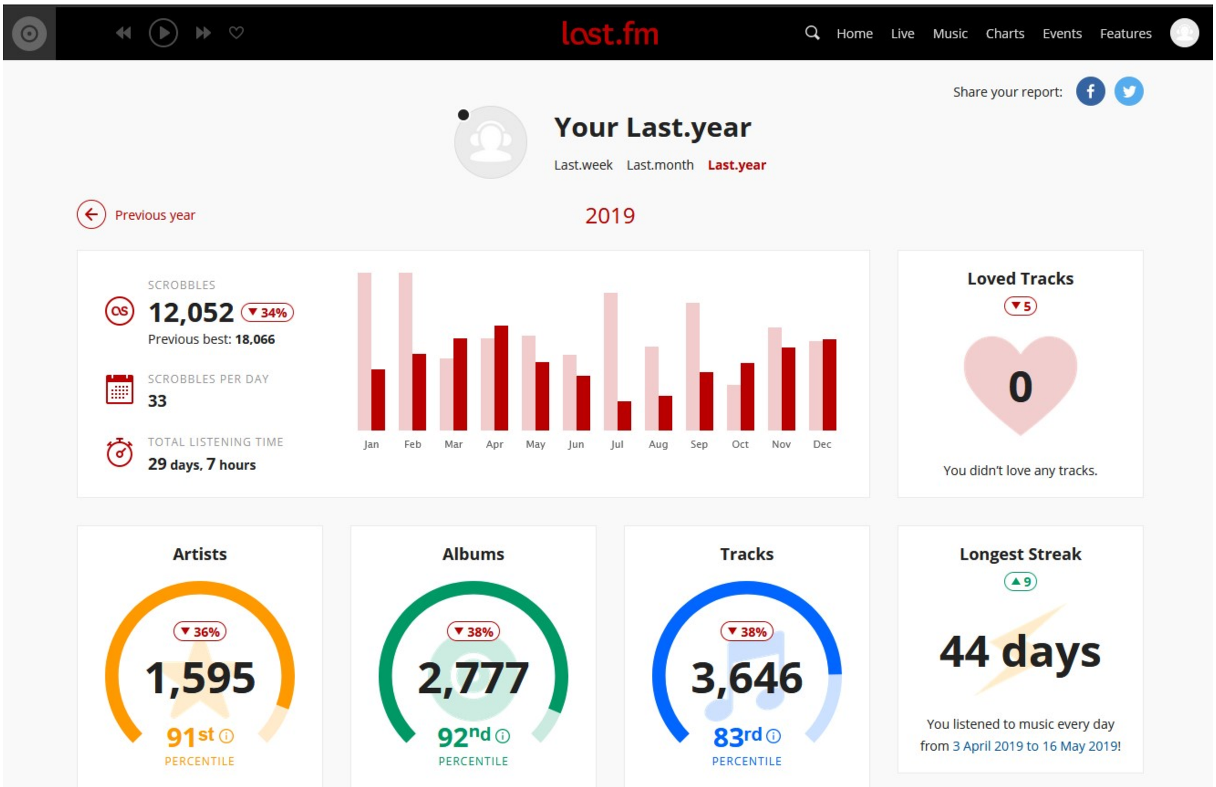


Figure 3.5: Last-FM Music Statistics - Overview year 2019

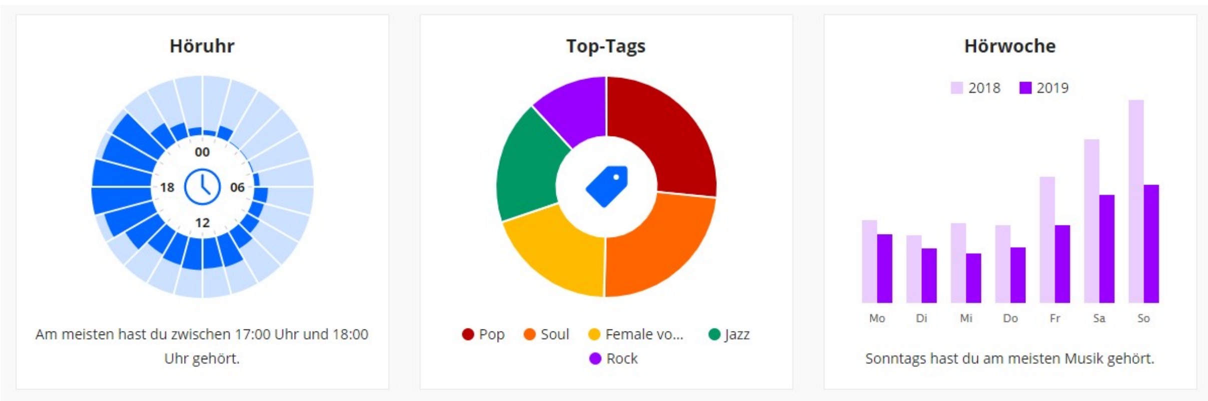


Figure 3.6: Overview listening clock 2019

Last-FM is creating nice graphics for 2019 as shown in fig. 3.5. 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. :-)

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 3.7: 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})
```

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".

```
#%% 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.
```

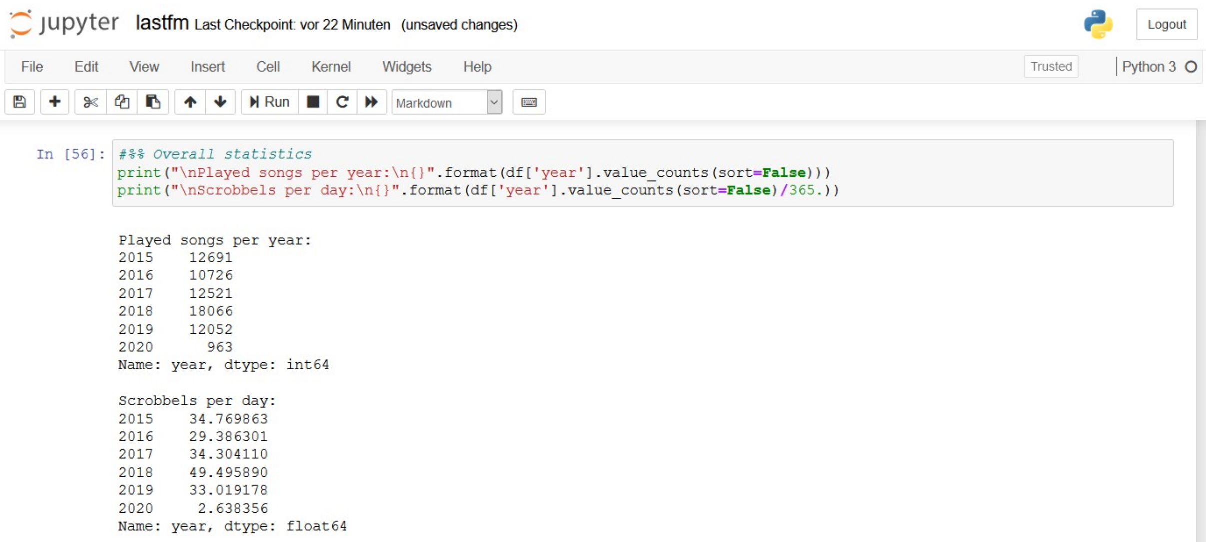


Figure 3.8: Last-FM Music Statistics - Overall statistics

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()
```

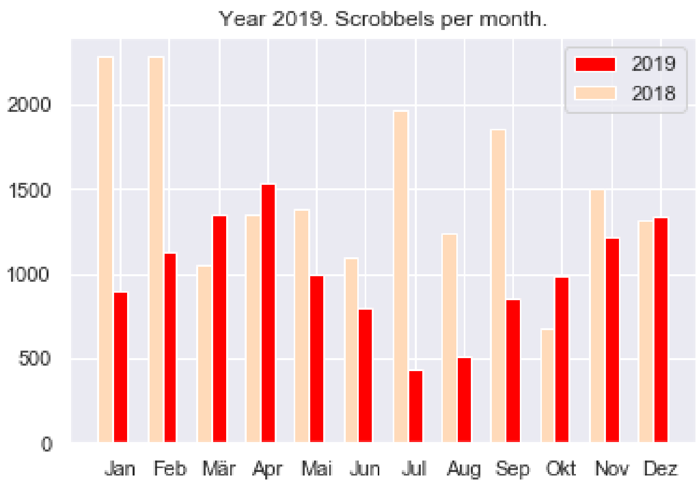


Figure 3.9: Last-FM Music Statistics - Reproduced Statistics of 2019

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 3.10: Last-FM Music Statistics - Overview year 2018

My graphics for 2018:

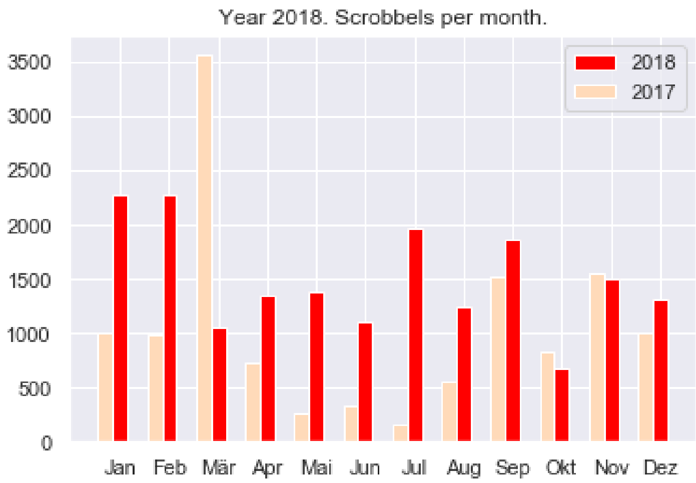


Figure 3.11: Last-FM Music Statistics - Reproduced Statistics of 2018

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 3.12: Last-FM Music Statistics - Overview year 2017

My graphics for 2017:

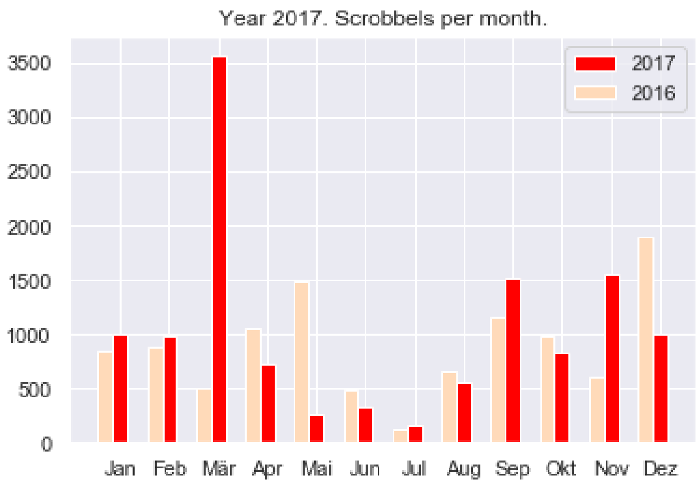


Figure 3.13: Last-FM Music Statistics - Reproduced Statistics of 2017

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()
```

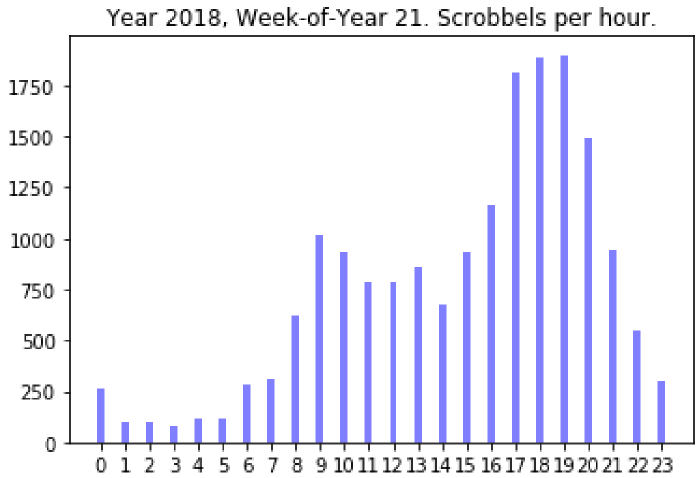


Figure 3.14: Overview listening clock 2018 (week 21)

On my GitHub profile you can find my Jupyter-Notebook for this example[8]

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?

[8] 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:

```
#%% 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')
```

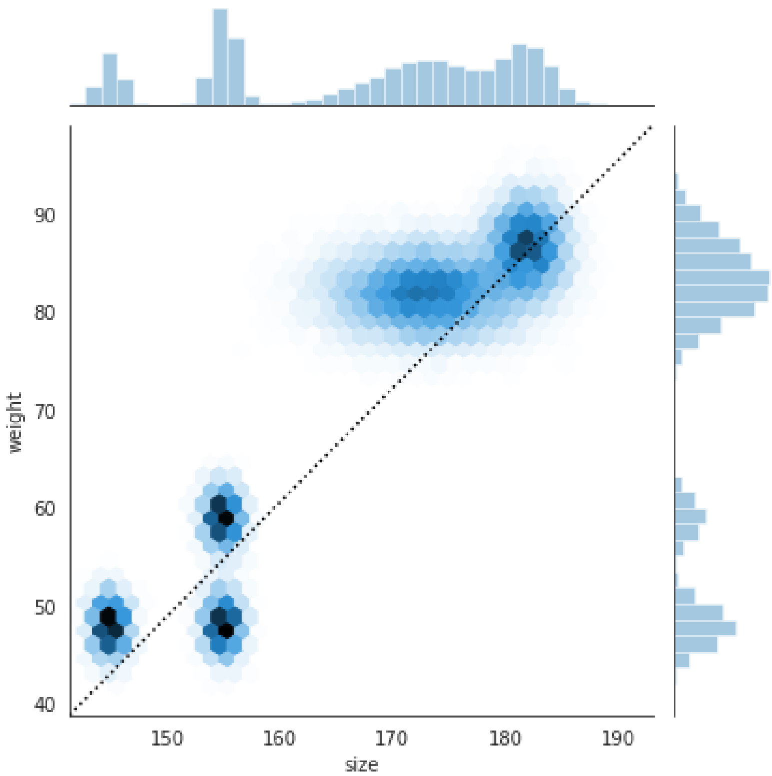


Figure 3.15: Marathon Example - Seaborn Jointplot

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:

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

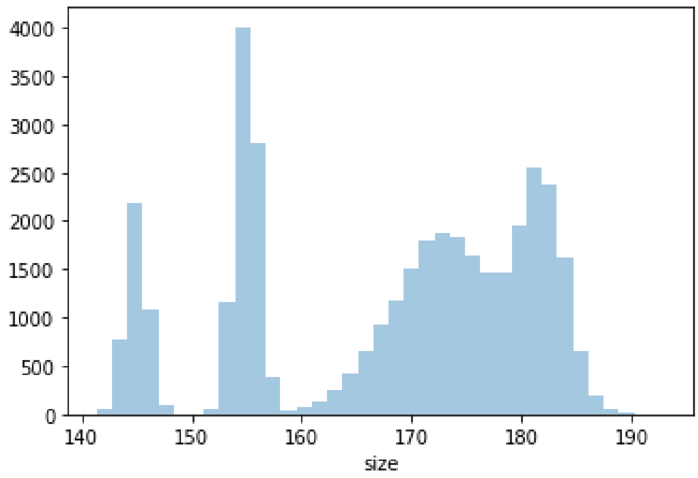


Figure 3.16: Marathon Example - Seaborn Histogram “size”

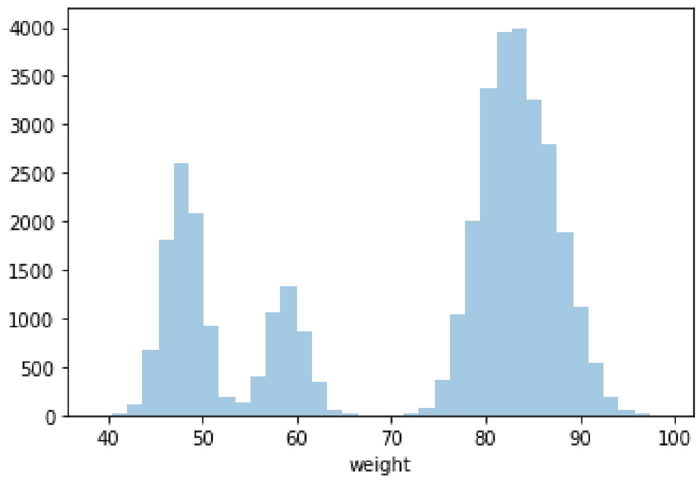


Figure 3.17: Marathon Example - Seaborn Histogram “weight”

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):

```
#%% 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();
```

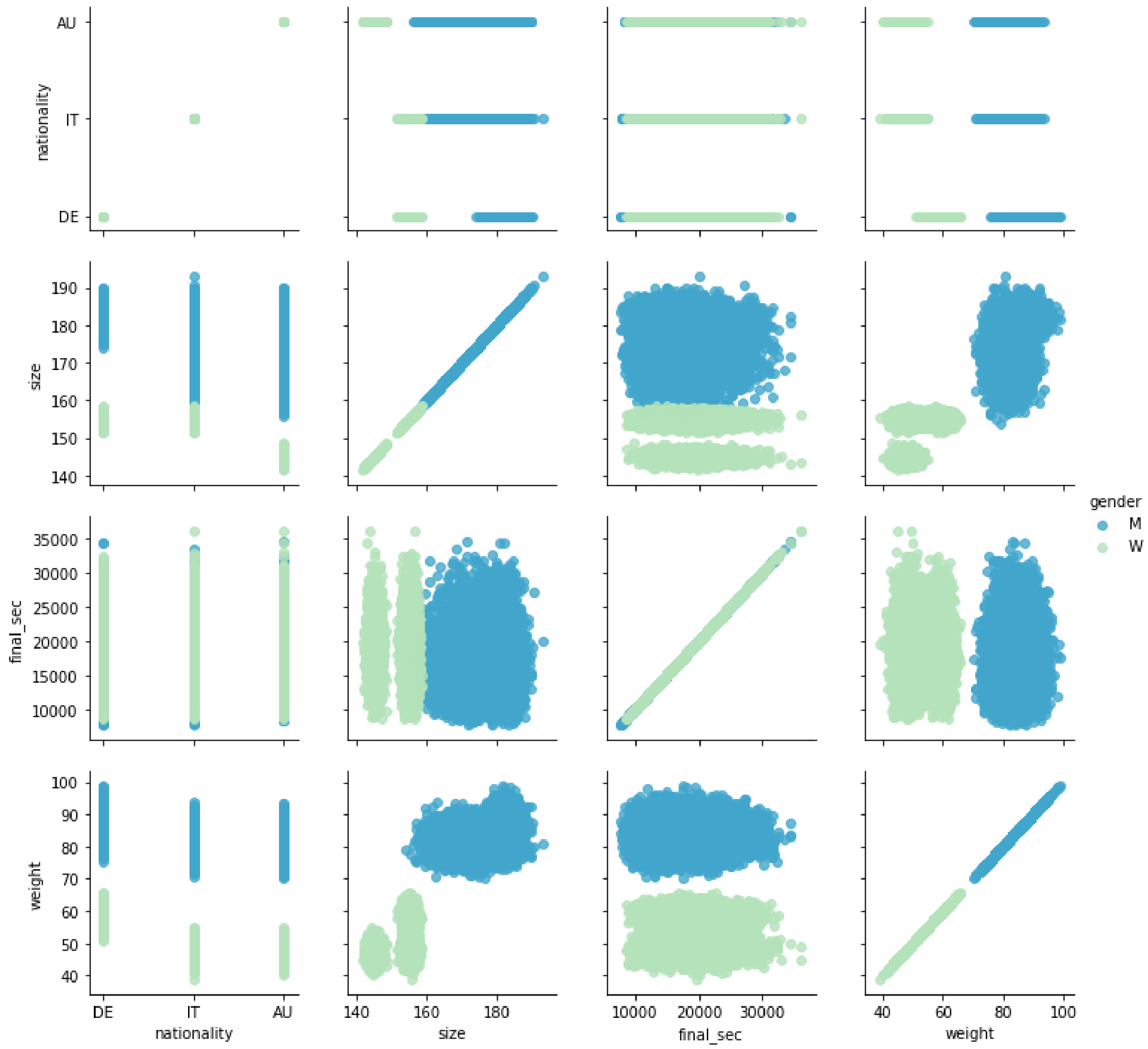


Figure 3.18: Marathon Example - Seaborn PairGrid

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()
```

