From Zero to Senior Data Science

An guide into
advanced Data Science approaches
and Big Data technologies

By Andreas Traut

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 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 MIT license (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 with it.

1.2 What is Data Science? Who is a Data Scientist?

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 and important field. Even HR managers struggle in correctly evaluating the meaning and requirements of "Data Science".

What each individual understands by "Data Science" is as elastic as chewing gum. Nevertheless, it is used as a job title in many job descriptions, often with completely different interpretations. One per-

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

son 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. The other 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. Who has lived two lifetimes?

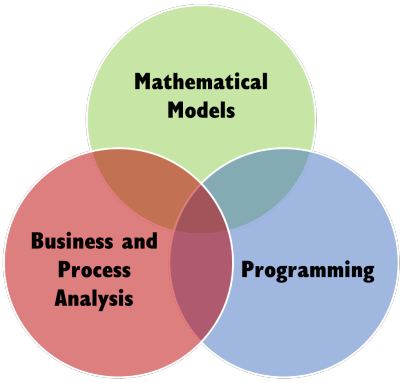


Figure 1.1: Data Science - Venndiagram

There is a much discussed and generally quite accepted definition of what a Data Scientist is: the Data Science Venn Diagram[2]. For those who are not yet familiar with this 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.

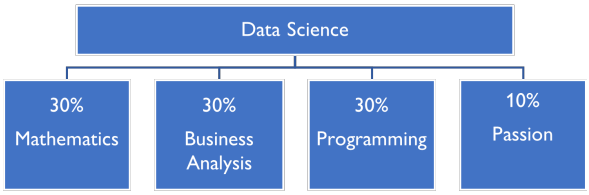


Figure 1.2: Data Science

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), 30% programming experience with machine-learning libraries and 10% passion for new exciting topics and willingness to dive into new tools.

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

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

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!

I hope that one day in the future the term “Data Scientist” will be protected a bit better with certificates or other proofs of qualification. At the moment, unfortunately, “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. **??**) I will explain how to install the basic Data Science Tools, which you need. In my examples I use Jupyter-Notebooks[3], which is a widespread standard today, but I also use an Integrated Development Environment[4]. 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. **??**) I will explain these examples. I used python and libraries like e.g Matplotlib[5] or Seaborn[6], which are available for free. I will show you in Section sec. **??** how to install these tools on your own computer.

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

In the **second part** (see [#sec:DataAppStreamlit] 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).

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

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

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

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

In the **third part** sec. **??** 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 Installation of Data Science Tools

In this chapter (see sec. **??**) I will explain how to install the basic Data Science Tools, which you need. In my examples I use Jupyter-Notebooks[7], which is a widespread standard today, but I also use an Integrated Development Environment[8]. 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.

1.3.4 Machine Learning with Python

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 this chapter (see sec. 4) I am particularly interested in better understanding 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 instead of digging into too many details of each of these approaches. 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. **??**) 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. **??**) is for being used in an IDE (Integrated Developer Environment), like the Spyder-IDE[9]from the Anaconda distribution[10] and applies the *Scikit-Learn Python Machine Learning Library*[11] (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.

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

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

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

[10] Anaconda distribution, https://www.anaconda.com/

[11] Scikit-Learn, https://scikit-learn.org/

The **third example** (see sec. **??**) is a "Big Data" example and will use a Docker environment[12] and apply the *Apache Machine Learning Library*[13], a scalable machine learning library. The mind-map from the second part will be extended and aligned to the second example.

The **fourth part** (see sec. **??**) is a digression (Excurs). I will explain some Big Data Visualizations techniques, show how the K-Means Clustering Algorithm in Apache Spark ML works and explain the Map-Reduce programming model on a Word-Count example.

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*[14] (see fig. 1.3) which covers topics like data visualization and exploratory data analysis on the basis of Python[15], Numpy[16], Pandas[17], Matplotlib[18] and Seaborn[19]. I can recommend this course and I wish I would have found this course before I wrote this repository, because it was very helpful.

[12] Docker environment, https://www.docker.com/
[13] Apache Machine Learning Library, https://spark.apache.org/mllib/
[14] Certificate of "Data Analysis with Python: Zero to Pandas, https://jovian.ai/certificate/MFQTGOBQGM
[15] Python, https://www.python.org/
[16] Numpy, https://numpy.org/
[17] Pandas, https://pandas.pydata.org/
[18] Matplotlib, https://matplotlib.org/
[19] Seaborn, https://seaborn.pydata.org/

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

1.5 MIT License

2 Installation of Data Science Tools

2.1 Jupyter-Notebooks and Spyder-IDE

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

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.

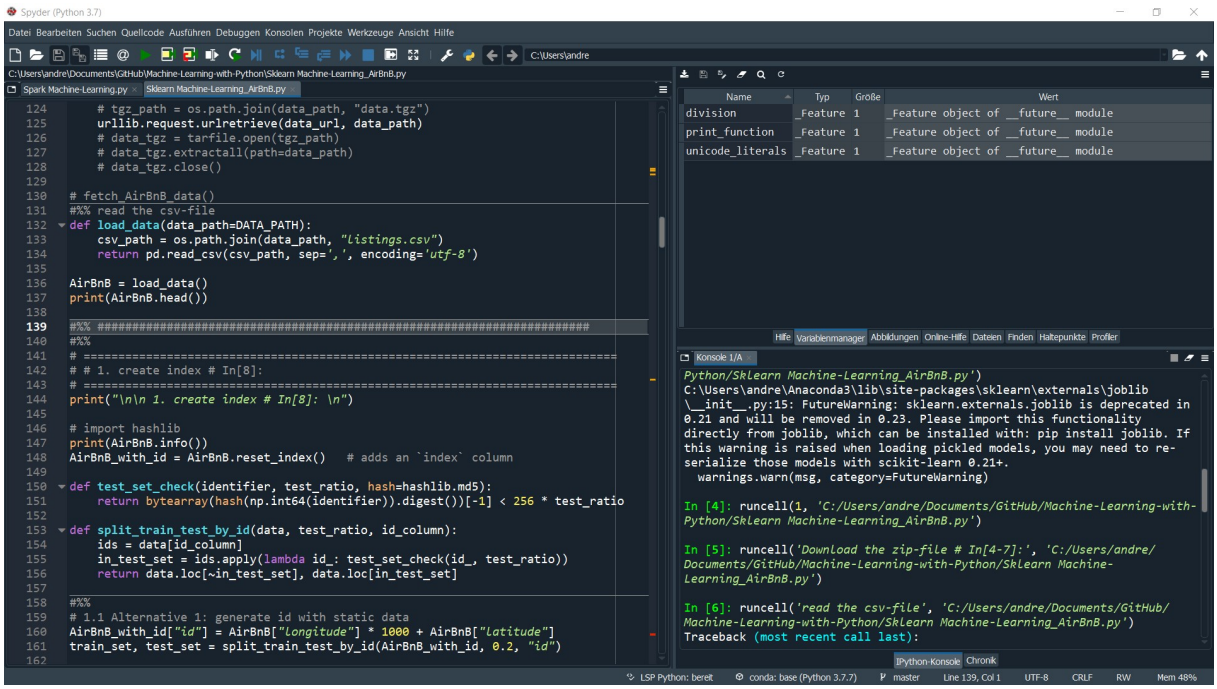


Figure 2.1: Spyder-IDE: An integrated development environment with debugger.

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.

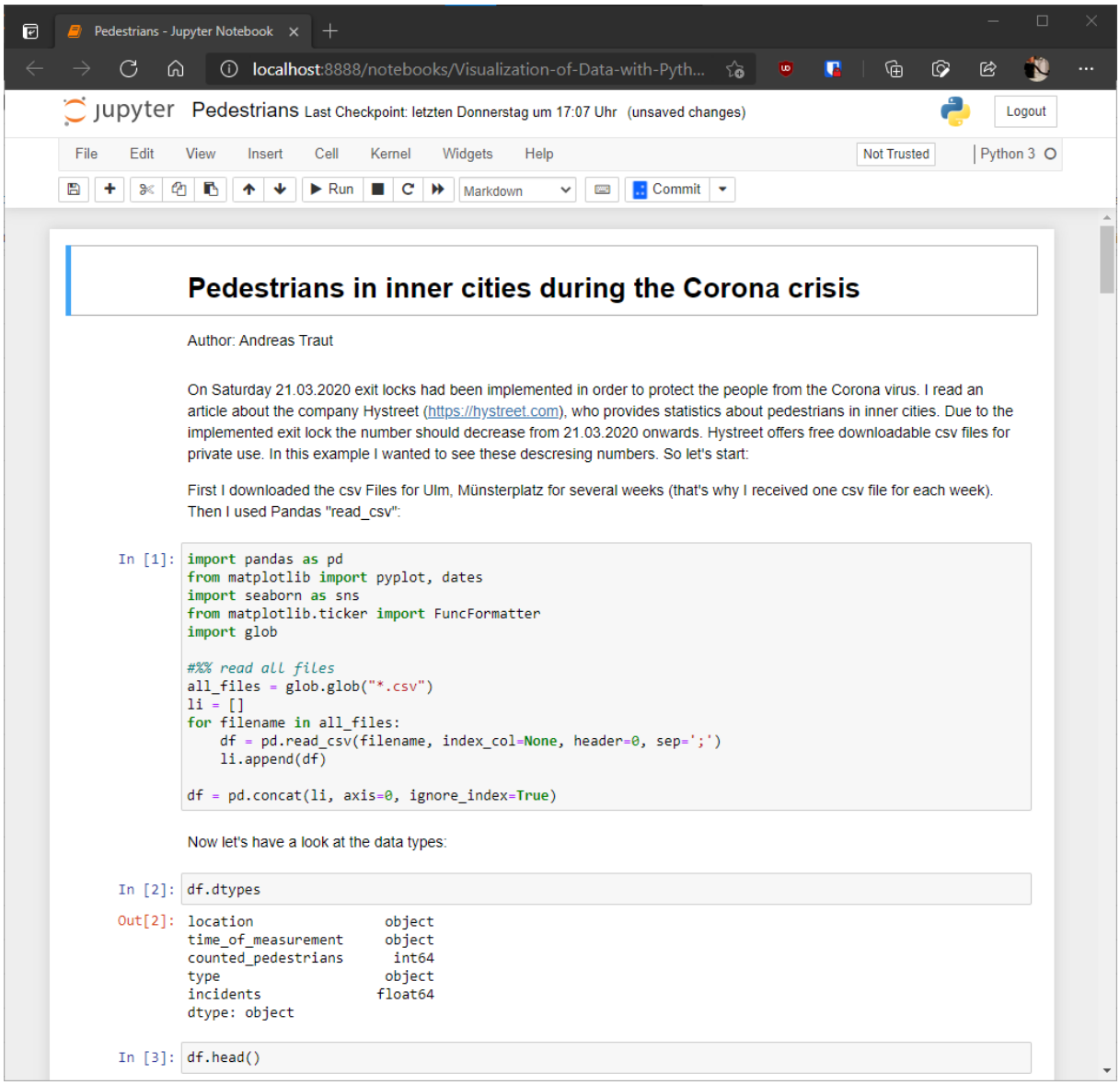


Figure 2.2: Jupyter-Notebook: Code and Documentation in one place.

[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[8] environment. :-)

2.3 Docker

For more advanced Data Science techniques, like the Big Data approaches on Apache Spark[9] and Hadoop[10], you will need Docker[11]. Download it from “Docker - Get Started”[12] and also create an account on the Docker website. The installation is easy.