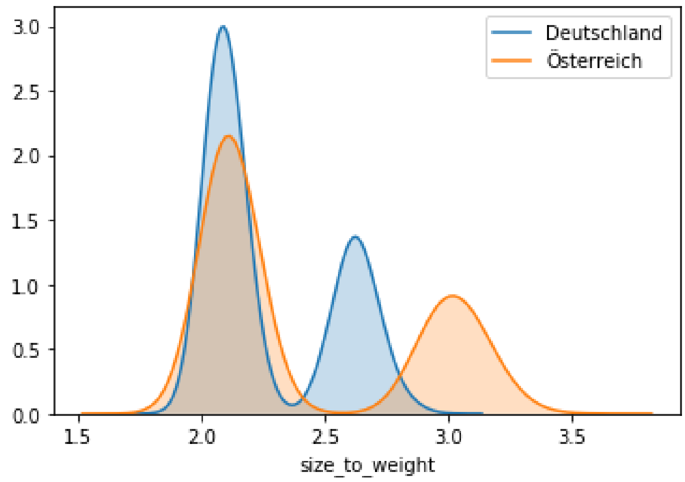


Figure 3.19: Marathon Example - Seaborn Kernel Density "size-to-weight"

```
#%% 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');
```

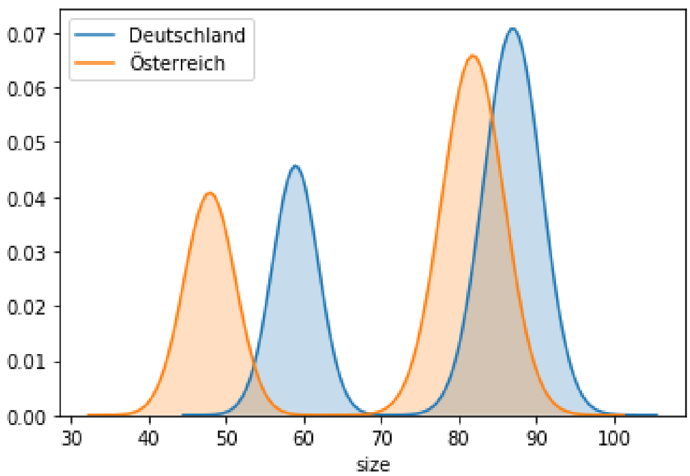


Figure 3.20: Marathon Example - Seaborn Kernel Density "size"

Now it would interesting to see some regression plots (step 9). On the left side for men and on the right side for women:

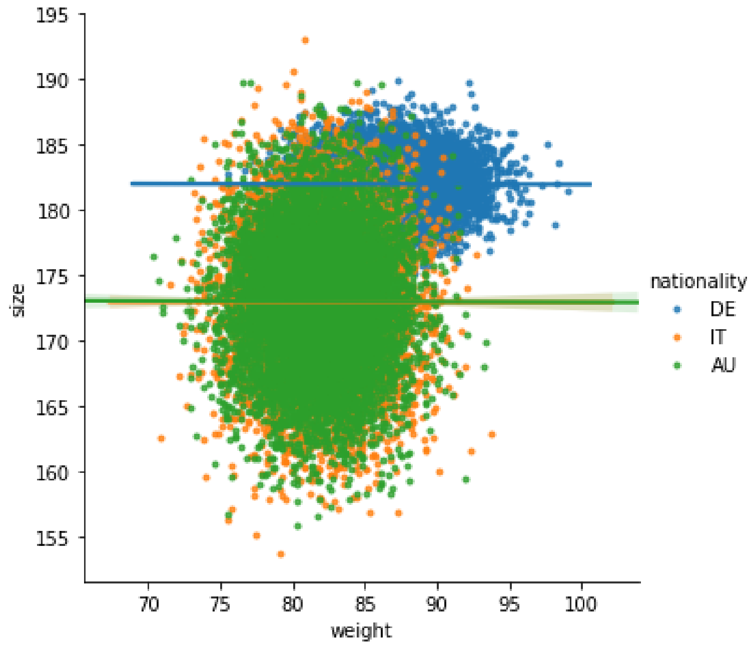


Figure 3.21: Marathon Example - Seaborn Regression Plots “men”

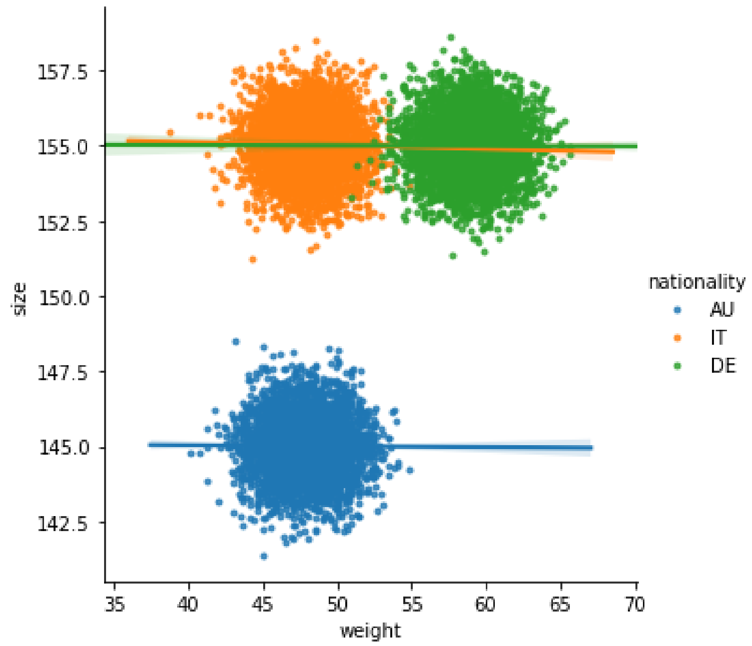


Figure 3.22: Marathon Example - Seaborn Regression Plots “women”

```
#%% 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 the left chart, 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).

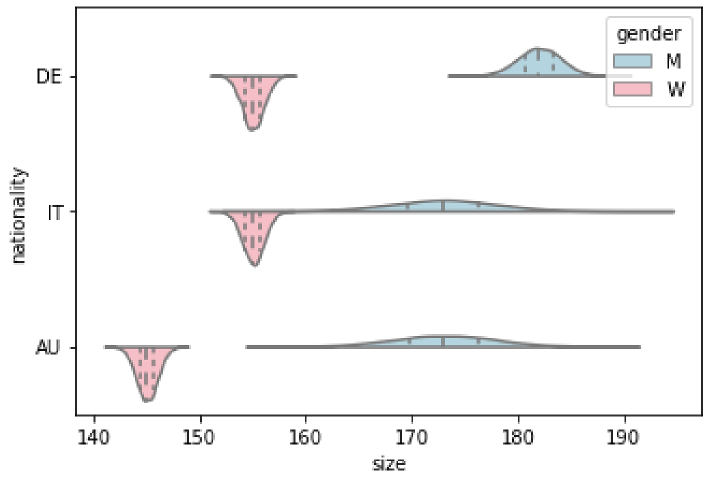


Figure 3.23: Marathon Example - Seaborn Violinplot "size"

```
#%% 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"]);
```

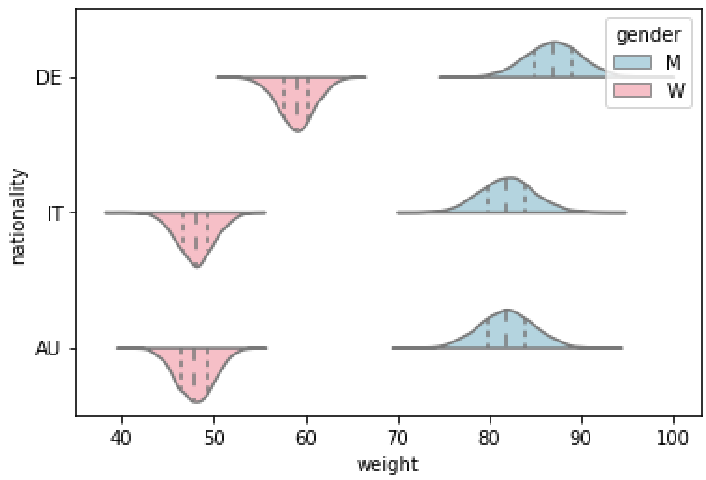


Figure 3.24: Marathon Example - Seaborn Violinplot "weight"

In the right chart 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[9] and Codecov[10]. Travis and Codecov provide small icons, like the following:

Figure 3.25: Build Status passing

The advantage of doing this is: when I share my python code or Jupyter-Notebooks on code-sharing platforms, like GitHub[11] 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).

On my GitHub profile you can find my Python-File for this example[12]

[9] Travis, https://travis-ci.com/

[10] Codecov, https://codecov.io/

[11] GitHub, https://github.com

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

3.1.4 Pedestrians during the first Corona-Lockdown

On Saturday 21.03.2020 exit locks had been implemented in order to protect the people from the Corona virus. I read an article about the company Hystreet[13], who provides statistics about pedestrians in inner cities. Due to the implemented exit lock the number should decrease from 21.03.2020 onwards. Hystreet offers free downloadable csv files for private use. In this example I wanted to see these decreasing numbers. I downloaded the statistics for several week for Ulm, Münsterplatz. The following plot shows the number of pedestrians walking on Ulm, Münsterplatz. From 21.03.2020 onwards the number of people decreased:

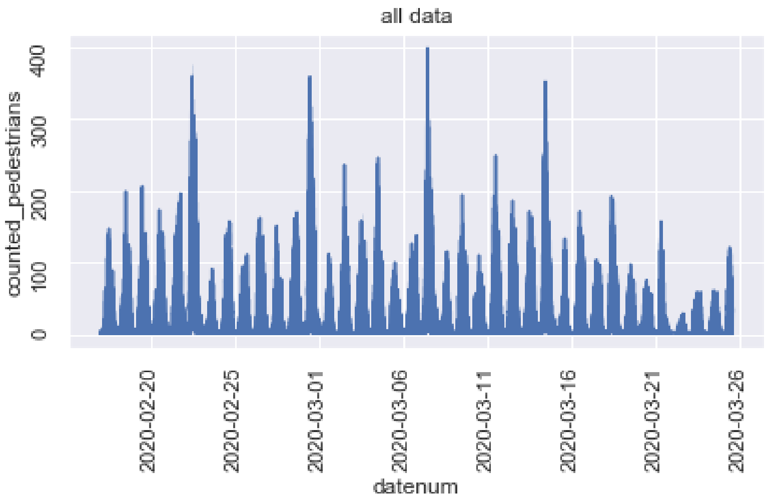


Figure 3.26: Pedestrians in Corona-Lockdown - Frequency

The numbers for Ulm, Münsterplatz in a barplot look like this:

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

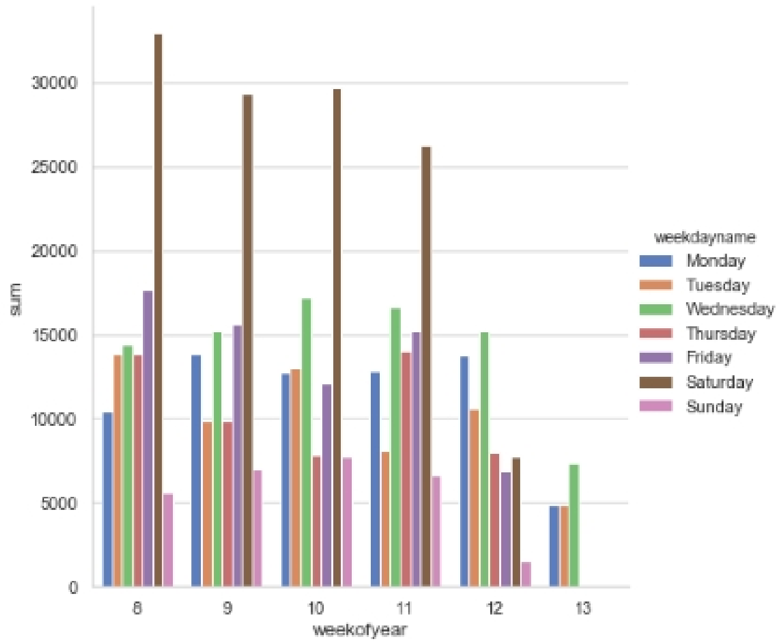


Figure 3.27: Pedestrians in Corona-Lockdown - Barplot “Ulm, Münsterplatz”

And here are some more graphics for München and Augsburg

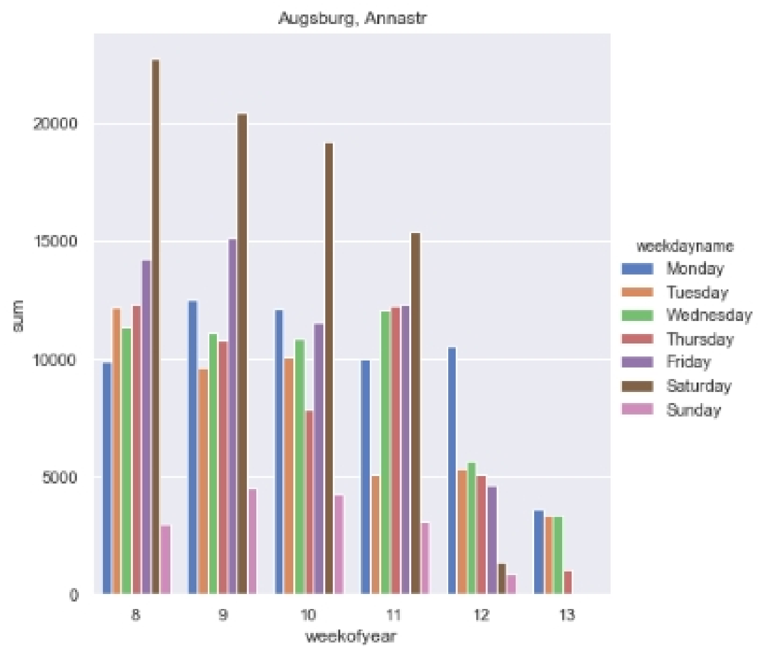


Figure 3.28: Pedestrians in Corona-Lockdown - Barplot “Augsburg”

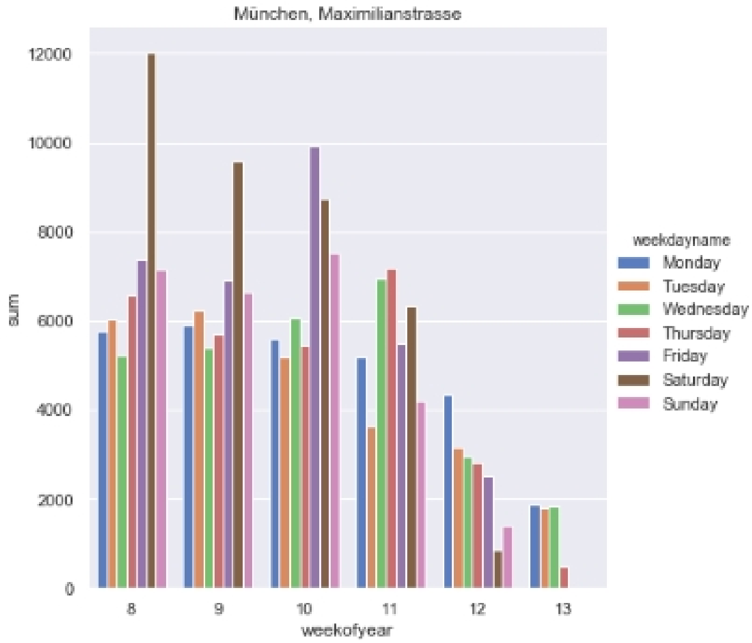


Figure 3.29: Pedestrians in Corona-Lockdown - Barplot "München, Maximilianstrasse"

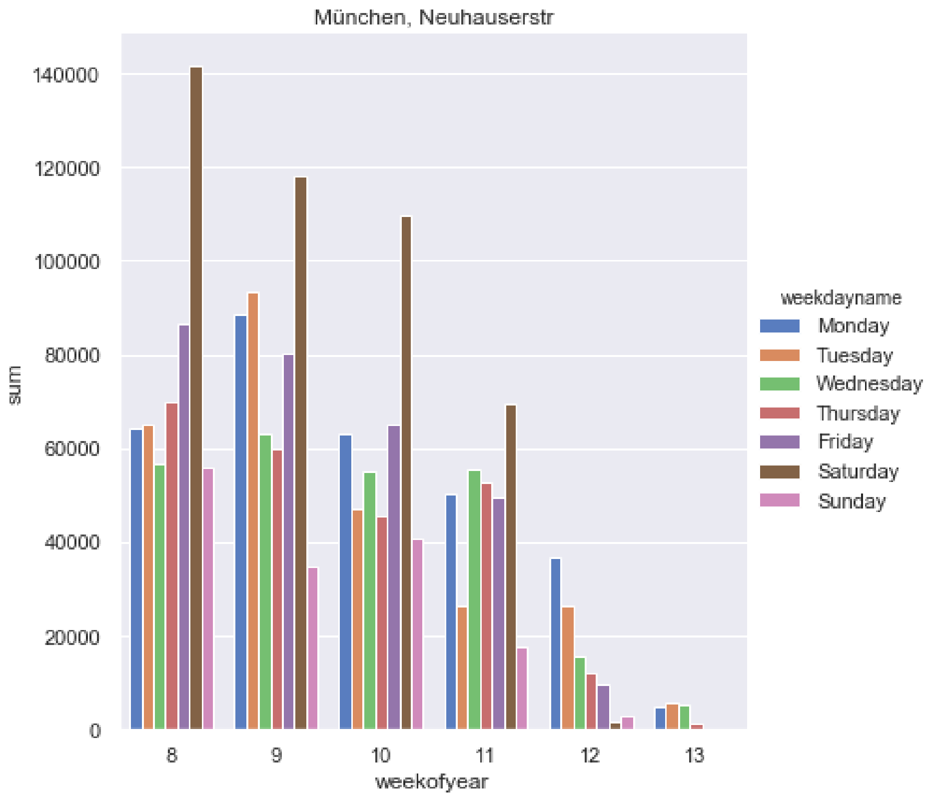


Figure 3.30: Pedestrians in Corona-Lockdown - Barplot "München, Neuhausstrasse"

Just to see, how it looks like for a longer time horizon: from August 2019 to March 2020. Here are the plots for Ulm, Münsterplatz and Munich, Maximilanstr:

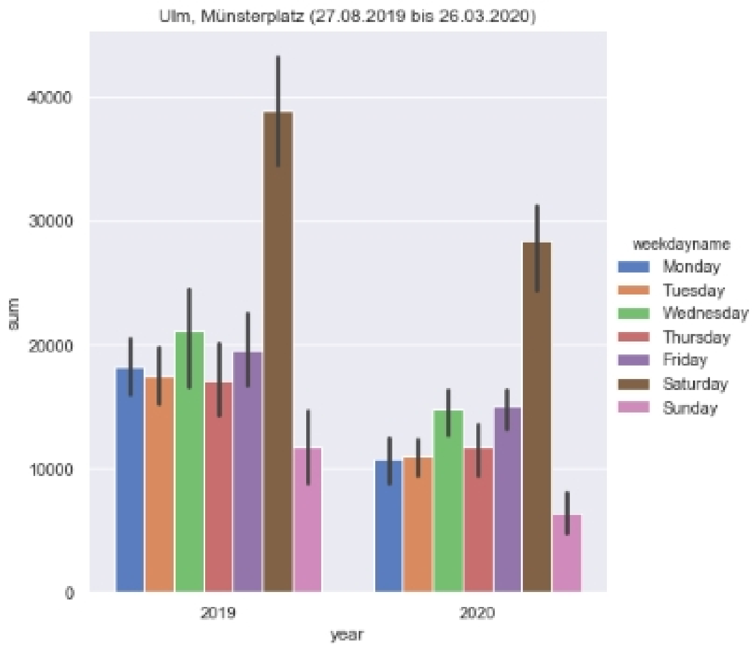


Figure 3.31: Pedestrians in Corona-Lockdown - Barplot "Ulm" 08/2019 to 03/2020

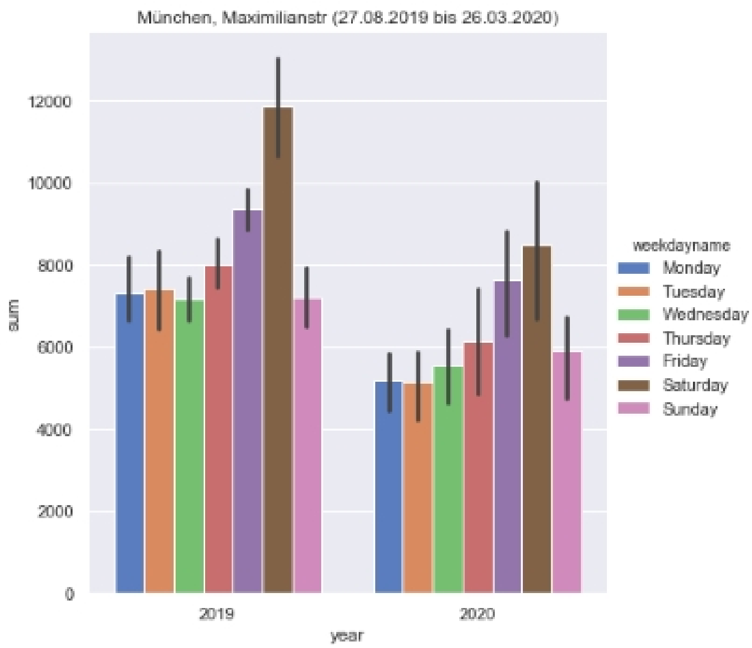


Figure 3.32: Pedestrians in Corona-Lockdown - Barplot "München, Maximilianstrasse" 08/2019 to 03/2020

As there are more datapoints in 2019 (126 from 27.08.2019 to 31.12.2019, compared to 85 from 01.01.2020 to 26.03.2020) the bars are higher in 2019. Here is an update of the above graphics as-of 07.04.2020 for Ulm, Münsterplatz:

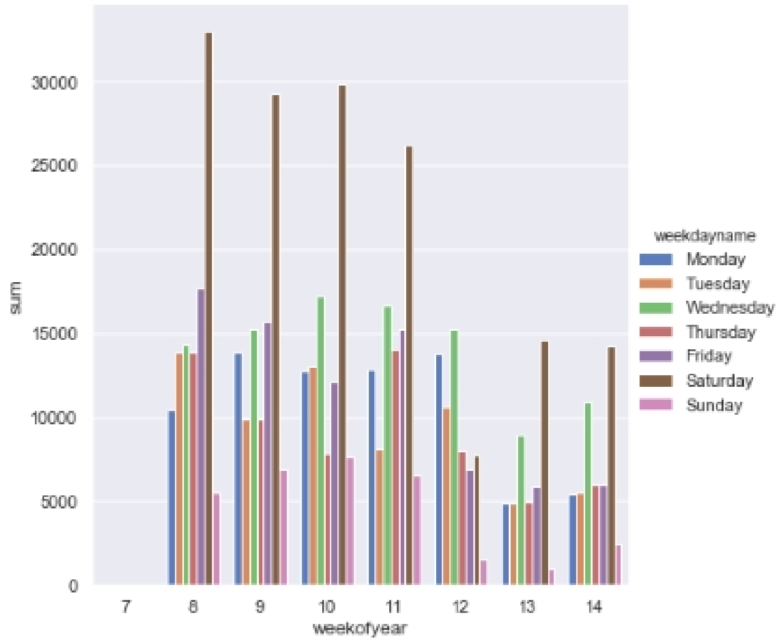


Figure 3.33: Pedestrians in Corona-Lockdown - Barplot "Ulm" April 2020

On my GitHub profile you can find my Python-File for this example[14]

3.1.5 Station Elevators of Deutsche Bahn: work with APIs

The visualization is sometimes a bit difficult, because the dataset is not yet available in the form you need to have them. 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[15], but I would need to work with a Python-Code (`.py` File) to extract the information I needed:

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

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-

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

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

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.

Have a look into my Python file to learn:

- how to access data via an API from Deutsche Bahn
- how extract station names, number of elevators, status of elevators and longitude/latitude
- how to use this information for visualization techniques

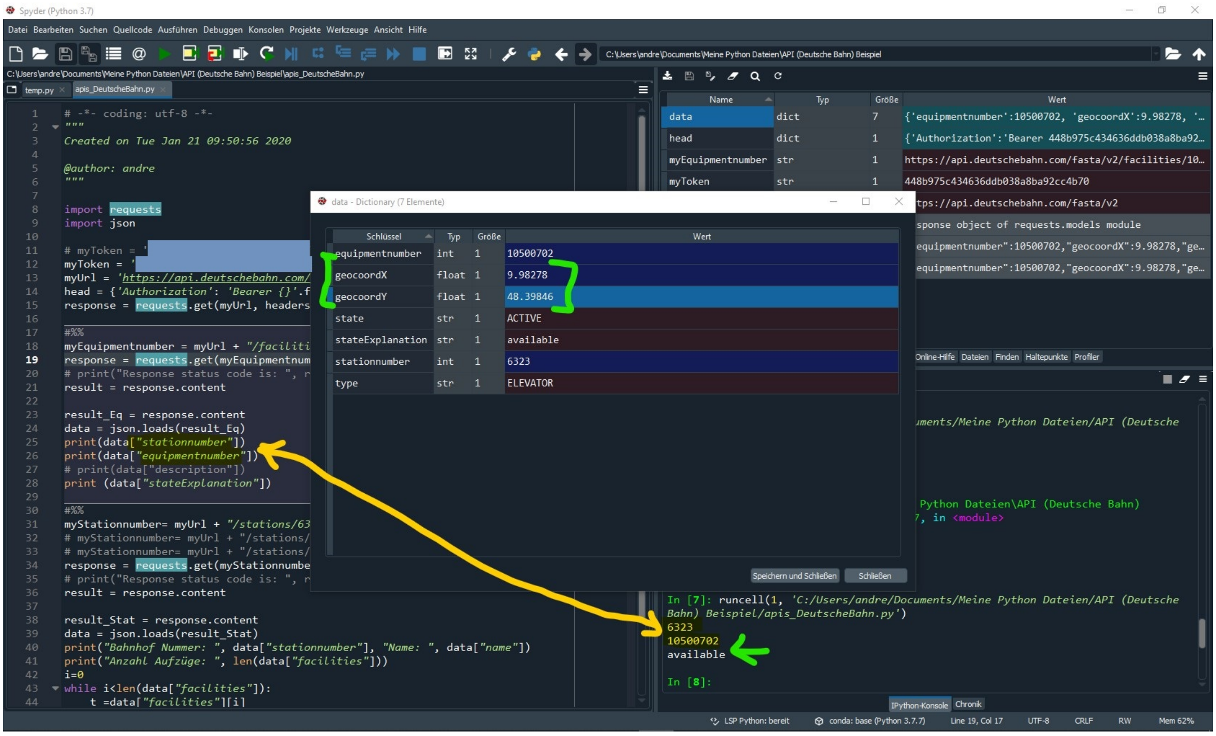


Figure 3.34: Deutsche Bahn - Elevators 1

After having understood and having extracted these meta data (station number, equipmentnumber) I was able to visualize them: as you can see in the screenshot fig. 3.35, when I handed over a stationnumber (e.g. 6323) 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. Taking these and using for example GPS-Coordinates[16] you can easily visualize these longitudes and latitudes as shown in fig. 3.36.

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

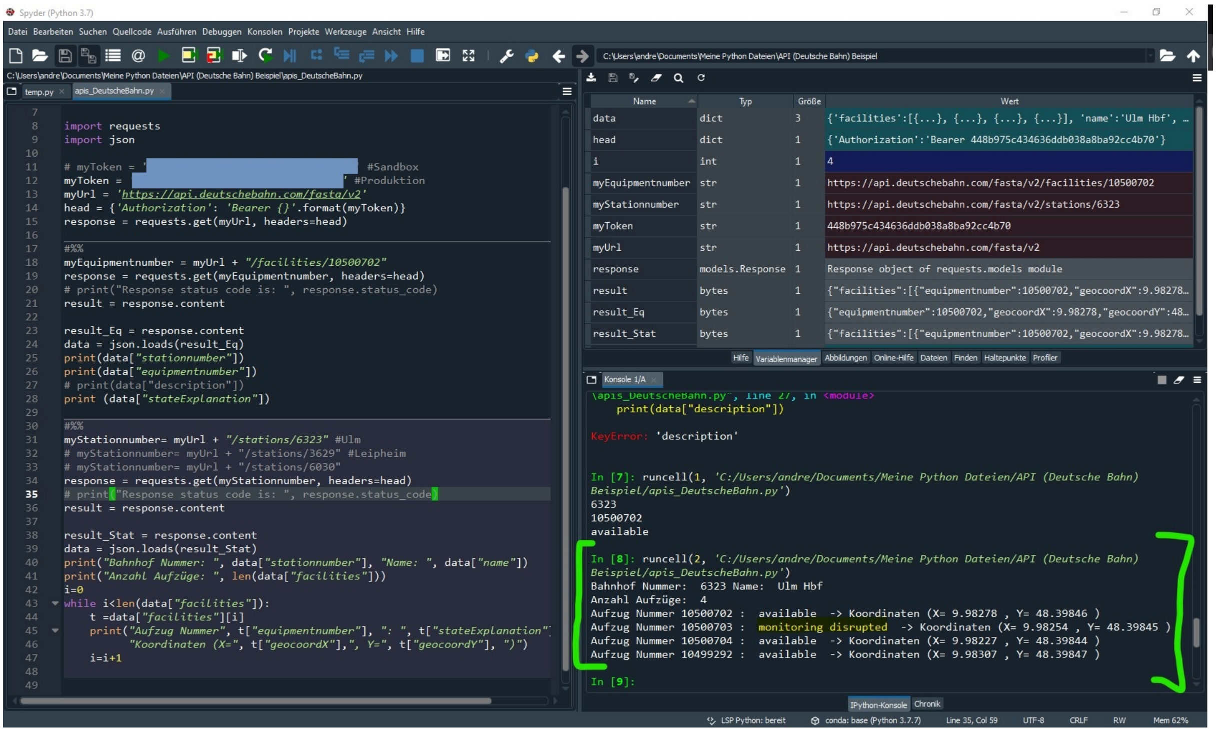


Figure 3.35: Deutsche Bahn - Elevators 2

Another possibility would have been to use Geopy[17] 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"
```