[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] MS-DOS, https://de.wikipedia.org/wiki/MS-DOS
[9] Apache Spark, https://spark.apache.org/
[10] Hadoop, https://hadoop.apache.org/
[11] Docker environment, https://www.docker.com/
[12] Docker Get Started, https://www.docker.com/get-started

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."*

After having installed it, you are able to pull my "machine-learning-pyspark" image to your computer and run my Jupyter-Notebooks. I will explain in section sec. 4 "Machine Learning with Python" how it works. Please read also the Docker-Curriculum[13] for more information. It is a very well structured and nice tutorial which I can recommend for learning about docker images, docker containers and more.

[13] 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:

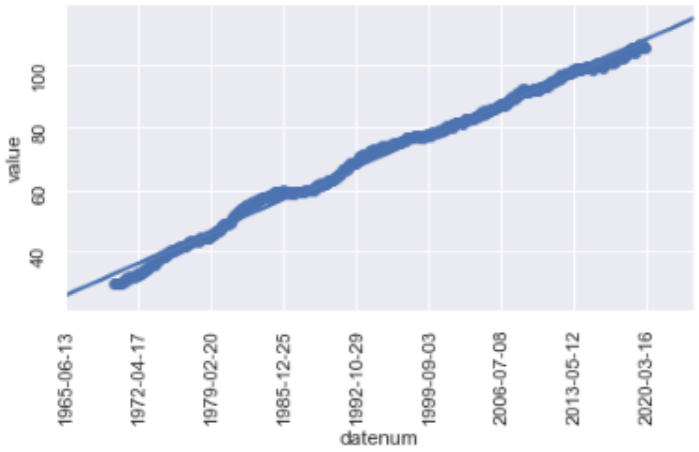


Figure 3.2: Consumer Price Index Figure

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))
```

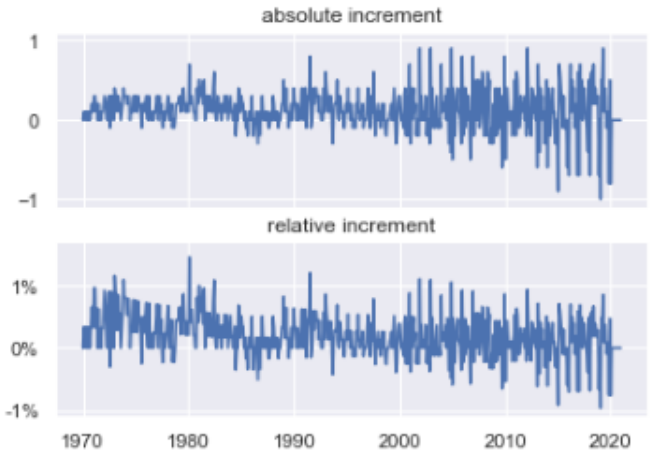


Figure 3.3: Consumer Price Figure - absolute and relative increments

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).

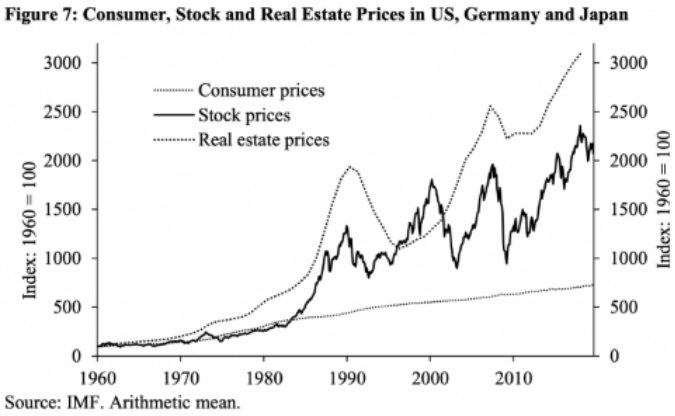


Figure 3.4: Consumer price index versus Stockprices, Real estate prices

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/master/ConsumerPricesExample/ConsumerPrices.ipynb

[7] LastFM, www.last.fm

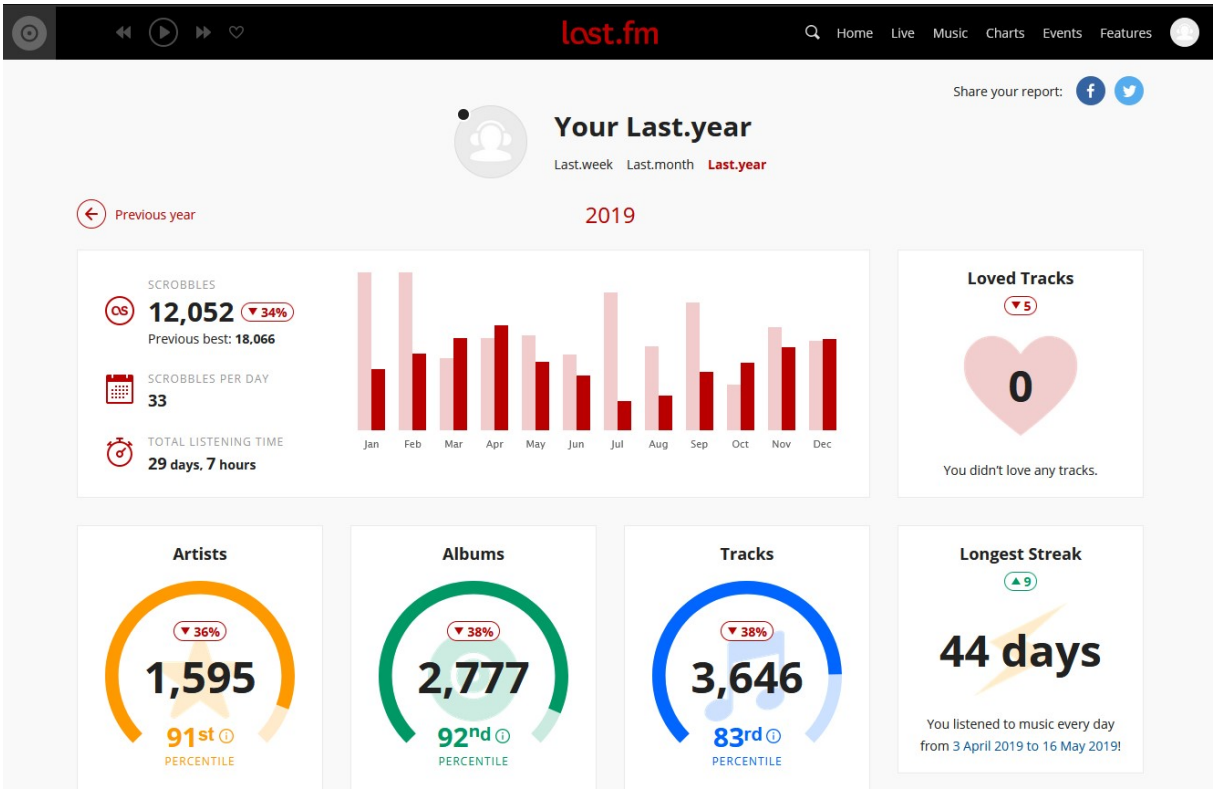


Figure 3.5: Last-FM Music Statistics - Overview year 2019

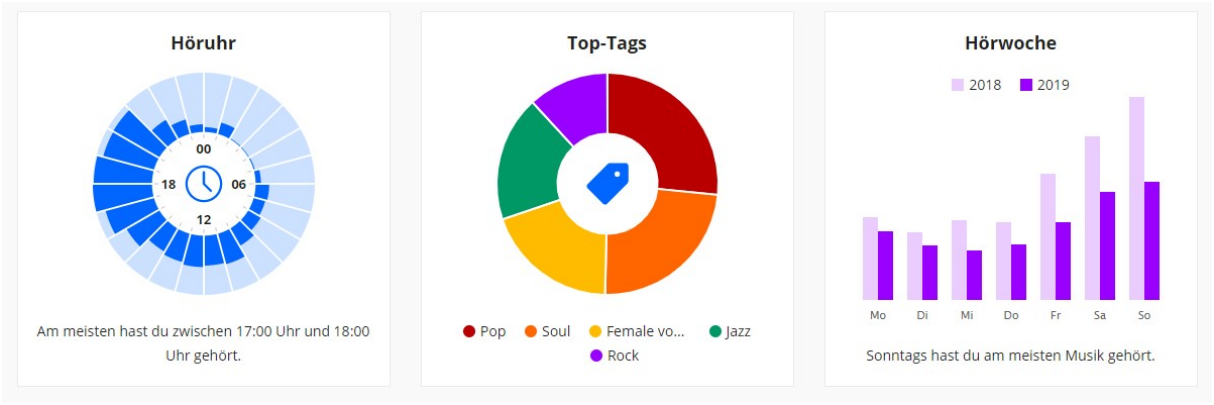


Figure 3.6: Overview listening clock 2019

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.
```

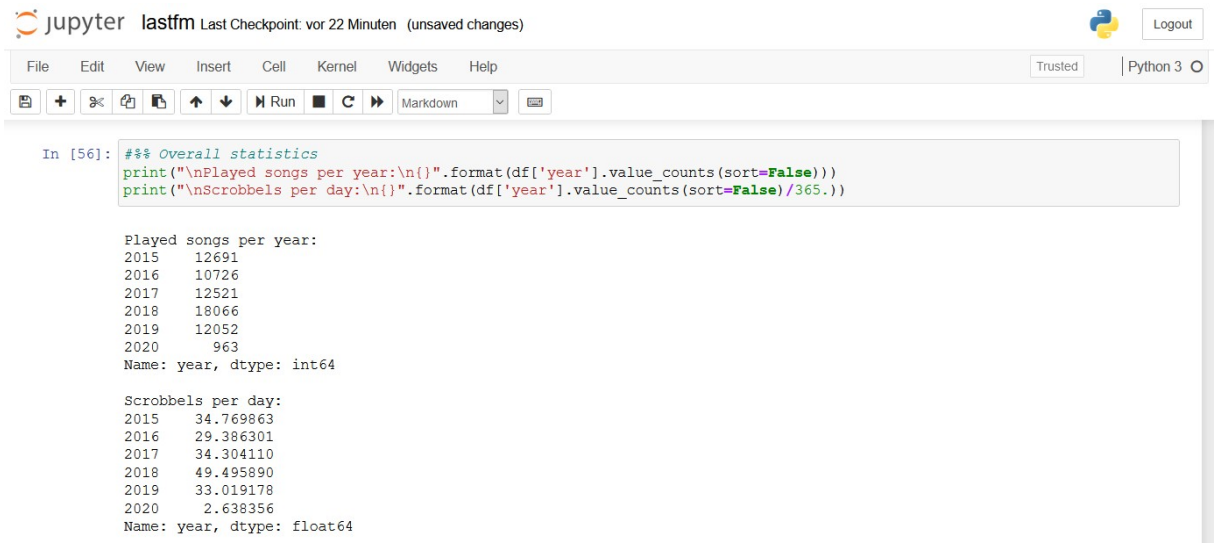


Figure 3.8: Last-FM Music Statistics - Overall statistics

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()
```

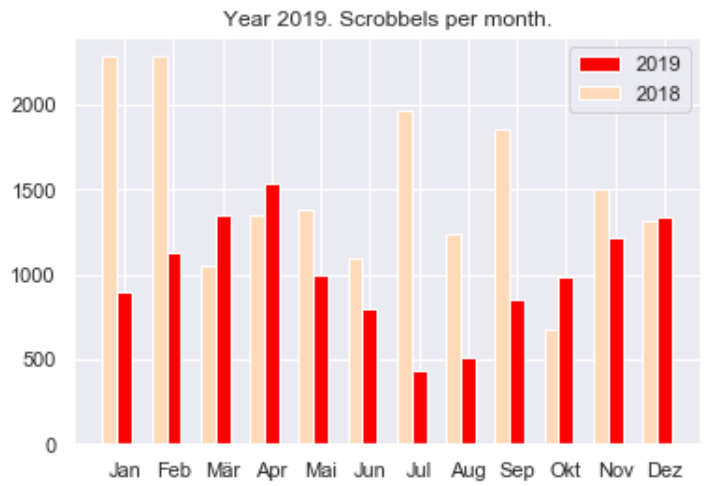


Figure 3.9: Last-FM Music Statistics - Reproduced Statistics of 2019

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:

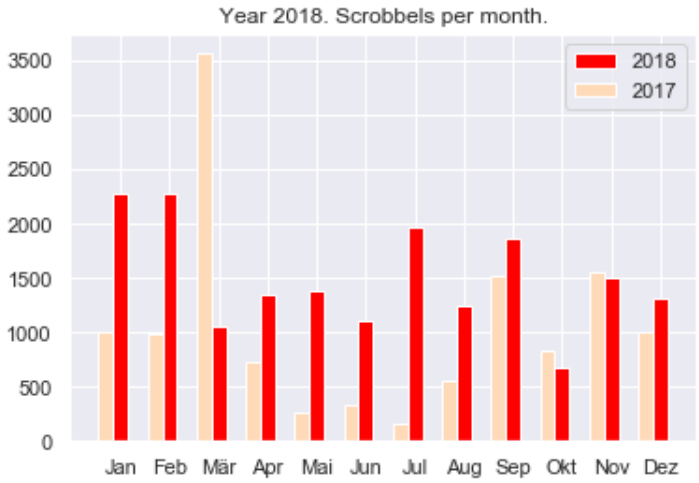


Figure 3.11: Last-FM Music Statistics - Reproduced Statistics of 2018