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

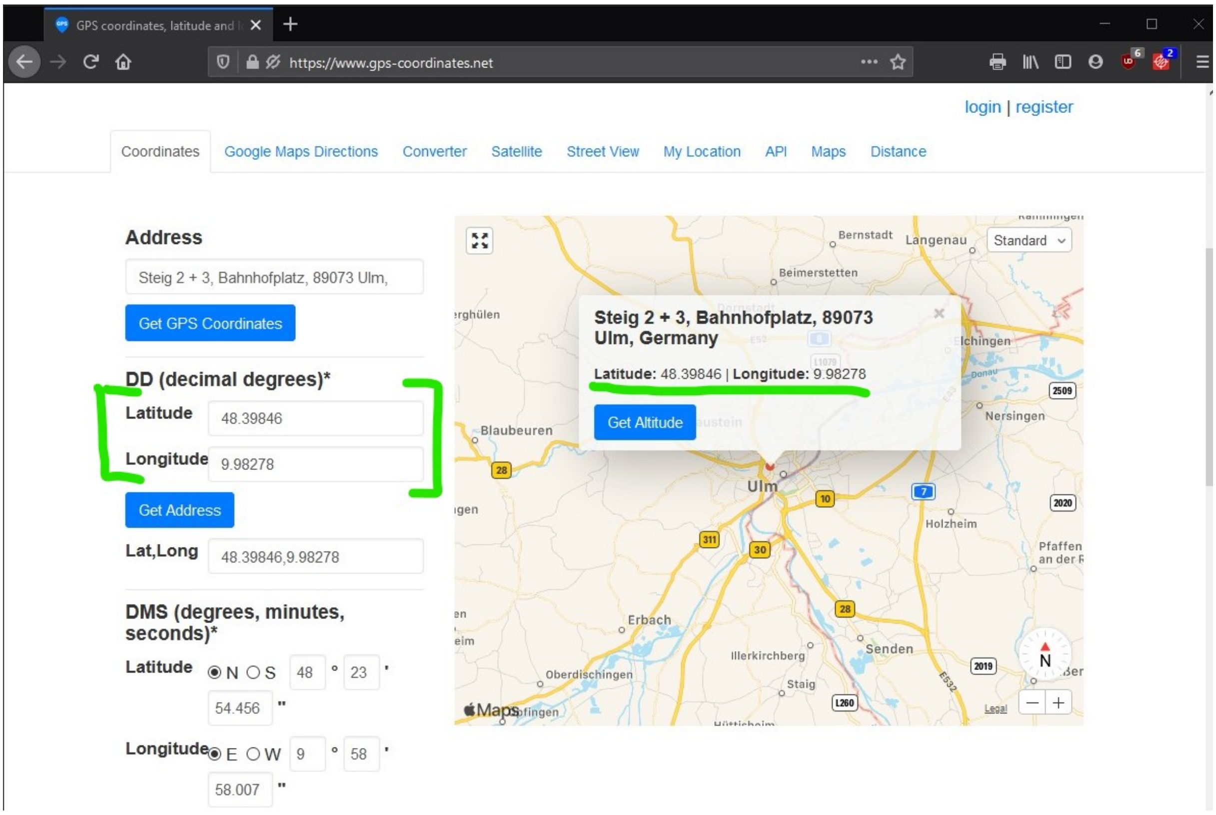


Figure 3.36: Deutsche Bahn - Latitude and Longitude

Building a longer history (not only one extraction of data, but many) 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 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. I think this is another example where visualization techniques are applied.

3.1.6 Interactive Pivots

Another nice possibility to visualize pandas data frames is by using the “*pivottablejs*”[18] 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:

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

"*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.

```
# Intall: !pip install pivottablejs
import pivottablejs
pivottablejs.pivot_ui(pandas_df)
```

Now you can drag and drop the columns and also use the filter select boxes:

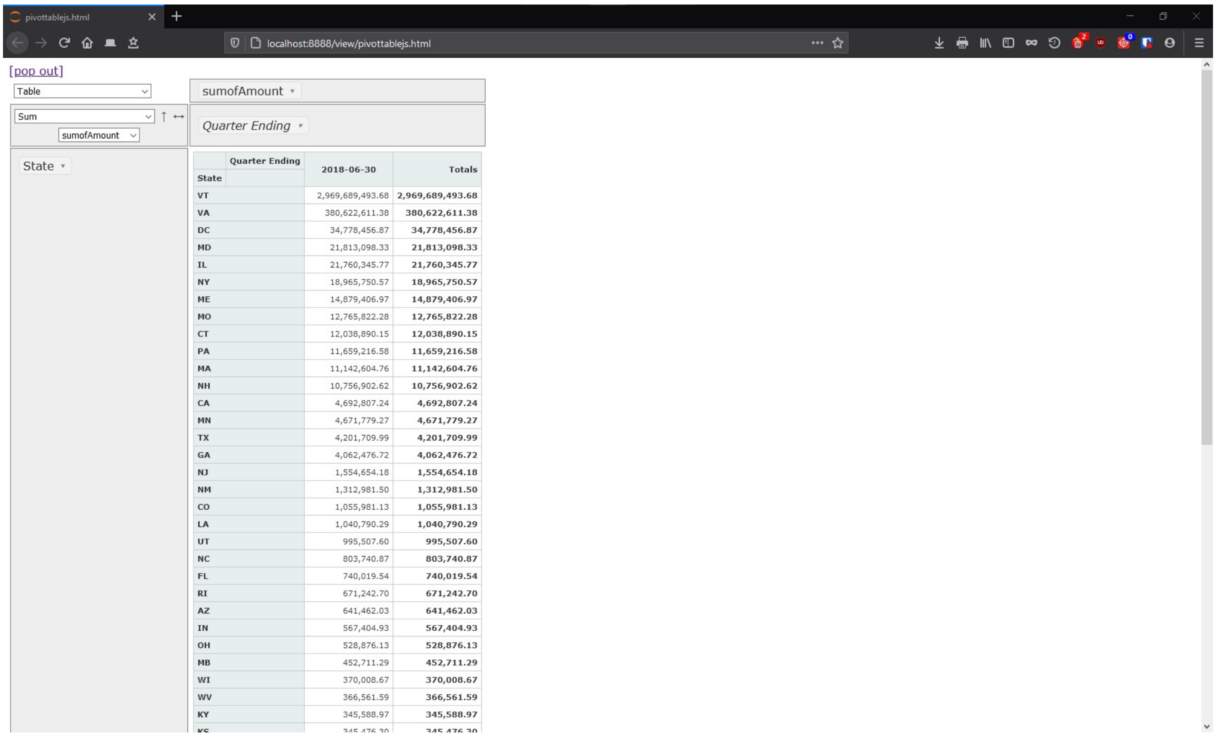


Figure 3.37: Interactive Pivots - Grouping and aggregating

Creating graphs is also possible:

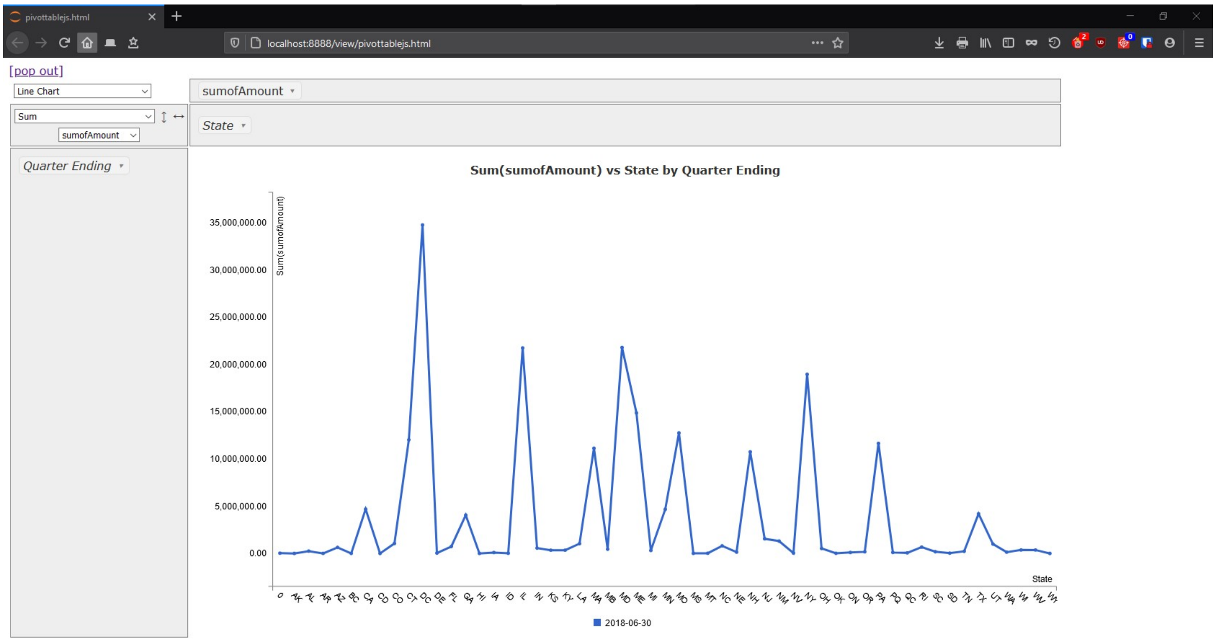


Figure 3.38: Interactive Pivots - Graph

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 [19] (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[20] 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[21], 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. 4.3.1.

```
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()
```

Next step is to read the huge csv file:

```
import os
datapath = os.environ['PWD']
filename = datapath + "/data/Vermont_Vendor_Payments.csv"
```

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

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

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

```
#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
```

Or you might want to group by `Quarter Endings`:

```
spark_df = data.groupBy('Quarter End-
↪ ing').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
```

Or you might select the top 6 states (VT, MA, NH, PA, VA, GA) from this aggregation:

```
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 End-
↪ ing').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[22] 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 have only a small amount of data points left. For example:

	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 3.39: 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 End-
↪ ing').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.

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

```
spark_df = data.groupBy('Quarter End-
 ↪ ing').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')
```

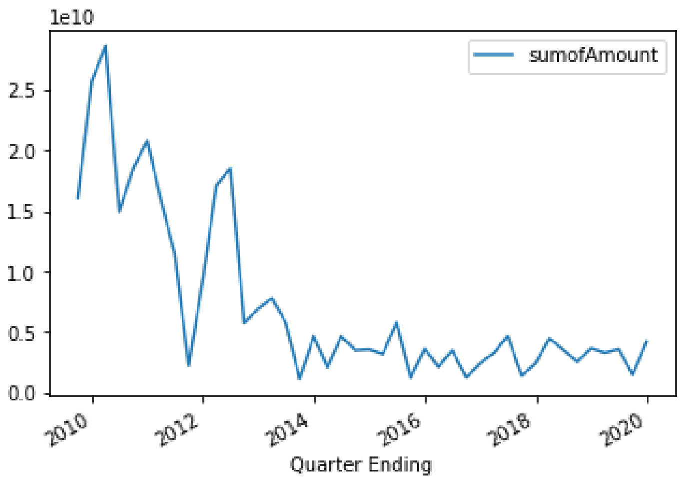


Figure 3.40: Big Data Visualization - Line Chart

```
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()
```

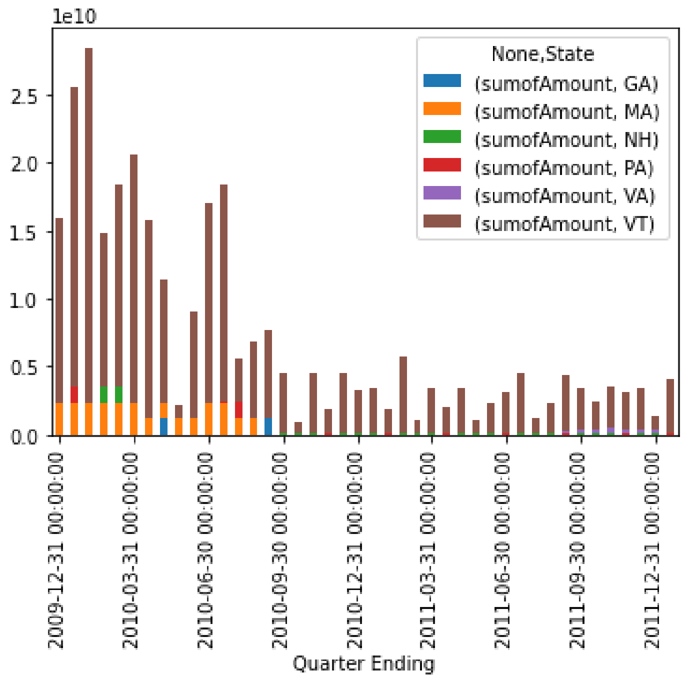


Figure 3.41: Big Data Visualization - Stacked Barplot

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[23] 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[24], 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**.

There are different solutions: when you are using the programming language R then the combination

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

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

of Tidyverse[25] and Shiny-App[26] 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[27] reveals a strong increasing popularity for Streamlit):

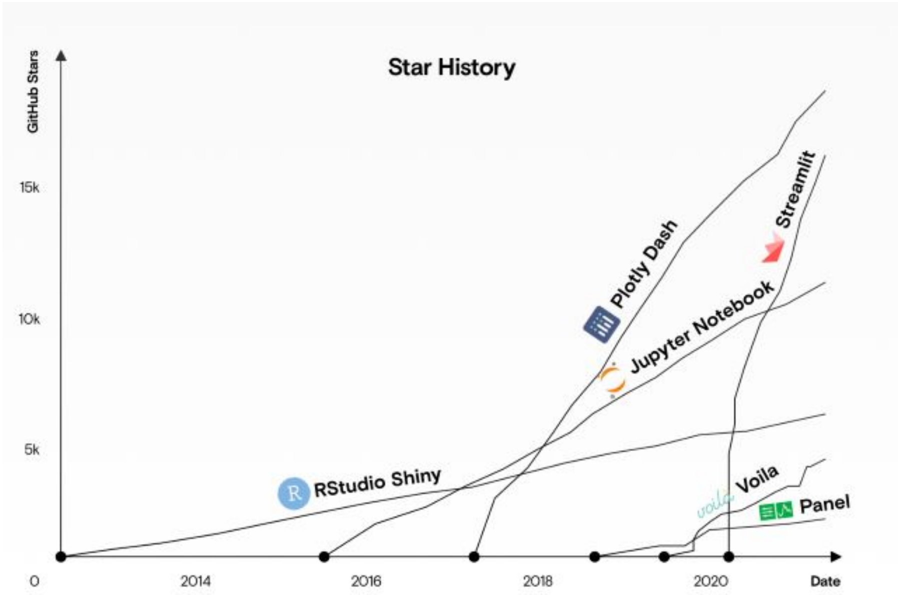


Figure 3.42: Data App - Comparison of Streamlit, RStudio-Shiny and others, Quelle: DataRevenue-Blog

I tested Streamlit[28] 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"[29] and play around yourself. I uploaded the results of these two datasets (the left and right side of the window below) in the results folder.

[25] Tidyverse, https://www.tidyverse.org/
[26] Shiny-App, https://shiny.rstudio.com/
[27] Data-Revenue, https://www.datarevenue.com/de-blog/streamlit-vs-dash-vs-shiny-vs-voila-vs-flask-vs-jupyter
[28] Streamlit, https://www.streamlit.io/
[29] Marathon Streamlit "data app", https://share.streamlit.io/andreastraut/visualize-results-in-apps/main/app_Example_Marathon_extended.py

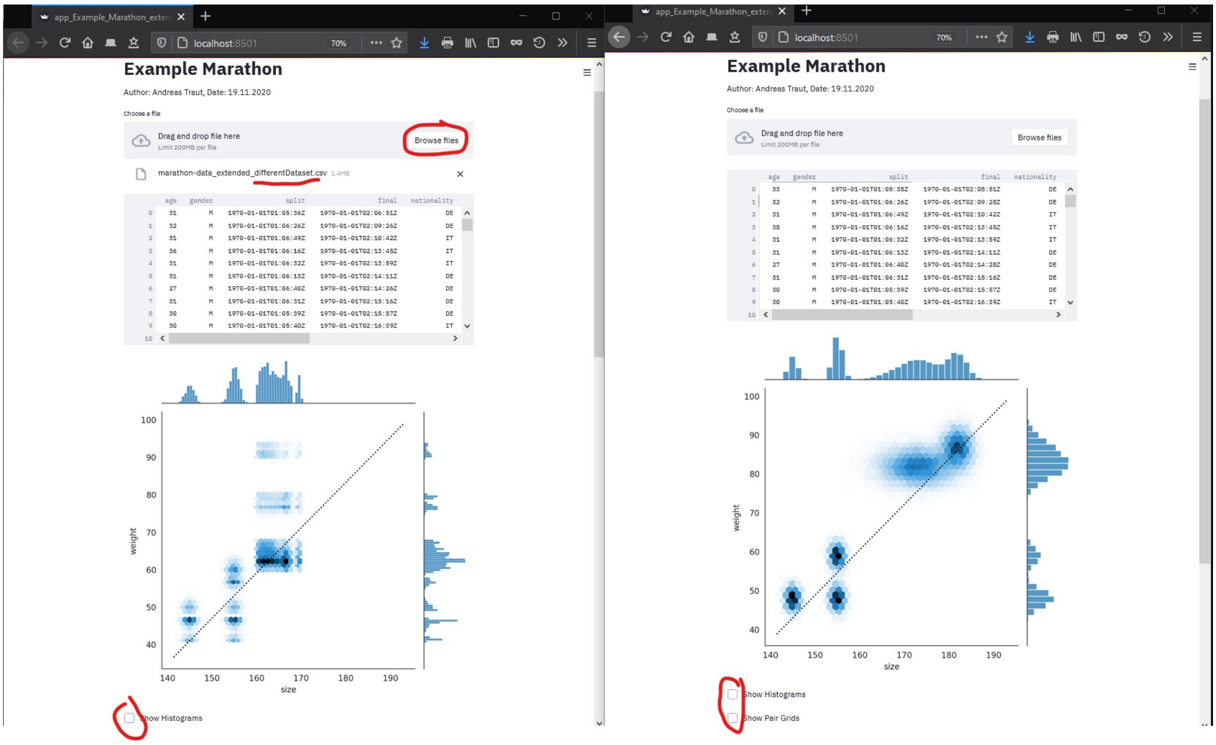


Figure 3.43: Data App - Marathon Example

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[30] and implemented a Streamlit app[31] in less than half an hour. This is why I love Streamlit: highly efficient, lovely design and easy to deploy.

[30] Wikipedia SEIR model, https://de.wikipedia.org/wiki/SEIR-Modell
[31] SEIR modell app, https://share.streamlit.io/andreastraut/visualize-results-in-apps/main/app_SEIR_model

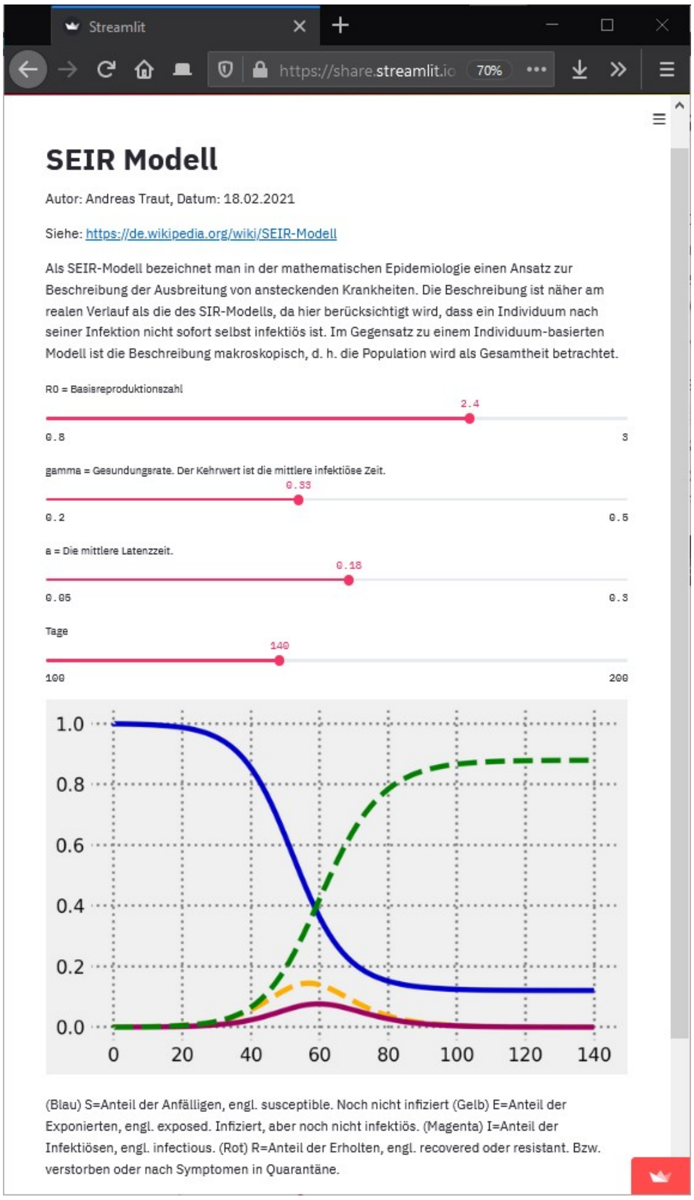


Figure 3.44: Data App - SEIR Model Example

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.

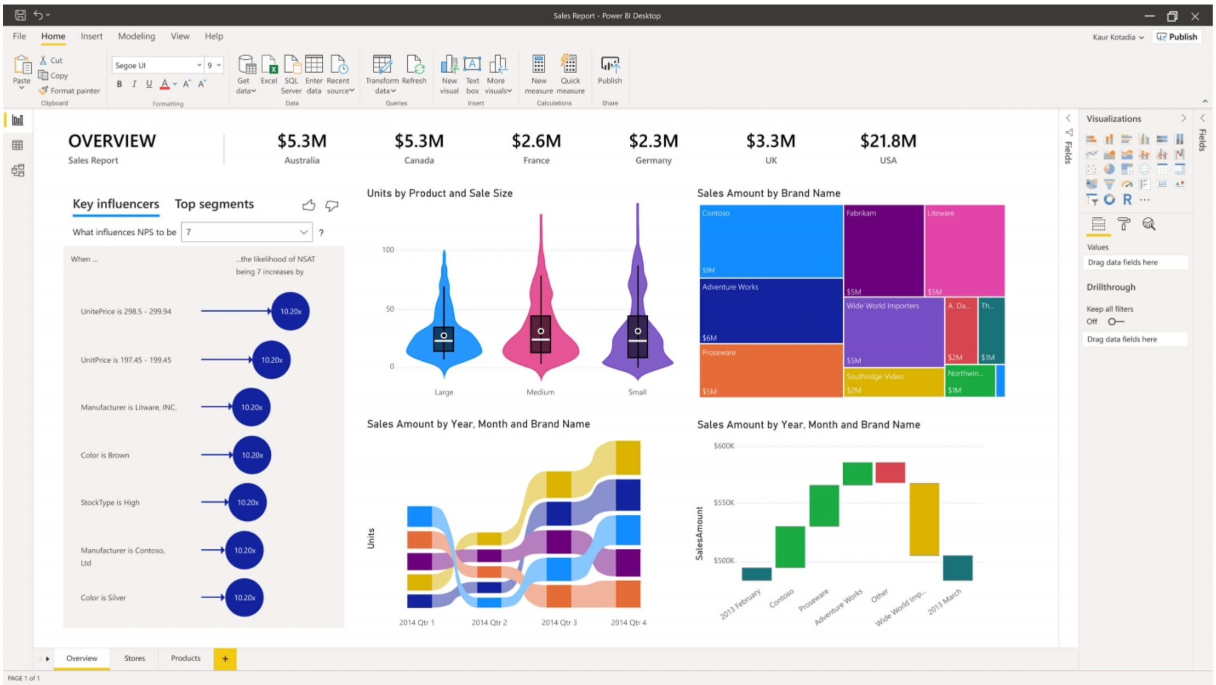


Figure 3.45: Professional BI - Power BI

Figure 3.46: 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.

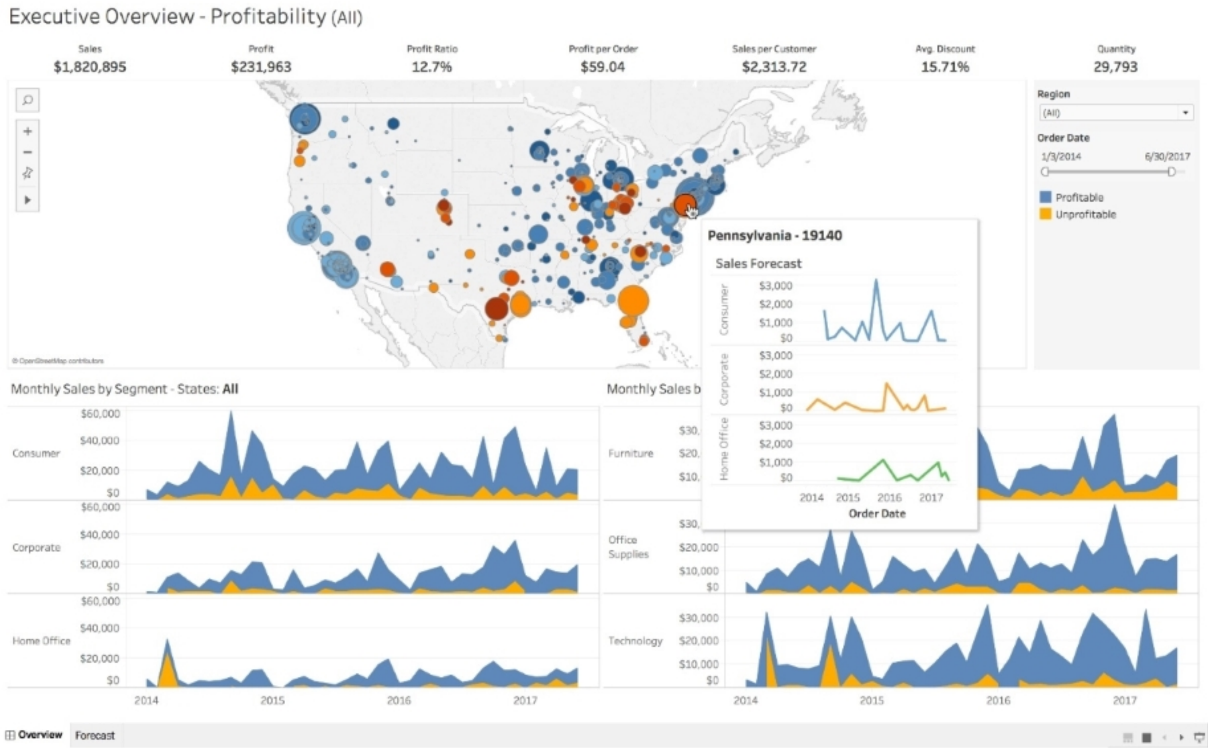


Figure 3.47: Professional BI - Tableau 1

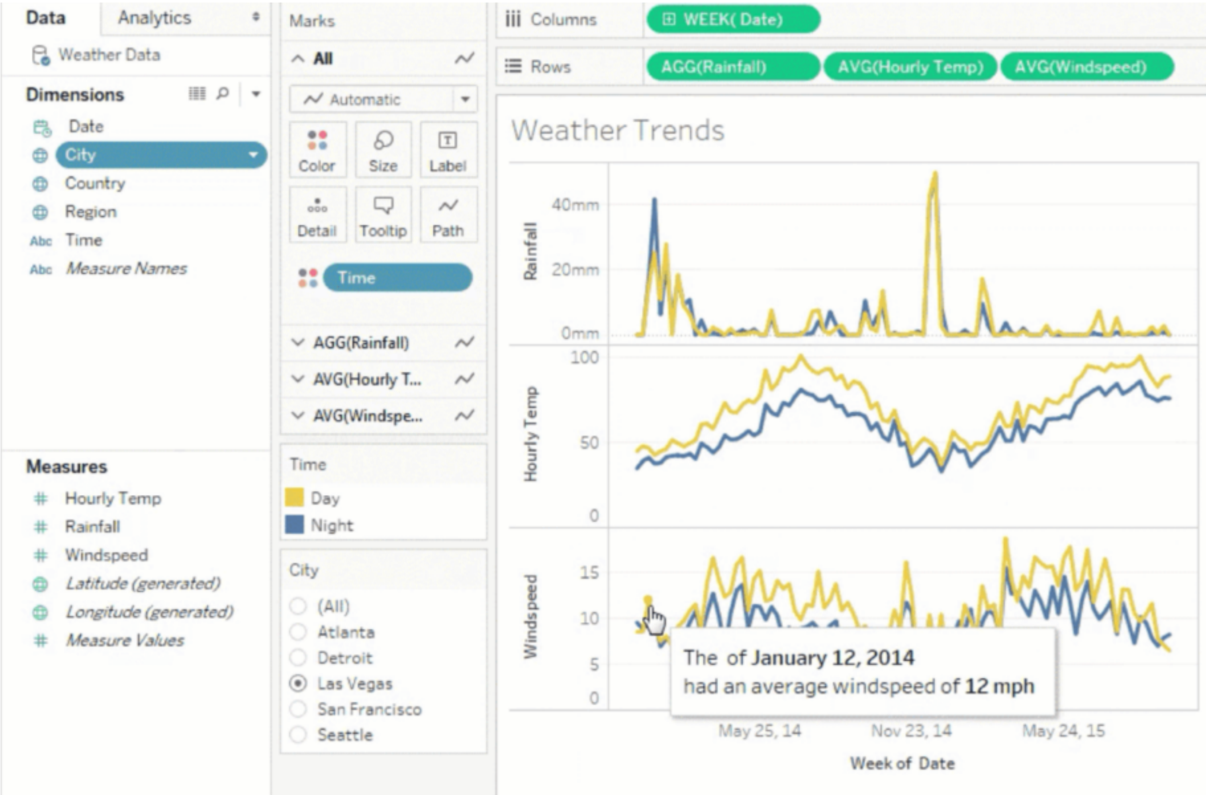


Figure 3.48: Professional BI - Tableau 2

Figure 3.49: 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.

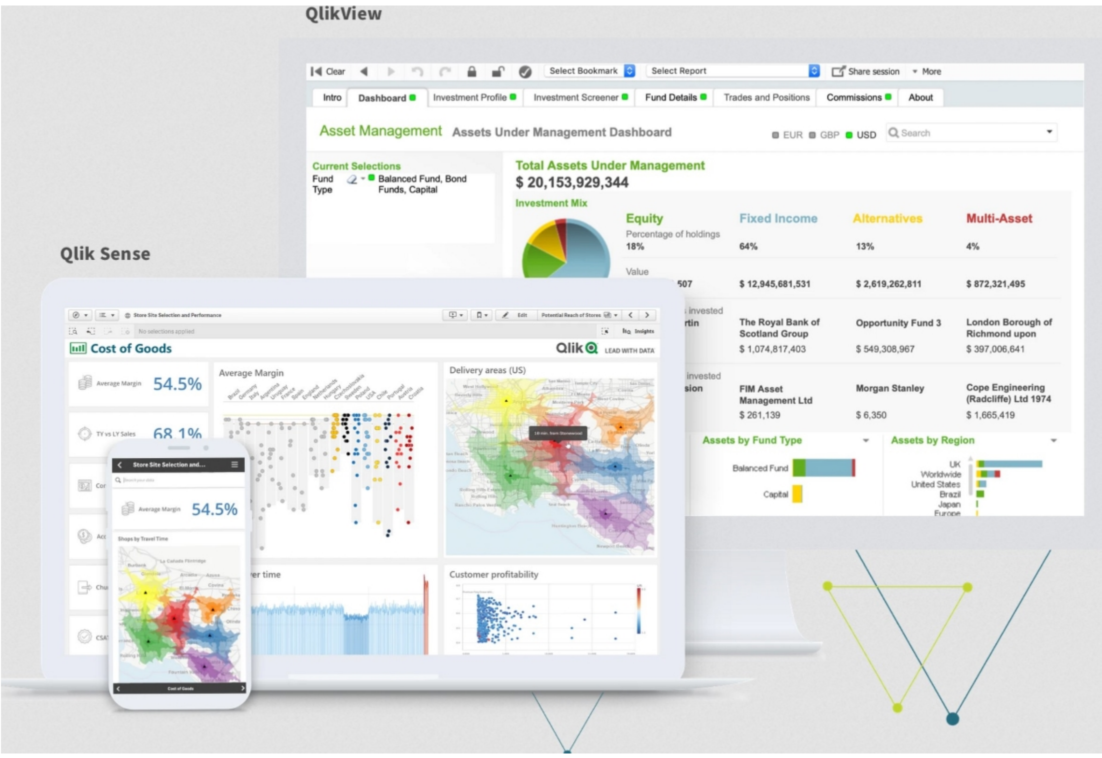


Figure 3.50: Professional BI - QLink

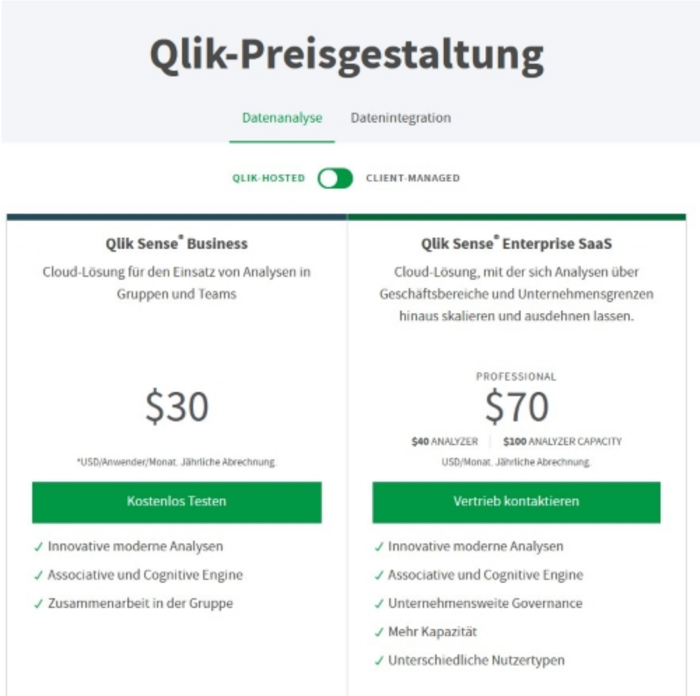


Figure 3.51: Professional BI - QLink Prices

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

4.1 Movies Database Example

A good starting point for finding useful 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 “NaN”-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:

4.1.1 Import the Data

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

[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