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:

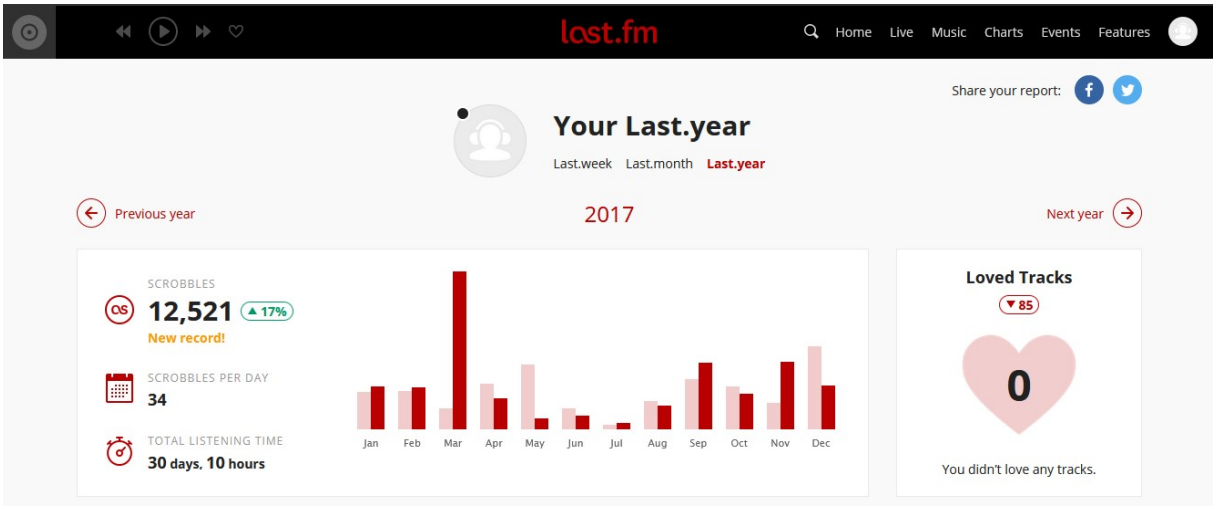


Figure 3.12: Last-FM Music Statistics - Overview year 2017

My graphics for 2017:

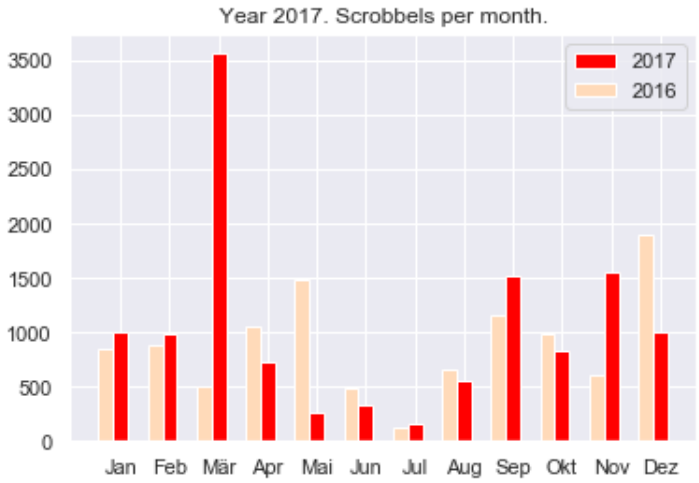
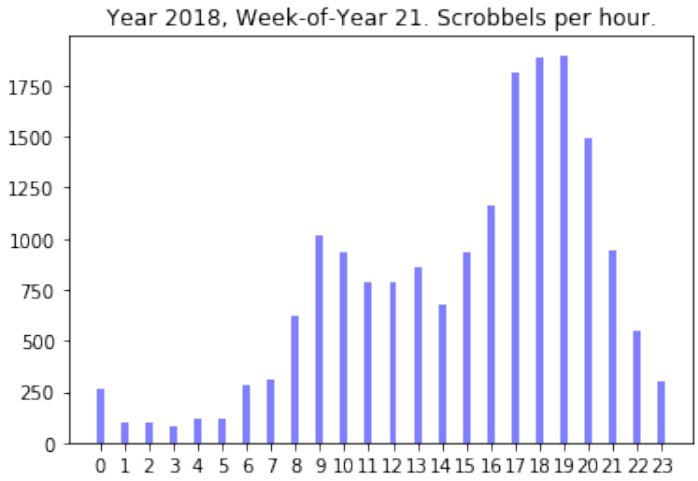


Figure 3.13: Last-FM Music Statistics - Reproduced Statistics of 2017

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()
```



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?

```
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
....
```

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

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')
```

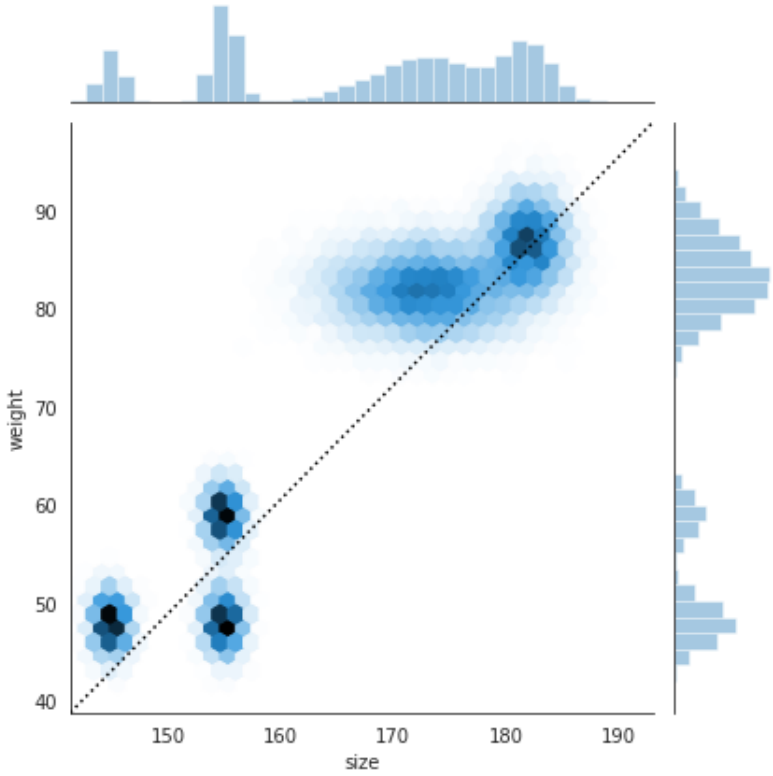


Figure 3.14: Marathon Example - Seaborn Jointplot

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)
```

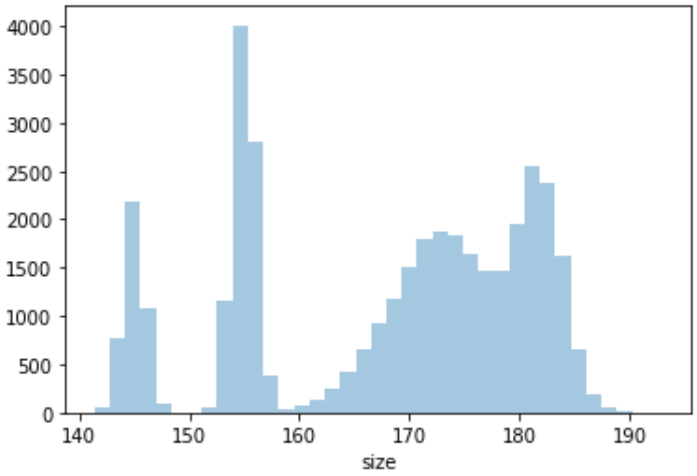


Figure 3.15: Marathon Example - Seaborn Histogram “size”

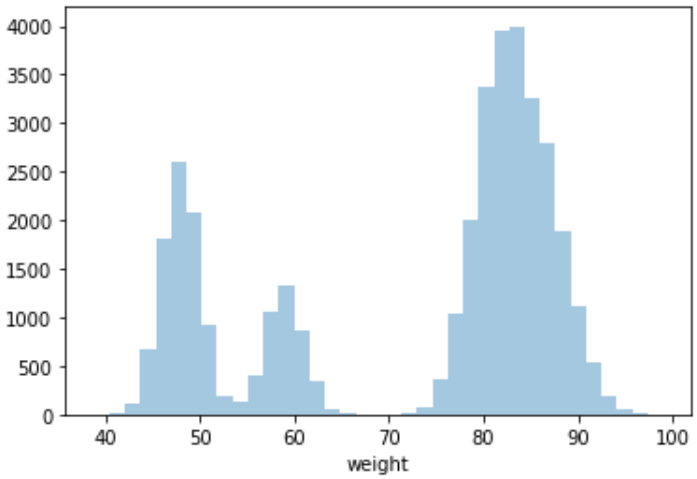


Figure 3.16: Marathon Example - Seaborn Histogram “weight”