```
movies = load_data()
movies.head()
```

Rank	Title	Year	Score	Metascore	Genre	Vote	Director	Runtime	Revenue	Description
1	The Shawshank Redemption	1994	9.3	80.0	Drama	2011509	Frank Darabont	142	28.34	Two imprisoned men bond over a number of years…
2	The Dark Knight	2008	9.0	84.0	Action, Crime, Drama	198020[illegible]	Christopher Nolan	152	534.86	When the menace known as the Joker emerges fro…
3	Inception	2010	8.8	74.0	Action, Adventure, Sci-Fi	176020[illegible]	Christopher Nolan	148	292.58	A thief who steals corporate secrets through t…
4	Fight Club	1999	8.8	66.0	Drama	1609459	David Fincher	139	37.03	An insomniac office worker and a devil-may-car…
5	Pulp Fiction	1994	8.9	94.0	Crime, Drama	1570194	Quentin Tarantino	154	107.93	The lives of two mob hitmen, a boxer, a gangst..

The datatypes of the columns are:

```
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 entires:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 11 columns):
 #   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)
memory usage: 859.5+ KB
```

4.1.2 Separate "NaN"-Values

In column "Revenue" there are 7473 "non-null" values, and 2527 "null" values. I separated the rows with "NaN"-values in column "Revenue". These are the 2527 datarows, where column "Revenue" is null (see fig. 4.1):

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 - NaN Values

```
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
```

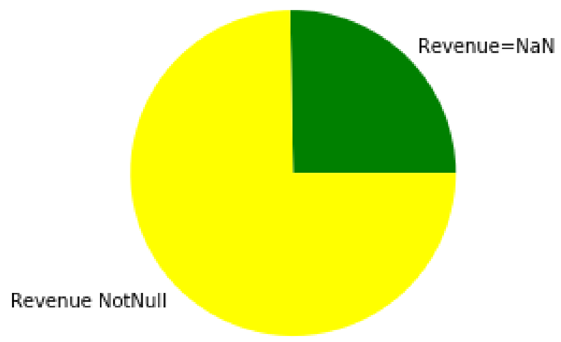


Figure 4.2: Movies Database - Plot Null and NotNull

4.1.3 Visualization of the Data

A first approach should always be create some visualization of the data in order to better understand them.

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

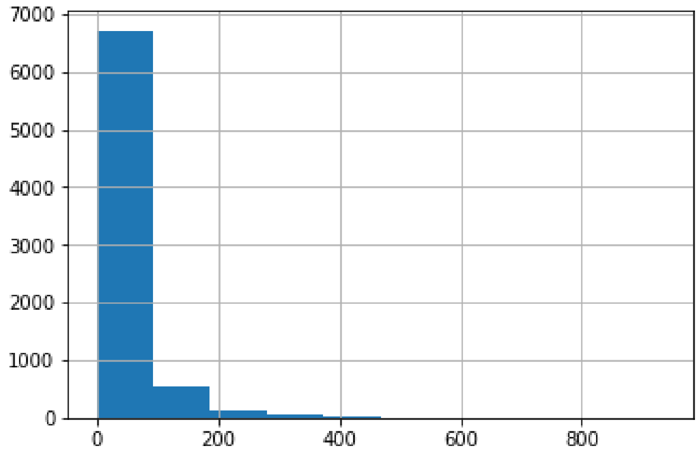


Figure 4.3: Movies Database - Histogram

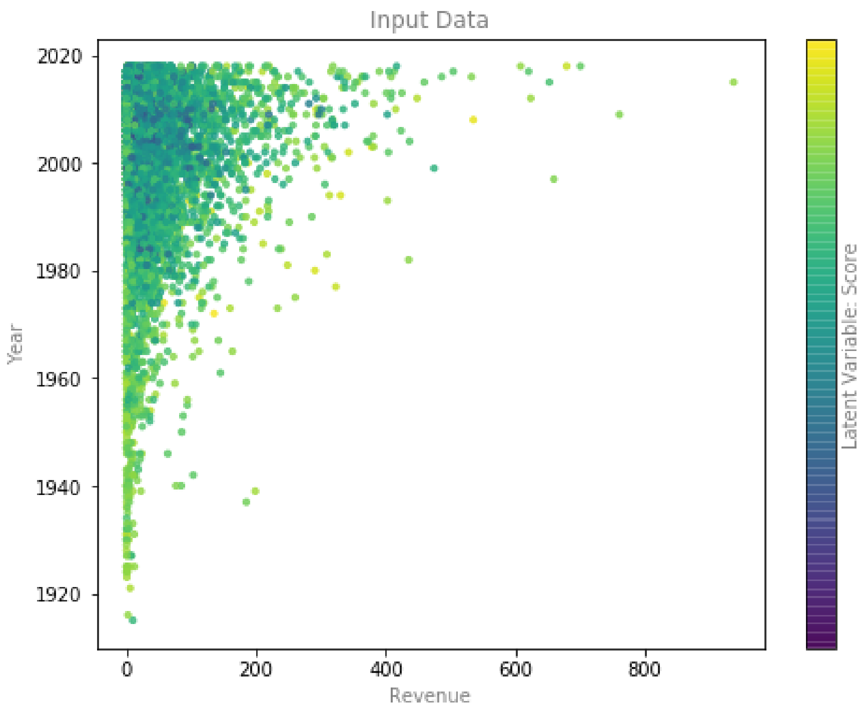


Figure 4.4: Movies Database - Scatterplot Revenue Year

```
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.

```
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')
```

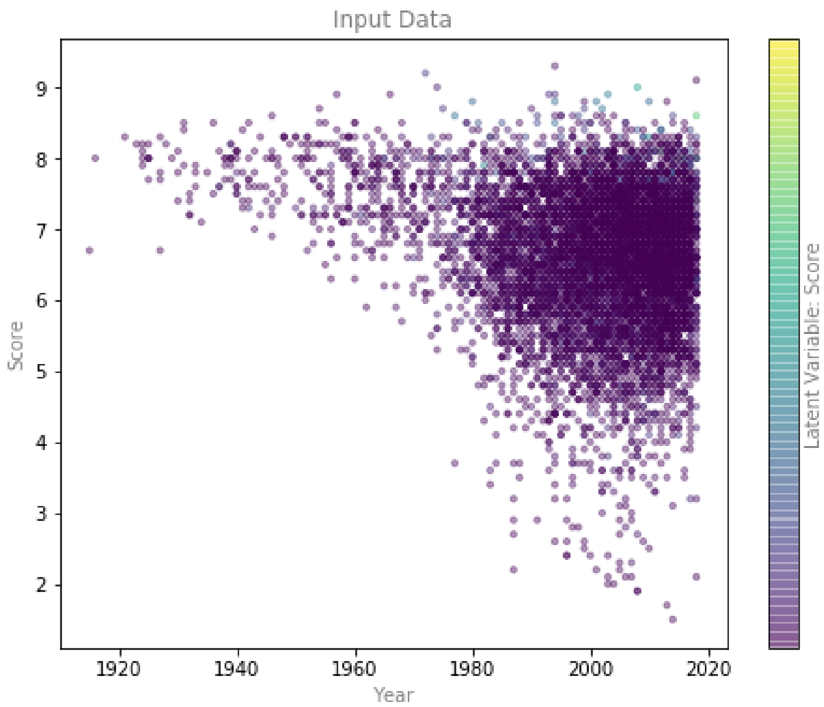


Figure 4.5: Movies Database - Scatterplot Year Score

4.1.4 Draw a Stratified Sample

I drew a stratified sample (based on "Revenue") on this remaining dataset and I received a training dataset and testing 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):

```
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:

```
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]
```

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”=“NaN”

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
```

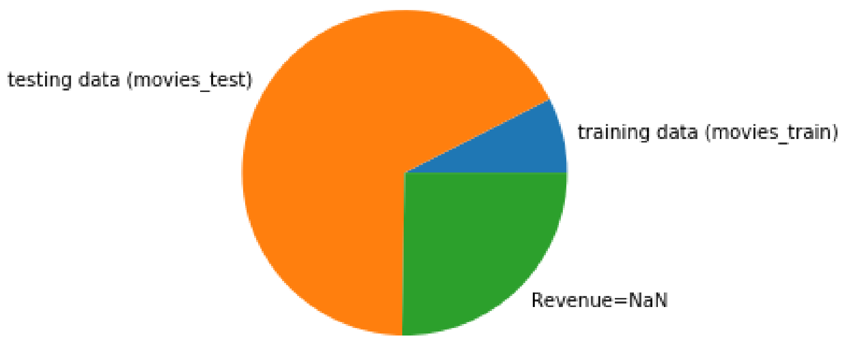


Figure 4.6: Movies Database - Stratified Sample (Train, Test, NaNs)

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 "NaN"-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 "NaN"-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	NaN	Action, Sci-Fi	18517	Don Taylor	103	A modern aircraft carrier is thrown back in ti…
59145915	The Chase	1994	5.8	NaN	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 "nan" values in the new prepared datast "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 "nan"-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 Pipline 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 fittet 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)
```

4.1.8 Cross-Validation

I verified with a cross-validation, how good this model/parameters are

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

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	Description
3123	3124 Victor Frankenstein	2015	6.0	36.0	Drama, Horror, Sci-Fi	45259	Paul McGuigan	110	Told from Igor's perspective, we see the troub…
3061	3062 Battleship Potemkin	1925	8.0	NaN	Drama, History	46636	Sergei M. Eisenstein	75	In the midst of the Russian Revolution of 1905…

As we didn't apply the pipeline on the testing dataset (we only did on the training dataset "movies_train"), there might still some "nan" 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),
 (159.6, 11.99),
 (33.63, 2.19),
 (44.9, 26.83),
 (38.52, 22.52),
 (33.04, 2.19),
 (3.2, 11.99),
 (37.49, 16.38),
 (17.88, 64.19),
 (41.19, 191.45),
 (16.19, 12.19),
 (0.05, 3.02),
 (16.68, 64.19),
 (35.11, 11.54),
```

```
(36.0, 40.22),
(3.61, 19.64),
(0.59, 0.05),
(0.99, 5.48)]
```

The mean-squared-error is as follows:

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

```
51.3429319867886
```

Taking now 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`. We want to calculate also the mean and standard deviation: `movies_test_labels.mean()` which results in `36.20`and `movies_test_labels.std()` which is `60.60`.

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)
```

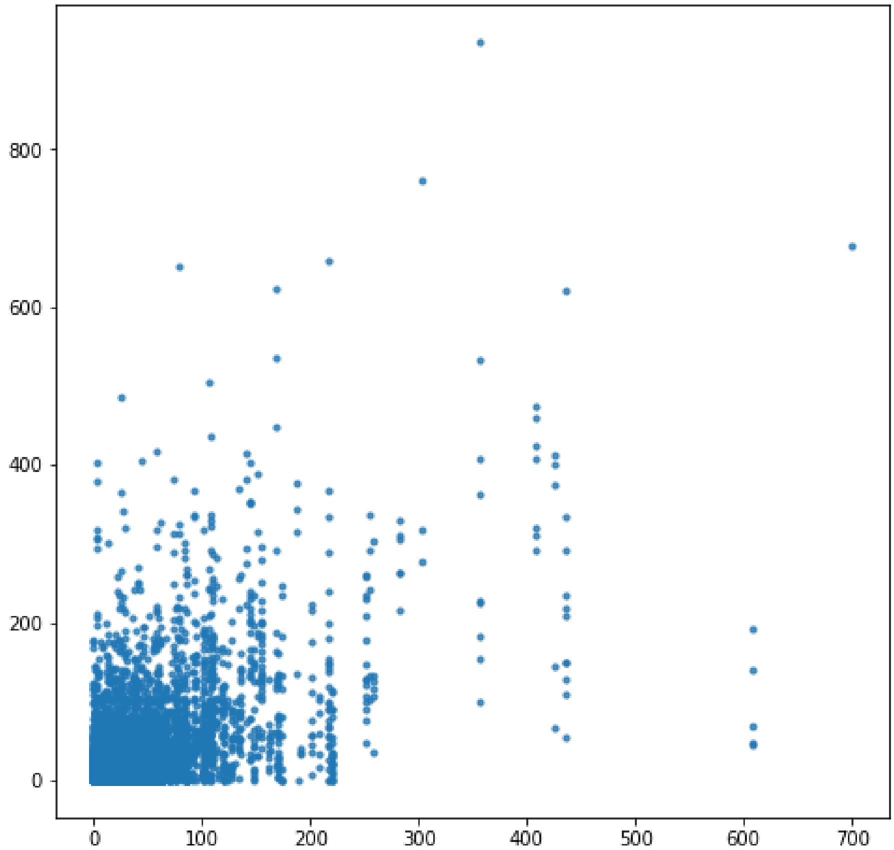


Figure 4.7: Movies Database - Plot True vs Predicted Value Side-by-Side Testdataset

Looks confusing. I would have expected something a bit more similar to the following plot. 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)
```

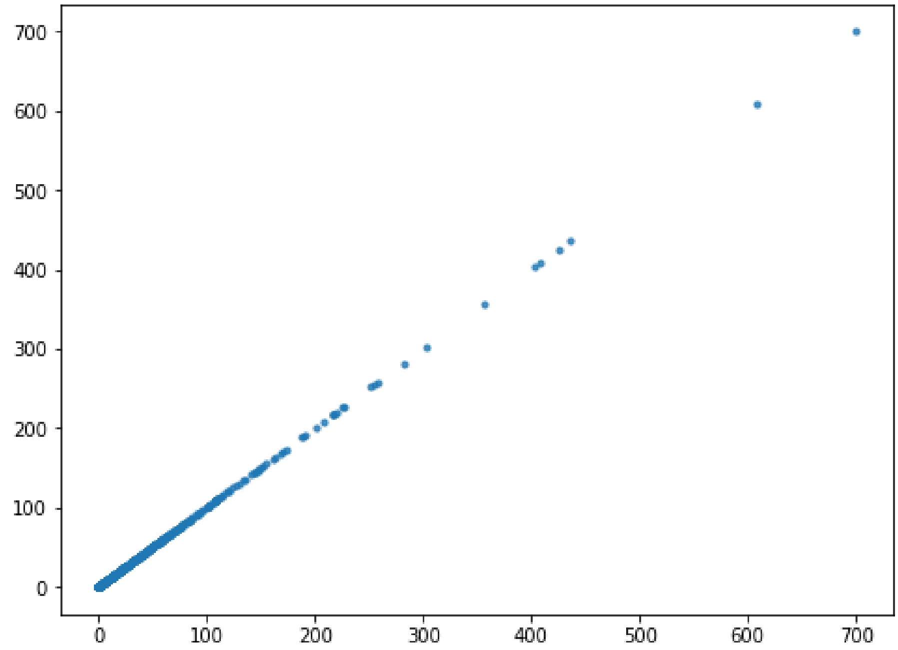


Figure 4.8: Movies Database - Plot True vs Predicted Value Side-by-Side Trainingdataset

Now I calculate the “Revenue” where it has “NaN”-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", ... and the other categorical like "genre" don't seem to have much more added value in my opinion). 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 mentionned right in the beginning of this Jupyter-Notebook the "OneHotEncoder" could be used. 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 Jupyter-Notebook.

On my GitHub profile you can find my Jupyter-Notebook for this example[4]

4.2 "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*[5] (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 (see fig. 4.9). The same structure will be applied on the third example, the "Big Data" example (see sec. 4.3).

So let's start with the "Scikit-Learn": the Mind-Map fig. 4.9 shows you what we need to do, starting from *"1. import and create index"*, then *discovering the data*, *preparing and cleaning the data*, *creating pipelines* and so on to *"7. select and train model"*. After that we will do the *cross-validation*, *save the*

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

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

model until we reach *"11. evaluate final results"*. You will find the same structure in the `.py` file and it should be a guide to work out your own problems with the same structure.

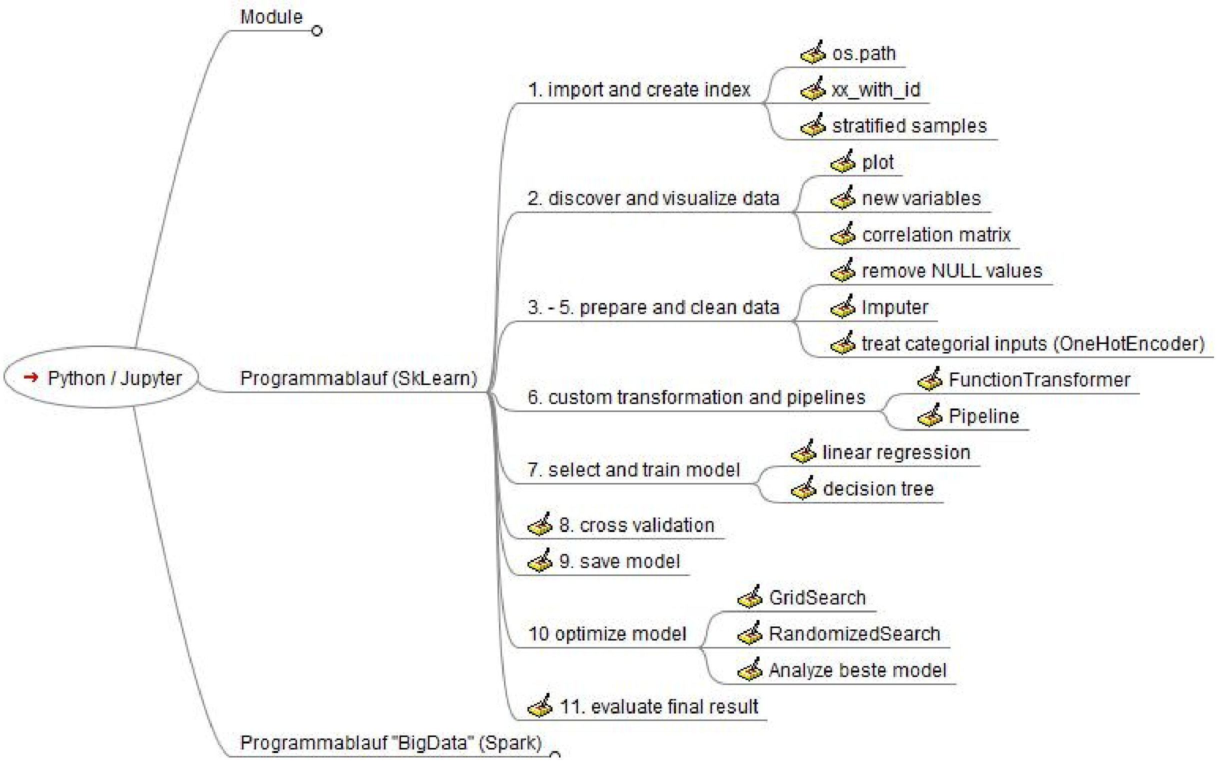


Figure 4.9: Mind Map - Scikit-Learn "Small Data"

```
#%% ######################################################################
# 1. create index
    # 1.1 Alternative 1: generate id with static data
    # 1.2 Alternative 2: generate stratified sampling
    # 1.3 verify if stratified example is good
# 2. Discover and visualize the data to gain insights
# 3. prepare for Machine Learning
    # 3.1 find all NULL-values
    # 3.2 remove all NULL-values
# 4. Use "Imputer" to clean NaNs
# 5. treat "categorial" inputs
# 6. custom transformer and pipelines
    # 6.1 custom transformer
    # 6.2 pipelines
# 7. select and train model
    # 7.1 LinearRegression model
    # 7.2 DecisionTreeRegressor model
# 8. crossvalidation
```

```
    # 8.1 for DecisionTreeRegressor
    # 8.2 for LinearRegression
    # 8.3 for RandomForestRegressor
    # 8.4 for ExtraTreesRegressor
# 9. Save Model
# 10. Optimize Model
    # 10.1 GridSearchCV
        # 10.1.1 GridSearchCV on RandomForestRegressor
        # 10.1.2 GridSearchCV on LinearRegressor
    # 10.2 Randomized Search
    # 10.3 Analyze best models
# 11. Evaluate final model on test dataset
#%% ########################################################################
```

I aligned this “Small Data” structure to the Apache Spark “Big Data” structure (see Mind Map fig. 4.14) in order to learn from each of this two approaches. Finally I will put these two Mind Maps into one big (see fig. 4.20) which you can take as a guide to navigate through all of your machine-learning problems.

Common Imports

These are the common imports which we need. For the moment it is not necessary to understand all of this. If you want to know more about what these imports do (e.g. `sklearn.model_selection`), then use my references and read to the official documentation (in case of `sklearn` you have to go to the Scikit-Learn website). Getting familiar with the official documentation and learning how to quickly find, what you need for your specific problem is always a good idea. Most of these official documentation are well structured and once you understood, how to navigate through them you won’t need Google or Stackoverflow[6] (which is a question and answer forum for programmers) to solve your problems.

```
#To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals

# Common imports
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import os
import tarfile
from six.moves import urllib
```

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

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
import hashlib
from sklearn.model_selection import StratifiedShuffleSplit
from pandas.plotting import scatter_matrix
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor
import joblib
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
from scipy import stats
```

Read the csv-file

It is always a good idea to create variables, which can be reused later. This makes the code easier to read. Here the path to the CSV file has been set to the variable HOUSING_PATH.

```
HOUSING_PATH = os.path.join("datasets", "housing")
def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
housing = load_housing_data()
```

4.2.1 Create Index (1)

4.2.1.1 Alternative 1: Generate ID with Static Data (1.1)

```
housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
```

4.2.1.2 Alternative 2: Generate Stratified Sampling (1.2)

```
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split.split(housing, housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
strat_test_set["income_cat"].value_counts() / len(strat_test_set)
housing["income_cat"].value_counts() / len(housing)
```

4.2.1.3 Verify if Stratified Example is good (1.3)

```
def income_cat_proportions(data):
    return data["income_cat"].value_counts() / len(data)
train_set, test_set = train_test_split(housing, test_size=0.2,
 ↪  random_state=42)
compare_props = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_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)
```

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

Create some visualizations as I described in sec. 3. For example:

```
housing.plot(kind="scatter",
             x="longitude",
             y="latitude",
             alpha=0.4,
             s=housing["population"]/100,
             label="population",
             figsize=(10,7),
             c="median_house_value",
             cmap=plt.get_cmap("jet"),
             colorbar=True,
```

```
             sharex=False,
             title="housing_prices_scatterplot")
plt.legend()
save_fig("housing_prices_scatterplot")
```

4.2.3 Prepare for Machine Learning (3)

```
housing = strat_train_set.drop("median_house_value",
                               axis=1) # drop labels for training set
housing_labels = strat_train_set["median_house_value"].copy()
```

4.2.3.1 Find all NULL-Values (3.1)

```
print(housing.info())
print("Are there nans in column total_bedrooms?\n",
      housing["total_bedrooms"].isnull().any())
print("Show rows with nan:\n",
      housing[housing["total_bedrooms"].isnull()])
```

4.2.3.2 Remove all NULL-Values (3.2)

There are different options for handling NULL values, which occur in a column: we can delete the entire row or we can delete the entire column or we 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 = housing[housing.isnull().any(axis=1)].head()
# sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
# sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
median = housing["total_bedrooms"].median()
sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) #
↪  option 3 # In[56]:
print("sample_incomplete_rows\n",
      sample_incomplete_rows)
```

4.2.4 Use "Imputer" to Clean NaNs (4)

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

```
imputer = SimpleImputer(strategy="median")
housing_num = housing.select_dtypes(include=[np.number]) #or: housing_num =
↪  housing.drop('ocean_proximity', axis=1)
imputer.fit(housing_num)
print("imputer.strategy\n",
      imputer.strategy)
print("imputer.statistics_\n",
      imputer.statistics_)
print("housing_num.median\n",
      housing_num.median().values)  # Check that this is the same as
↪  manually computing the median of each attribute:
print("housing_num.mean\n",
      housing_num.mean().values)  # Check that this is the same as manually
↪  computing the median of each attribute.
```

Transform the training set:

```
X = imputer.transform(housing_num) # :
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
↪  index=housing.index)
housing_tr.loc[sample_incomplete_rows.index.values]
```

4.2.5 Treat "Categorial" Inputs (5)

Use the OneHotEncoder for the categorial values:

```
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

4.2.6 Custom Transformer and Pipelines (6)

4.2.6.1 Custom Transformer (6.1)

```
def add_extra_features(X, add_bedrooms_per_room=True):
    rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
```

```
    population_per_household = X[:, population_ix] / X[:, household_ix]
    if add_bedrooms_per_room:
        bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
        return np.c_[X, rooms_per_household,
                     population_per_household,
                     bedrooms_per_room]
    else:
        return np.c_[X, rooms_per_household,
                     population_per_household]

attr_adder = FunctionTransformer(add_extra_features,
                                 validate=False,
                                 kw_args={"add_bedrooms_per_room": False})
housing_extra_attribs = attr_adder.fit_transform(housing.values)

housing_extra_attribs = pd.DataFrame(
    housing_extra_attribs,
    columns=list(housing.columns)+["rooms_per_household",
 ↪  "population_per_household"],
    index=housing.index)
print("housing_extra_attribs.head()\n", housing_extra_attribs.head())
```

4.2.6.2 Pipelines (6.2)

```
num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', FunctionTransformer(add_extra_features,
                                              validate=False)),
        ('std_scaler', StandardScaler()),
    ])
housing_num_tr = num_pipeline.fit_transform(housing_num)
print("housing_num_tr\n", housing_num_tr)

try:
    from sklearn.compose import ColumnTransformer
except ImportError:
    from future_encoders import ColumnTransformer # Scikit-Learn < 0.20

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
```

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

4.2.7 Select and Train Model (7)

4.2.7.1 LinearRegression Model (7.1)

```
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
# let's try the full preprocessing pipeline on a few training instances

some_data = housing.iloc[:1]
some_labels = housing_labels.iloc[:1]
some_data_prepared = full_pipeline.transform(some_data)
print("Predictions:\n", lin_reg.predict(some_data_prepared))
print("Labels:\n", list(some_labels)) # Compare against the actual values:

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
print("lin_rmse\n", lin_rmse)
```

4.2.7.2 DecisionTreeRegressor Model (7.2)

```
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(housing_prepared, housing_labels)
housing_predictions = tree_reg.predict(housing_prepared)
tree_mse = mean_squared_error(housing_labels, housing_predictions)
tree_rmse = np.sqrt(tree_mse)
print("tree_rmse\n", tree_rmse)
```

4.2.8 Cross-Validation (8)

4.2.8.1 For DecisionTreeRegressor (8.1)

```
scores = cross_val_score(tree_reg,
                         housing_prepared,
                         housing_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())
```

4.2.8.2 For LinearRegression (8.2)

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

4.2.8.3 For RandomForestRegressor (8.3)

```
forest_reg = RandomForestRegressor(n_estimators=10,
                                   random_state=42)
forest_reg.fit(housing_prepared,
               housing_labels)

housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels,
                                housing_predictions)
forest_rmse = np.sqrt(forest_mse)
print(forest_rmse)
```

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

4.2.8.4 For ExtraTreesRegressor (8.4)

```
extratree_reg = ExtraTreesRegressor(n_estimators=10,
                                    random_state=42)
extratree_reg.fit(housing_prepared,
                  housing_labels)
housing_predictions = extratree_reg.predict(housing_prepared)
extratree_mse = mean_squared_error(housing_labels,
                                   housing_predictions)
extratree_rmse = np.sqrt(extratree_mse)
print(extratree_rmse)
extratree_scores = cross_val_score(extratree_reg,
                                   housing_prepared,
                                   housing_labels,
                                   scoring = "neg_mean_squared_error",
                                   cv=10)
extratree_rmse_scores = np.sqrt(-extratree_scores)
display_scores(extratree_rmse_scores)
```

4.2.9 Save Model (9)

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

4.2.10 Optimize Model (10)

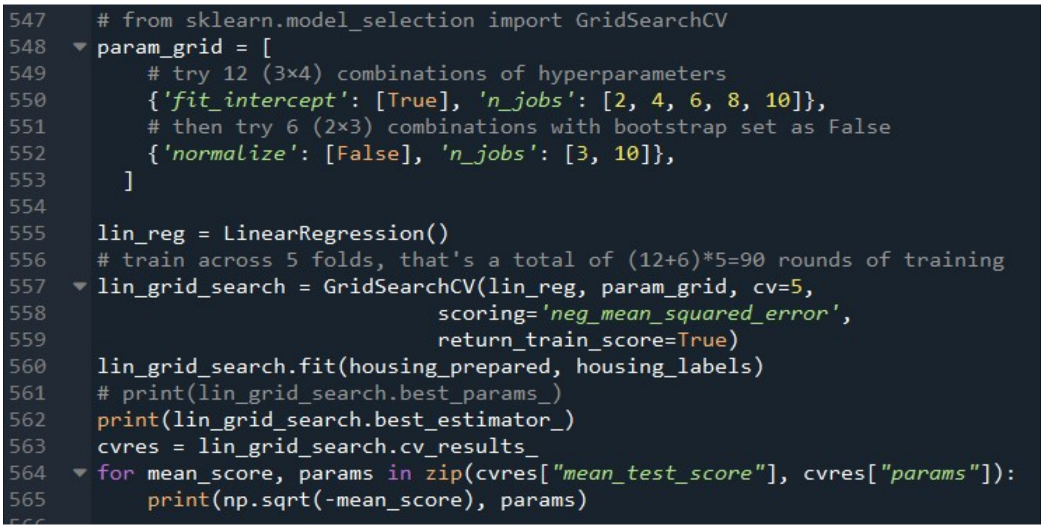
4.2.10.1 GridSearchCV (10.1)

```
param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'n_estimators': [30, 40, 50],
     'max_features': [2, 4, 6, 8, 10]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'bootstrap': [False],
     'n_estimators': [3, 10],
     'max_features': [2, 3, 4]},
  ]
forest_reg = RandomForestRegressor(random_state=42)
# train across 5 folds, that's a total of (12+6)*5=90 rounds of training
grid_search = GridSearchCV(forest_reg,
                           param_grid,
                           cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(housing_prepared, housing_labels)
print(grid_search.best_params_)
print(grid_search.best_estimator_)
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"],
                              cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

4.2.10.1.1 GridSearchCV on RandomForestRegressor (10.1.1)

```
param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'fit_intercept': [True],
     'n_jobs': [2, 4, 6, 8, 10]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'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(housing_prepared,
                    housing_labels)

# print(lin_grid_search.best_params_)

print(lin_grid_search.best_estimator_)
cvres = lin_grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"],
                              cvres["params"]):
    print(np.sqrt(-mean_score), params)
```



```
# from sklearn.model_selection import GridSearchCV
param_grid = [
    # try 12 (3×4) combinations of hyperparameters
    {'fit_intercept': [True], 'n_jobs': [2, 4, 6, 8, 10]},
    # then try 6 (2×3) combinations with bootstrap set as False
    {'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(housing_prepared, housing_labels)
# print(lin_grid_search.best_params_)
print(lin_grid_search.best_estimator_)
cvres = lin_grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

Figure 4.10: Small Data - Optimize Model GridSearchCV on LinearRegressor (10.1.2)

4.2.10.1.2 GridSearchCV on LinearRegressor (10.1.2)

4.2.10.2 Randomized Search (10.2)

```
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(housing_prepared,
               housing_labels)
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"],
                              cvres["params"]):
    print(np.sqrt(-mean_score), params)
```

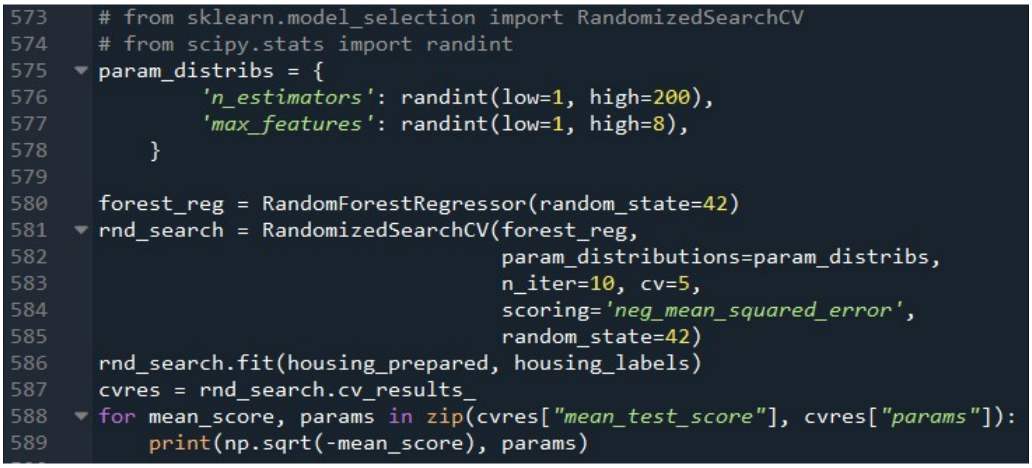


Figure 4.11: Small Data - Optimize Model Randomized Search (10.2)

4.2.10.3 Analyze best models (10.3)

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
#cat_encoder = cat_pipeline.named_steps["cat_encoder"] # old solution
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
sorted(zip(feature_importances, attributes), reverse=True)
```

```
feature_importances = grid_search.best_estimator_.feature_importances_
feature_importances
extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
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
sorted(zip(feature_importances, attributes), reverse=True)
```

Figure 4.12: Small Data - Optimize Model Analyze best models (10.3)

4.2.11 Evaluate final model on test dataset (11)

```
final_model = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].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 = grid_search.best_estimator_

X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].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)))
      )
```

Figure 4.13: Small Data - Evaluate final model on test dataset (11)

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

This will be an example for a "Big-Data"[7] environment and uses the *Apache Spark MLib*[8] 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 [9]. A few more helpful sources are the following GitHub repositories:

- tirthajyoti/Spark-with-Python[10] (MIT Licence)
- Apress/learn-pyspark[11] (Freeware License)
- mahmoudparsian/pyspark-tutorial[12] (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"*. I must say: Understanding the concept of "Big-Data" and how to differentiate "standard" machine learning from a "scalable" environment is not easy. I recommend

[7] Big-Data, Wikipedia, https://en.wikipedia.org/wiki/Big_Data
[8] Apache Spark MLib, https://spark.apache.org/mllib/
[9] Official Apache Spark ML documentation, https://spark.apache.org/docs/latest/ml-guide.html
[10] GitHub repository "tirthajyoti/Spark-with-Python", https://github.com/tirthajyoti/Spark-with-Python (MIT Licence)
[11] GitHub repository "Apress/learn-pyspark", https://github.com/Apress/learn-pyspark (Freeware License)
[12] GitHub repository "mahmoudparsian/pyspark-tutorial", https://github.com/mahmoudparsian/pyspark-tutorial (Apache License v2.0)

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.

I added a **"Digression (Excurs)"** at the end of this chapter which covers the topics *"Big Data Visualization"*, *"K-Means-Clustering in Spark"* and *"Map-Reduce"* [13] (one of the powerful programming models for Big Data).

Let's start with the "Big Data" structure, which I put into a mind map (you can download it from my GitHub repository). I aligned the structure to the Scikit-Learn mind map (see fig. 4.9) in order to learn from each of this two approaches.

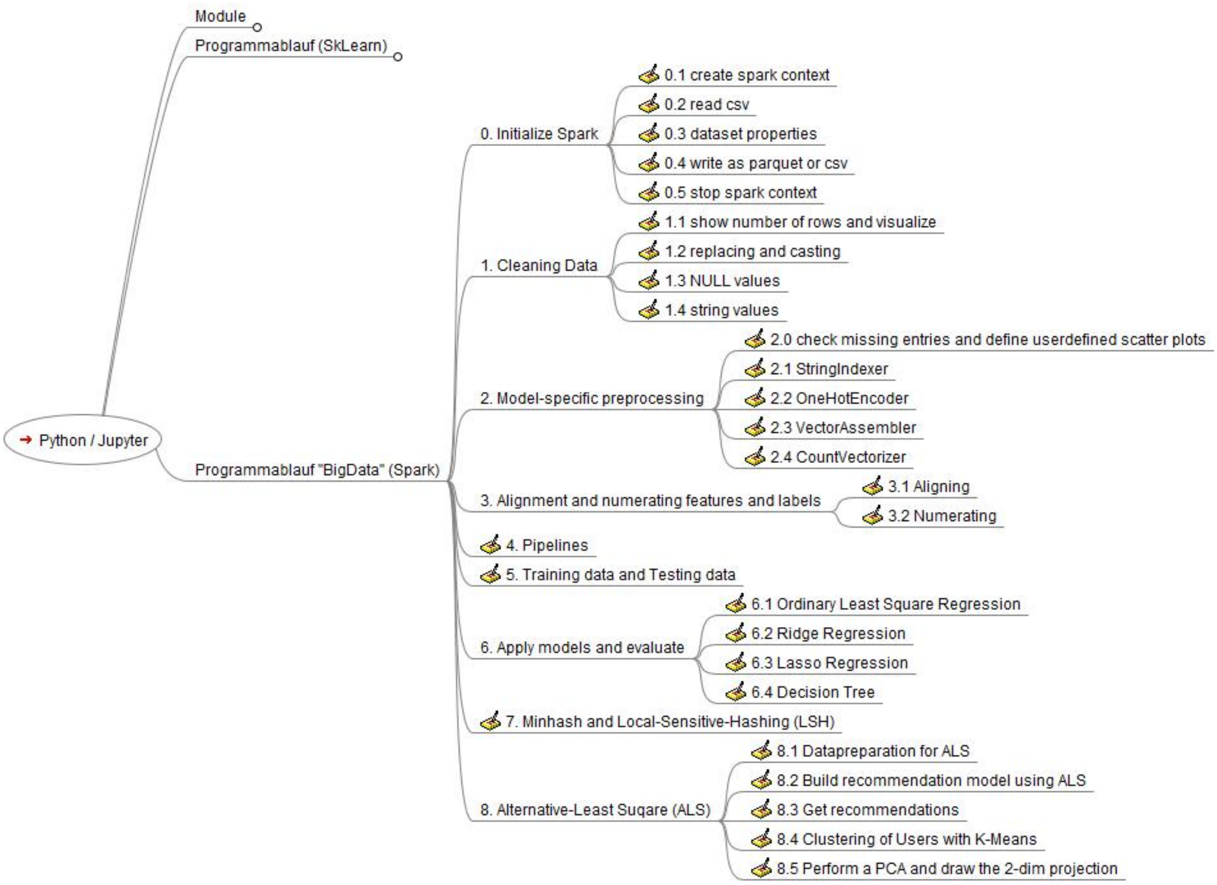


Figure 4.14: Mind Map - Apache Spark ML "Big Data"

4.3.1 Get Ready

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).