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();
```

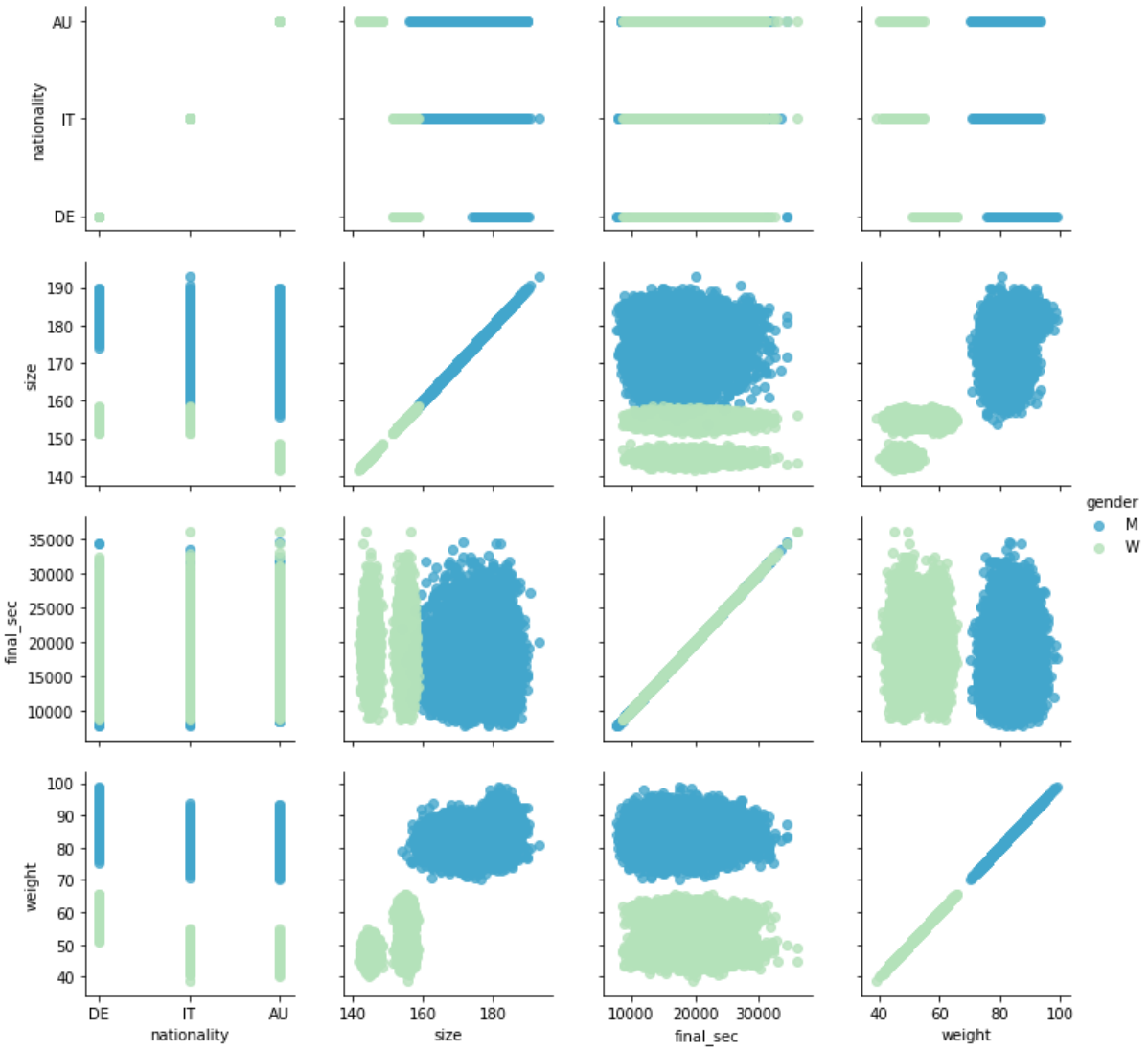


Figure 3.17: Marathon Example - Seaborn PairGrid

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()
```

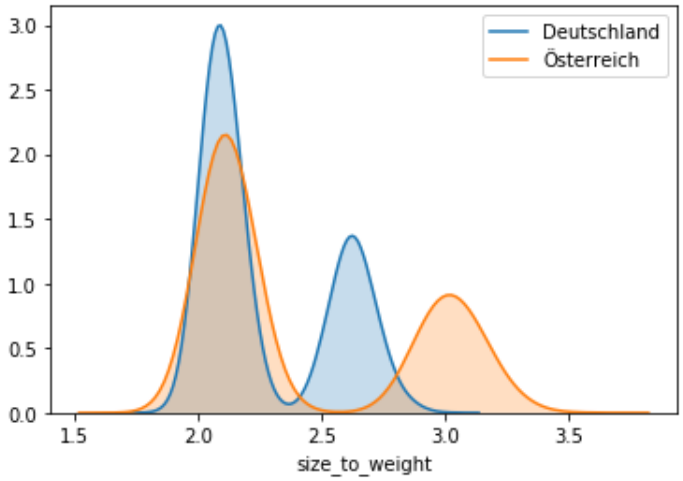


Figure 3.18: Marathon Example - Seaborn Kernel Density "size-to-weight"

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

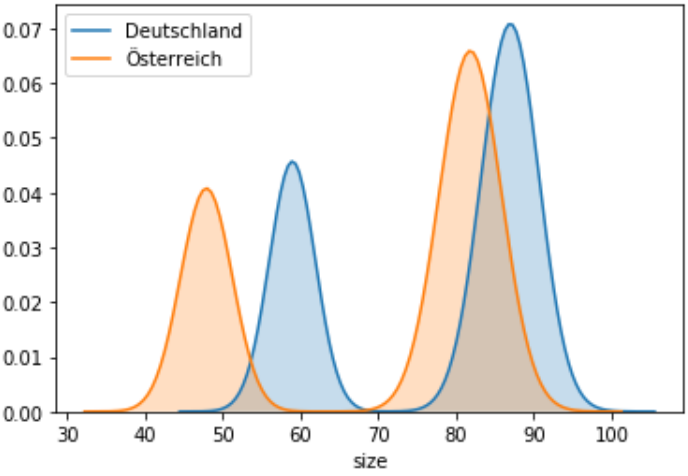


Figure 3.19: Marathon Example - Seaborn Kernel Density "size"