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

- 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[14], 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[15], 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[16] what docker container and docker images are and how it works. Despite the difficult subject matter it is a nice tutorial.

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[17]. 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
```

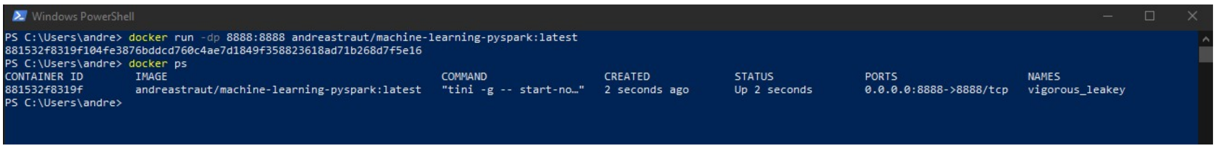


Figure 4.15: Big Data - Run Docker

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

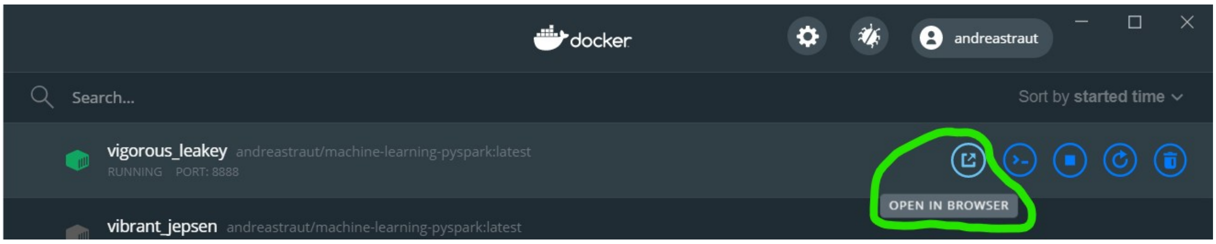


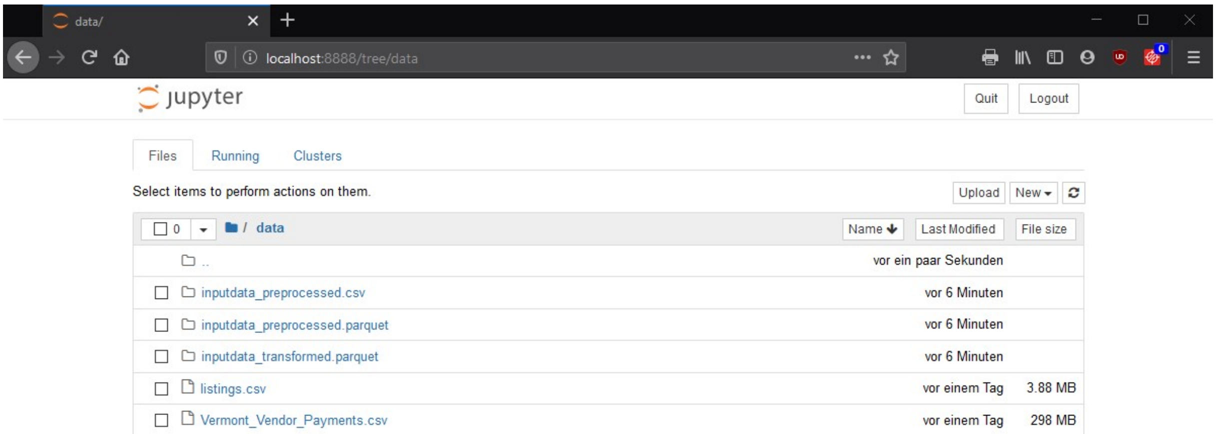
Figure 4.16: Big Data - Docker Dashbord

[14] My experiences with Microsoft Azure, https://github.com/AndreasTraut/Experiences-with-MicrosoftAzure
[15] Docker, https://www.docker.com/
[16] Docker-Curriculum, https://docker-curriculum.com/
[17] My docker machine-learning repository, https://hub.docker.com/repository/docker/andreastraut/machine-learning-pyspark

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

Figure 4.17: 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[18]. It is available below under consideration of the *Creative Commons CC0 1.0 Universal (CC0 1.0) Public Domain Dedication Licence*[19]. The data for the Vermont-Vendor-Payments[20] is available under condieration of the Open Data Commons Open Database License[21].



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

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

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

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

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

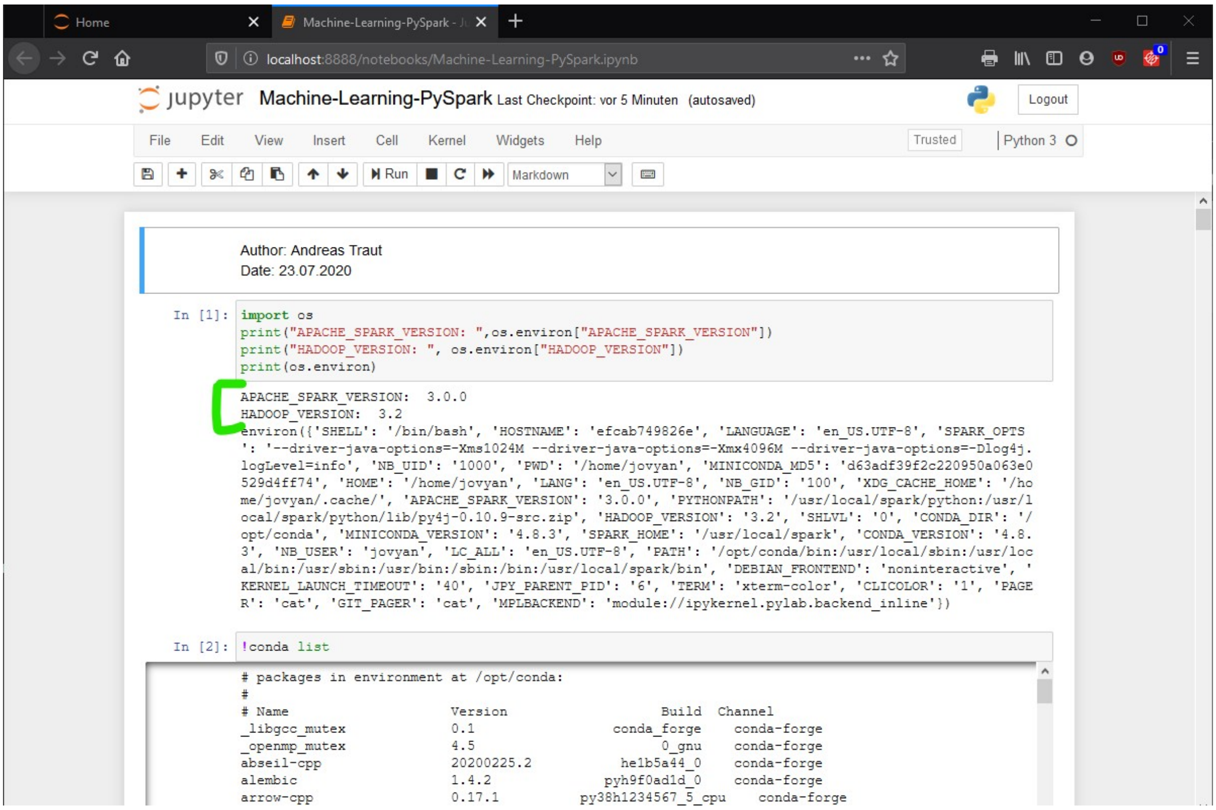


Figure 4.18: Big Data - Docker Jupyter Notebook

4.3.2 Initialize Spark (0)

I recommend to read the Apache Spark "Get Started Guide"[22]. Initializing a Spark sessions and read a CSV file (see also the official Spark documentation, https://spark.apache.org/docs/latest/sql-getting-started.html#starting-point-sparksession)

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

4.3.2.1 Create Spark Context and Spark Session (0.1)

```
sc = pyspark.SparkContext(appName='Spark Modelling Context')

spark = SparkSession.builder \
```

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

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

4.3.2.2 Read CSV (0.2)

```
import os
datapath = os.environ['PWD']
filename = datapath + "/data/listings.csv"
#read in data from 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]
```

4.3.2.3 Dataset Properties and some Select, Group and Aggregate Methods (0.3)

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
```

4.3.2.4 Write as Parquet or CSV (0.4)

If you want to persist (=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)
```

Reading works as follows:

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

4.3.2.5 Stop a Spark Session and Spark Context (0.5)

```
spark.stop()
sc.stop()
```

4.3.3 Cleaning the data (1)

The data-cleaning and data preparation (eliminating of null values, visualization techniques) the structure is pretty similar to the “Small data” (Sklearn) approach.

4.3.3.1 Show Number of Rows and Columns and Do Some Visualizations (1.1)

```
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
```

```
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();
```

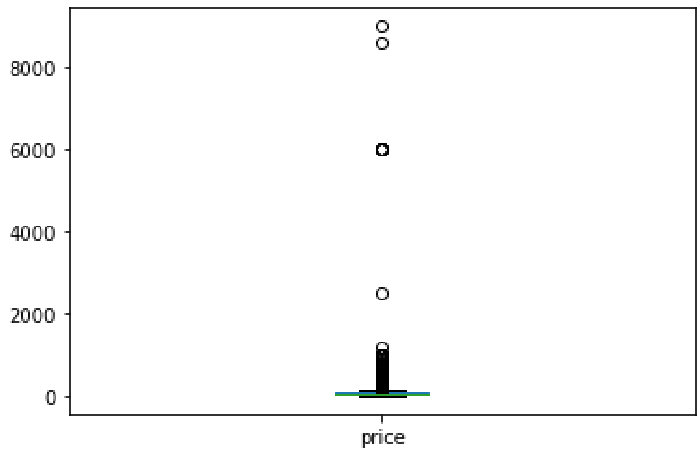


Figure 4.19: Big Data - Price Box Plot

4.3.3.2 Replacing and Casting (1.2)

```
data = data.withColumn('price', F.regexp_replace('price','\$',''))
data = data.withColumn('price', F.regexp_replace('price',',',''))
data = data.withColumn('price', data['price'].cast('double'))
```

4.3.3.3 Null-Values (1.3)

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

```
104 missing values for price
```

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

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

```
0 missing values for price
```

4.3.3.4 String Values (1.4)

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

4.3.4 Model-specific preprocessing (2)

4.3.4.1 Check Missing Entries and Define Userdefined Scatter Plot (2.0)

```
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()
```

4.3.4.2 StringIndexer (2.1)

Please read the official documentation in order to understand, what the "StringIndexer"[23] does.

```
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('roo
↪   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
```

4.3.4.3 OneHotEncoder (2.2)

Please read the official documentation in order to understand, what the "OneHotEncoder"[24] does.

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

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

```
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)
```

4.3.4.4 VectorAssembler (2.3)

Please read the official documentation in order to understand, what the "VectorAssembler"[25] does.

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

```
+-----------------+
|number_of_reviews|
+-----------------+
|            145.0|
|             27.0|
|            133.0|
|            292.0|
|              8.0|
|             24.0|
|             48.0|
|            262.0|
|             86.0|
|             60.0|
|             86.0|
|            307.0|
|            130.0|
|             21.0|
|              5.0|
```

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

```
|             188.0|
|              31.0|
|              74.0|
|             296.0|
|              39.0|
+------------------+
```

4.3.4.5 CountVectorizer (2.4)

Please read the official documentation in order to understand, what the "CountVectorizer"[26] does.

4.3.5 Aligning and numerating Features and Labels (3)

4.3.5.1 Aligning (3.1)

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
```

[26] "CountVectorizer", https://spark.apache.org/docs/latest/ml-features.html#countvectorizer

4.3.5.2 Numerating (3.2)

```
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))
```

4.3.6 Pipelines (4)

```
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])
```

4.3.7 Training data and Testing data (5)

```
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.

4.3.8 Apply models and evaluate (6)

4.3.8.1 Ordinary Least Square Regression (6.1)

After having extracted, transformed and selected features we will apply some models, for example the "OLS Regression"[27]. Please read the official documentation for learning more about OLS.

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

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

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)
```

4.3.8.2 Ridge Regression (6.2) 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”[28] in detail.

4.3.8.3 Decision Tree (6.4)

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”[29] in detail.

4.3.9 Minhash und Local-Sensitive-Hashing (7)

I worked on the Minhashing and on the Local-Sensitive Hasing in a separate example[30].

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

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

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

4.4 Summary Mind-Map

To summarize the whole coding structure have a look at the following and also the provided mind-maps. My mind map below may help you to structure your code:

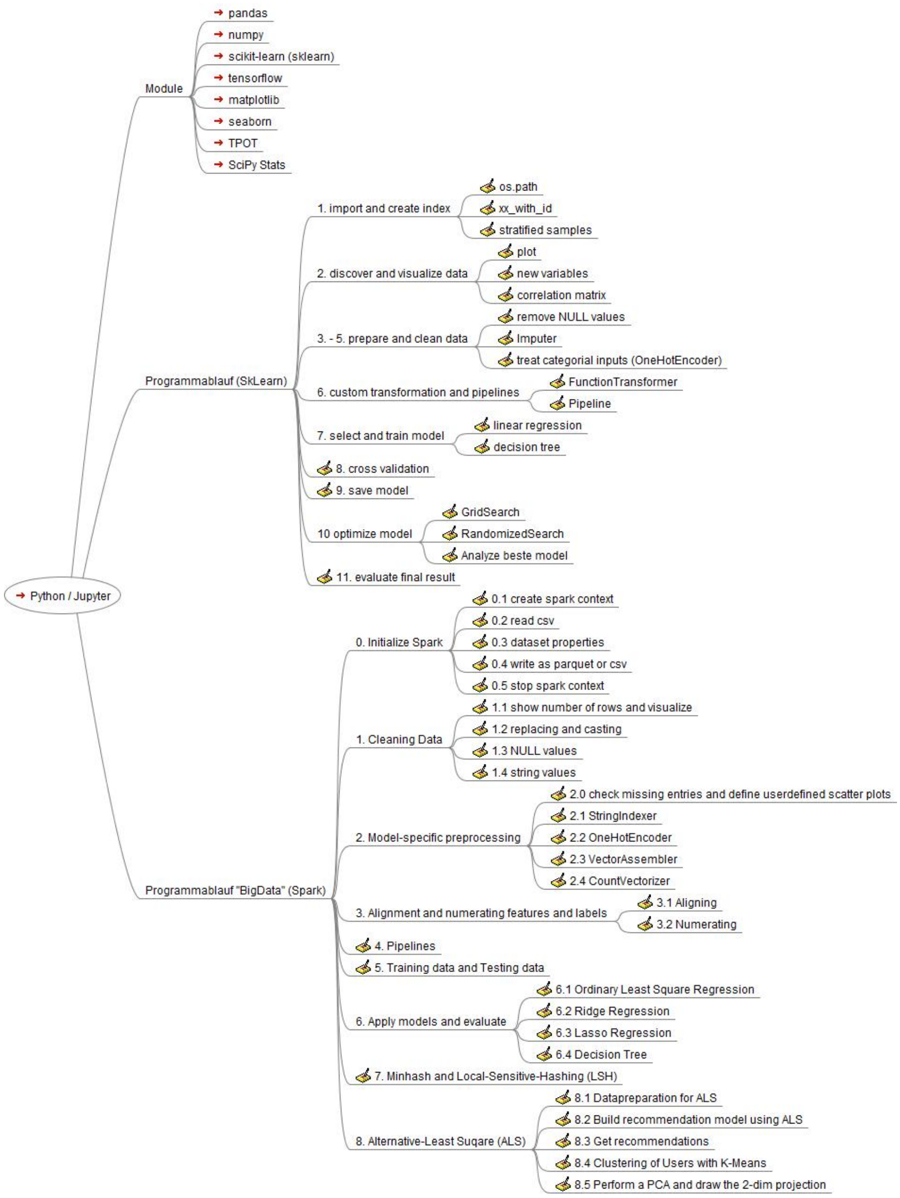


Figure 4.20: Mind Map - Scikit-Learn and Apache SparkML

4.5 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[31], Signal-Messenger[32], Threema-Messenger[33], Corona-Warnapp[34], … to be mentioned. There are many positive examples of sharing code and data "for free".

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*[35]. The data for the Vermont-Vendor-Payments can be downloaded under consideration of the *Open Data Commons Open Database License* [36]. The movies database doesn't even mention a license. There you find a lot of more datasets and also coding examples for your studies.

[31] Firefox, https://github.com/mozilla
[32] Signal-Messenger, https://github.com/signalapp
[33] Threema-Messenger, https://github.com/threema-ch
[34] Corona-Warnapp, https://github.com/corona-warn-app
[35] Creative Commons 1.0 Universal Public Domain Dedication Licence, http://creativecommons.org/publicdomain/zero/1.0/
[36] 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

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. 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:

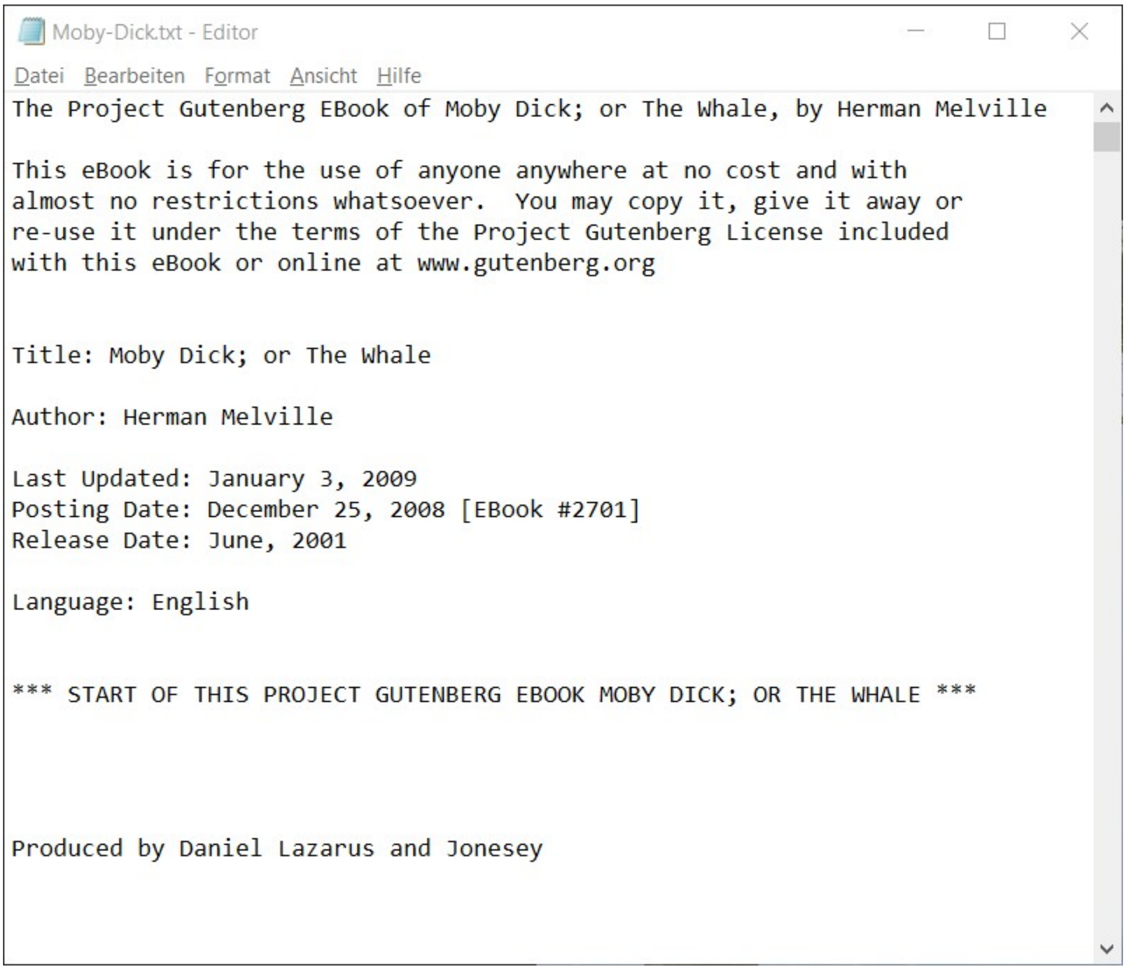
```
from pyspark import SparkContext
sc=SparkContext(master="local[4]")
text_file = sc.textFile("data/Moby-Dick.txt")
type(text_file)
```

```
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



Moby-Dick.txt - Editor
Datei Bearbeiten Format Ansicht Hilfe

```
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


*** START OF THIS PROJECT GUTENBERG EBOOK MOBY DICK; OR THE WHALE ***




Produced by Daniel Lazarus and Jonesey
```

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', '', '',
 ↪ '*** START OF THIS PROJECT GUTENBERG EBOOK MOBY DICK; OR THE WHALE ***',
 ↪ '', '', '', '', 'Produced by Daniel Lazarus and Jonesey', '', '', '',
 ↪ '', '', 'MOBY DICK; OR THE WHALE', '', 'By Herman Melville', '', '', '',
 ↪ '', "Original Transcriber's Notes:", '', 'This text is a combination of
 ↪ etexts, one from the now-defunct ERIS', "project at Virginia Tech and
 ↪ one from Project Gutenberg's archives. The", 'proofreaders of this
 ↪ version are indebted to The University of Adelaide', 'Library for
 ↪ preserving the Virginia Tech version. The resulting etext', 'was
 ↪ compared with a public domai
```

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(100))
```

```
['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', '', '', '***', 'START', 'OF', 'THIS', 'PROJECT',
```

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(100))
```

```
['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',
↪  '***', 'START', 'OF', 'THIS', 'PROJECT', 'GUTENBERG', 'EBOOK', 'MOBY',
↪  'DICK;', 'OR', 'THE', 'WHALE', '***', 'Produced', 'by', 'Daniel']
```

As we can see the empty entries are gone.

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(100))
```

```
[('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), ('it,', 1), ('give', 1), ('it', 1),
 ↪  ('away', 1), ('or', 1), ('re-use', 1), ('it', 1), ('under', 1), ('the',
 ↪  1), ('terms', 1), ('of', 1), ('the', 1), ('Project', 1), ('Gutenberg',
 ↪  1), ('License', 1), ('included', 1), ('with', 1), ('this', 1), ('eBook',
 ↪  1), ('or', 1), ('online', 1), ('at', 1), ('www.gutenberg.org', 1),
 ↪  ('Title:', 1), ('Moby', 1), ('Dick;', 1), ('or', 1), ('The', 1),
 ↪  ('Whale', 1), ('Author:', 1), ('Herman', 1), ('Melville', 1), ('Last',
 ↪  1), ('Updated:', 1), ('January', 1), ('3,', 1), ('2009', 1), ('Posting',
 ↪  1), ('Date:', 1), ('December', 1), ('25,', 1), ('2008', 1), ('[EBook',
 ↪  1), ('#2701]', 1), ('Release', 1), ('Date:', 1), ('June,', 1), ('2001',
 ↪  1), ('Language:', 1), ('English', 1), ('***', 1), ('START', 1), ('OF',
 ↪  1), ('THIS', 1), ('PROJECT', 1), ('GUTENBERG', 1), ('EBOOK', 1),
 ↪  ('MOBY', 1), ('DICK;', 1), ('OR', 1), ('THE', 1), ('WHALE', 1), ('***',
 ↪  1), ('Produced', 1), ('by', 1), ('Daniel', 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':

```
[('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), ('u
se', 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), ('it,', 1), ('give', 1), ('it
', 1), ('away', 1), ('or', 1), ('re-use', 1), ('it', 1), ('under', 1), ('the', 1), ('terms', 1), ('of', 1), ('the', 1), ('P
roject', 1), ('Gutenberg', 1), ('License', 1), ('included', 1), ('with', 1), ('this', 1), ('eBook', 1), ('or', 1), ('online
', 1), ('at', 1), ('www.gutenberg.org', 1), ('Title:', 1), ('Moby', 1), ('Dick;', 1), ('or', 1), ('The', 1), ('Whale', 1),
('Author:', 1), ('Herman', 1), ('Melville', 1), ('Last', 1), ('Updated:', 1), ('January', 1), ('3,', 1), ('2009', 1), ('Pos
ting', 1), ('Date:', 1), ('December', 1), ('25,', 1), ('2008', 1), ('[EBook', 1), ('#2701]', 1), ('Release', 1), ('Date:',
1), ('June,', 1), ('2001', 1), ('Language:', 1), ('English', 1), ('***', 1), ('START', 1), ('OF', 1), ('THIS', 1), ('PROJEC
T', 1), ('GUTENBERG', 1), ('EBOOK', 1), ('MOBY', 1), ('DICK;', 1), ('OR', 1), ('THE', 1), ('WHALE', 1), ('***', 1), ('Produ
ced', 1), ('by', 1), ('Daniel', 1)]
```

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.

```
print(counts.take(100))
```

```
[('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),
↪  ('Lazarus', 6), ('Original', 1), ('combination', 2), ('now-defunct', 1),
↪  ('project', 3), ('version', 3), ('are', 586), ('University', 1),
↪  ('Adelaide', 1), ('preserving', 3), ('version.', 1), ('resulting', 3),
↪  ('was', 1566), ('compared', 9), ('public', 13), ('domain', 8), ('89,',
↪  1), ('we', 395), ('L', 1), ('symbol', 10), ('currency.', 1),
↪  ('ETYMOLOGY.', 1), ('(Supplied', 2), ('Consumptive', 1), ('Usher', 1),
↪  ('School)', 1), ('pale', 12), ('in', 3878), ('now.', 19), ('ever', 175),
↪  ('dusting', 2), ('his', 2415), ('lexicons', 1), ('grammars,', 1),
↪  ('mockingly', 1), ('flags', 1), ('known', 68), ('nations', 10),
↪  ('loved', 3), ('grammars;', 1), ('somehow', 30), ('mildly', 9),
↪  ('reminded', 4), ('mortality.', 1), ('take', 116), ('school', 5),
↪  ('others,', 9), ('them', 273), ('name', 47), ('whale-fish', 1), ('out,',
↪  37), ('ignorance,', 2), ('letter', 8), ('maketh', 5), ('--HACKLUYT', 1),
↪  ('named', 9), ('arched', 6), ('vaulted."', 1), ("--WEBSTER'S", 1),
↪  ('It', 260), ('more', 428), ('Dut.', 1), ('Ger.', 1)]
```

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())      # <- include the
 ↪  UPPERCASE here.
```

```
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 another application of Map-Reduce I found the very popular term frequency–inverse document frequency (TF-idf)[4] very interesting . This is a numerical statistic, which is often used in text-based recommender systems and for information retrieval. In my example I used the texts of "Moby Dick"

[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

and "Tom Sawyer". The result are two lists of most important words for each of these documents. This is what the TF-idf found:

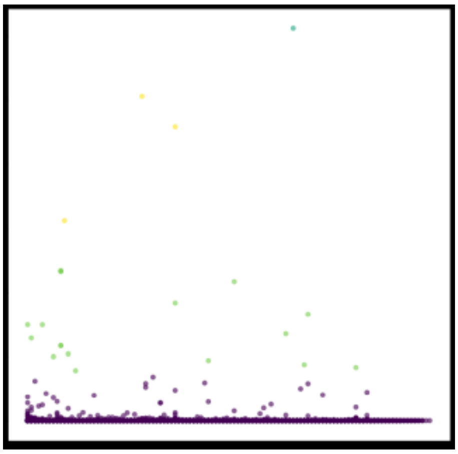
```
Moby Dick:
    WHALE, AHAB, WHALES, SPERM, STUBB, QUEEQUEG, STRARBUCK, AYE

Tom Sawyer:
    HUCK, TOMS, BECKY, SID, INJUN, POLLY, POTTER, THATCHER
```

Applications for using TF-idf are in the information retrieval[5] or to classify documents.

On my Docker machine-learning repository[6] you will find the Jupyter-Notebook, with this example.

5.2 K-Means Clustering Algorithm

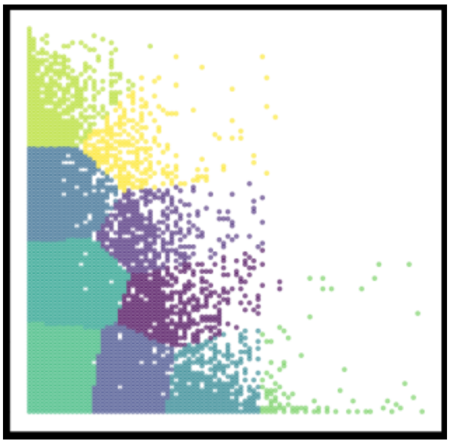
Additionally I worked on this dataset to show how the K-Means Clustering Algorithm[7] can be applied by using the Spark Machine-Learning Library. I will show how the "Vermont Vendor Payments" dataset can be clustered. In the images below every color represents a different cluster:



[5] Information Retrieval, https://en.wikipedia.org/wiki/Information_retrieval

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

[7] K-Means Clustering Algorithm, https://spark.apache.org/docs/latest/ml-clustering.html#k-means



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 industry” is huge and I hope this short introduction is helpful to get started.

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).

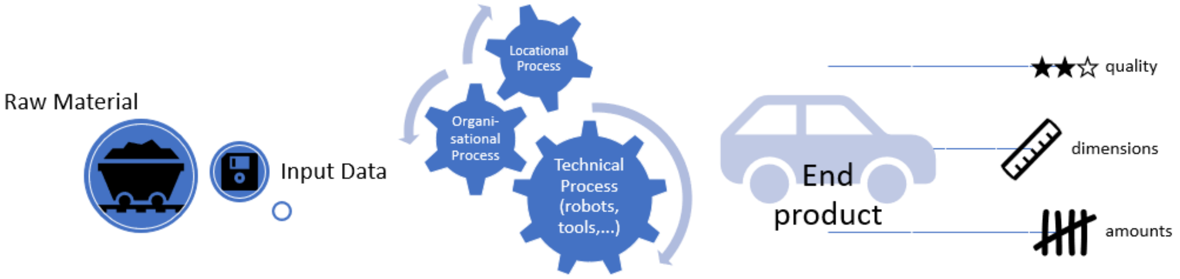


Figure 6.1: Use Cases of AI - Chart

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]

[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

- **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]

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". 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 (to calculate with networked

[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-in-der-qualitaetskontrolle-wenn-kein-detail-uebersehen-werden-darf-a-860403/

computers) or Hadoop (to process very large data sets distributed across several computers). I have been working here to describe the difference when working with a Big Data system compared to conventional data processing.

- 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”. Here, the log files that record the processes are often transformed into models and then evaluated.

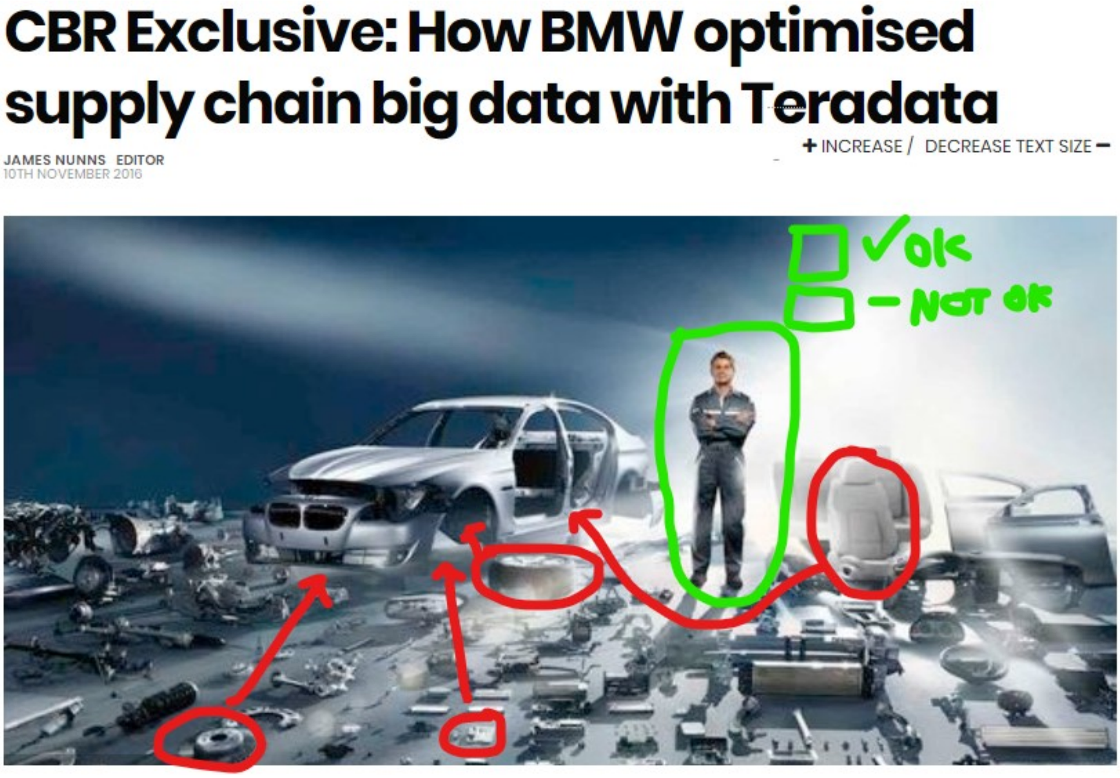
- One speaks of a ”deep learning” approach when neural networks are used. This is often the case when the input data are not just simple data points, as is the case with sensor data, but have a complex structure, such as images or documents. An image consists of several thousand pixels in each of the two image axes and for each of these pixels there are many possible shades of colour. 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 here.

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 an interesting study here in December 2019: In it, 300 companies from the industrial manufacturing, automotive, consumer goods, aerospace and defence 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 on which this study is based, among others (see https://www.cbronline.com/big-data/analytics/bmw-optimised-supply-chain-teradata-big-data/):



Source: BMW Group

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, data lake).

BMW celebrated a success in 2016 when it was able to gain valuable insights from analysing its inventory: the teams were able to reduce inventory costs by 70% by gaining more transparency across their many production sites and optimising processes.

In 2016, Klaus Straub, CIO at BMW, described in this article the ideas for the digital transformation of the company. Even then, 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) 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 programme 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" network already trained on the "ImageNet" 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 on the left image it sees a "cup" with 86% probability and a "coffee cup (coffepot)" with 6.8%. In the right picture, [6]the

[6] Pixabay, network map, https://pixabay.com/de/photos/netzwerkkarte-bauteil-schaltkreis-550544/

model sees a "switch component" with 76% probability. You can see my programme code here

```
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)]]
```

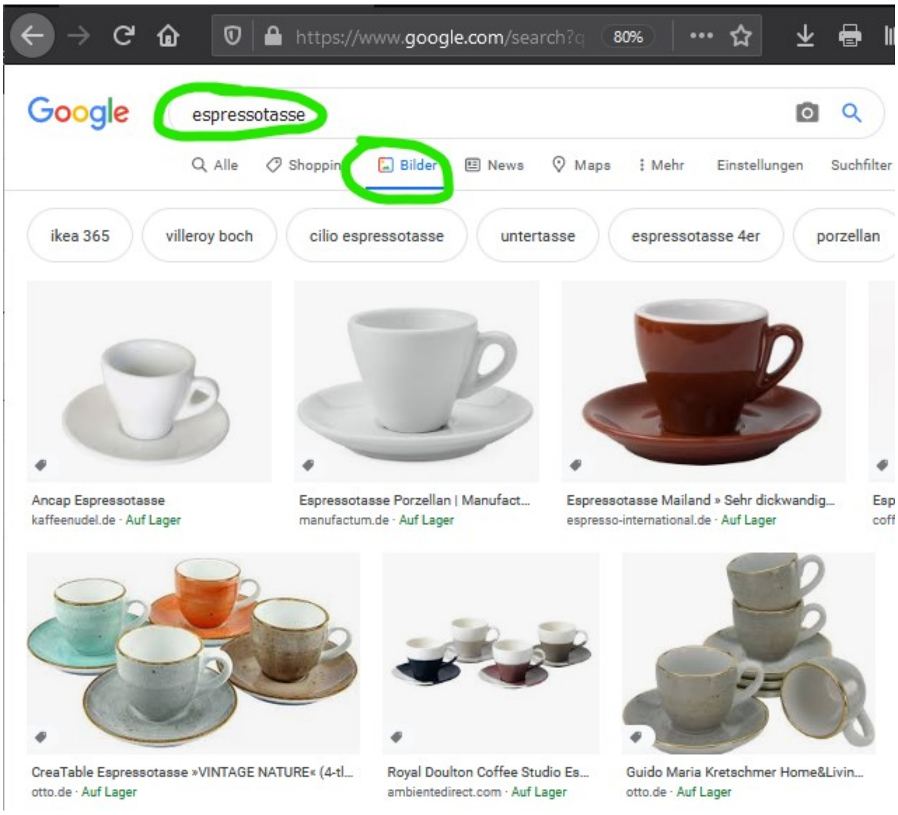
```
[[('n04372370', 'switch', 0.76144683),
  ('n03777754', 'modem', 0.08100146),
  ('n03272010', 'electric_guitar', 0.032640774),
  ('n03492542', 'hard_disc', 0.015652671),
  ('n04070727', 'refrigerator', 0.008717526)]]
```

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[7]. 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

[7] Die Zeit, Artificial Intelligence: When Computers Evaluate X-Ray Images, https://www.zeit.de/wissen/2019-09/kuenstliche-intelligenz-medizin-diagnose-krankheiten-bilddiagnostik

to show similar images. You enter a term (e.g. espresso cup) in the search bar and similar images are displayed:



Two questions arise when implementing the program code. The first question is:

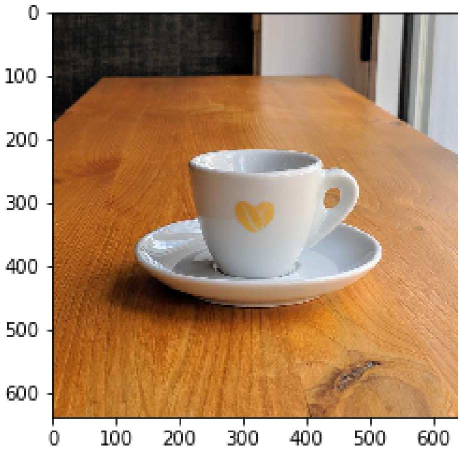
How do you compare two pictures? Or two texts? It's not easy, but this problem has been analyzed many times. So there are procedures that you just have to copy, and I will show you how in a moment.

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. 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).[8] 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.

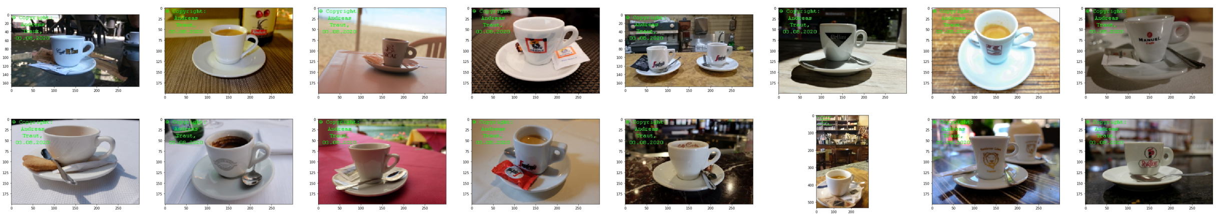
The result of my work was: I applied the LSH algorithm to over 9000 images (each about 300*300 pix-

[8] Towards Datascience, Locality Sensitive HashingAn effective way of reducing the dimensionality of your data, https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

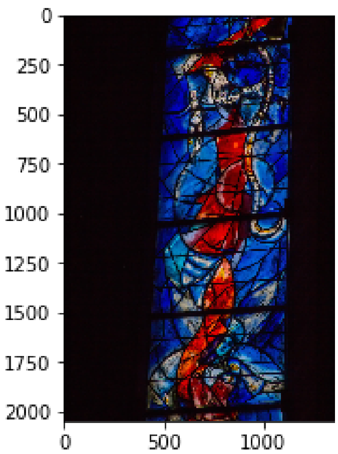
els), including my collection of church windows and espresso cups (I like to drink espresso and started photographing the cups at some point). On my own computer, this grouping took a few minutes and was only necessary once. After that, I downloaded a completely new picture of an espresso cup from the internet:



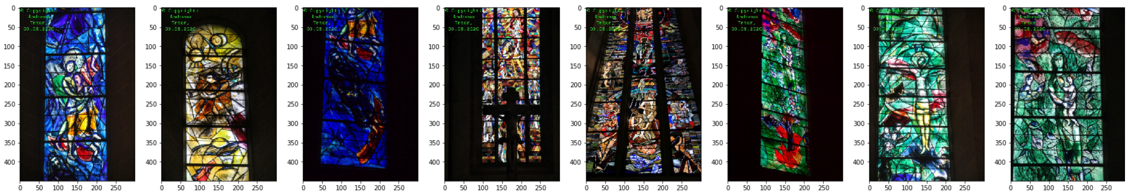
I gave this picture to the program and then used it to search for similar pictures from my picture collection. After a few seconds, the program showed me these 15 pictures:



I then tested the same for my church window images with this test image:



Result:



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 see my program code here and read my further explanations here. I have also explained the deep-learning topic in more depth here.

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:

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 here.

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.

List of Figures