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

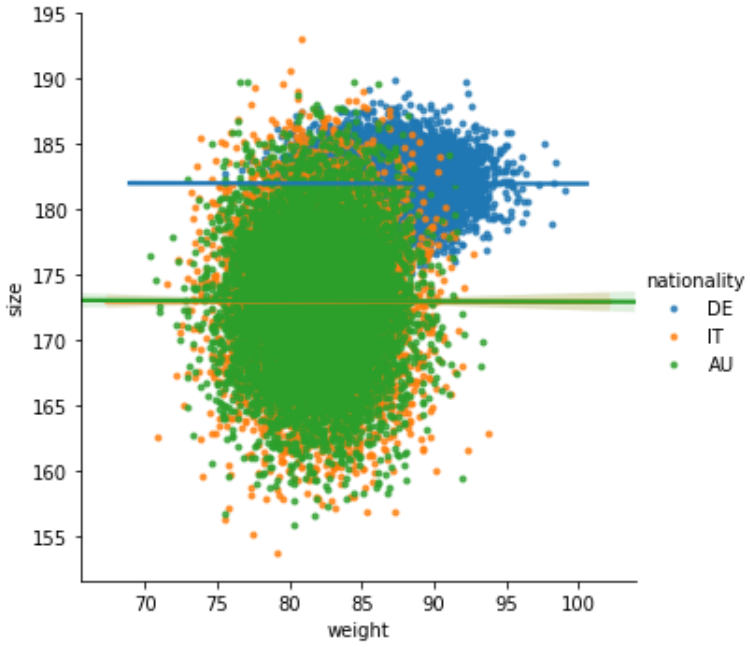


Figure 3.20: Marathon Example - Seaborn Regression Plots “men”

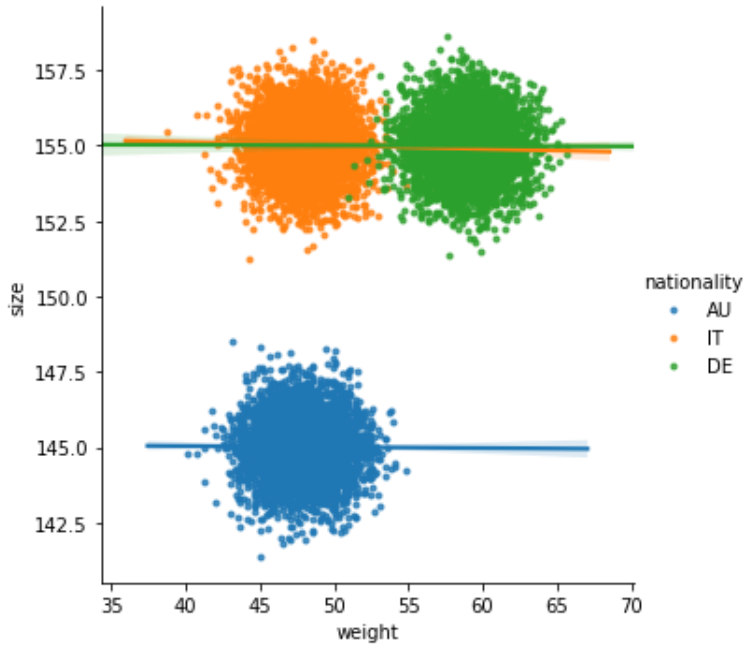


Figure 3.21: Marathon Example - Seaborn Regression Plots “women”

```
#%% 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).

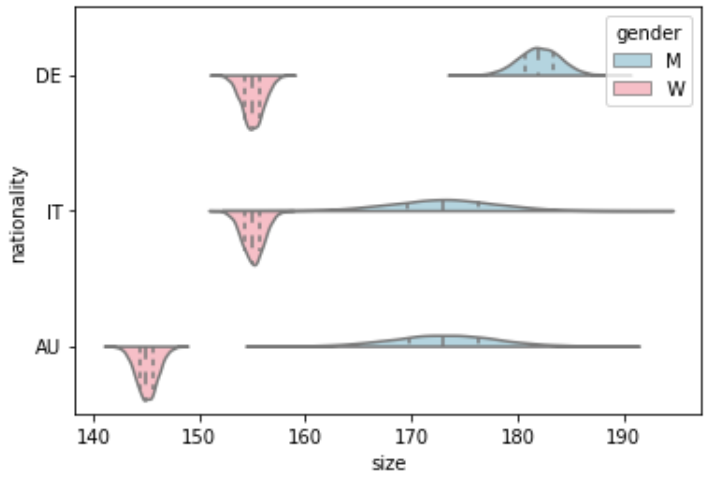


Figure 3.22: Marathon Example - Seaborn Violinplot “size”

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

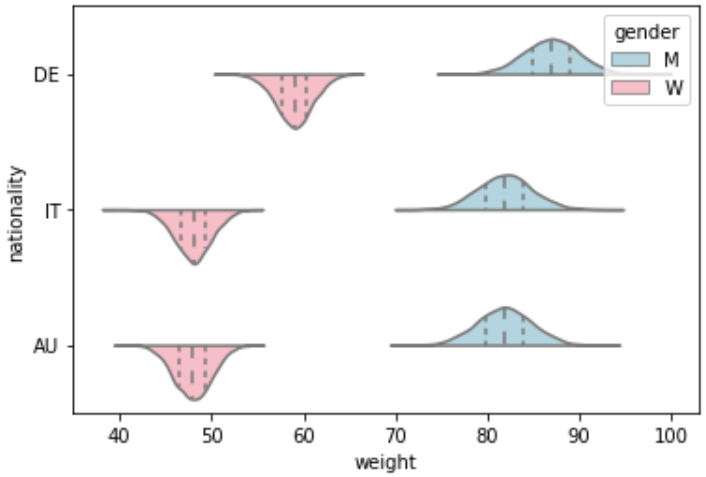


Figure 3.23: Marathon Example - Seaborn Violinplot “weight”

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. **??** 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:

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

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

Figure 3.24: 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]

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:

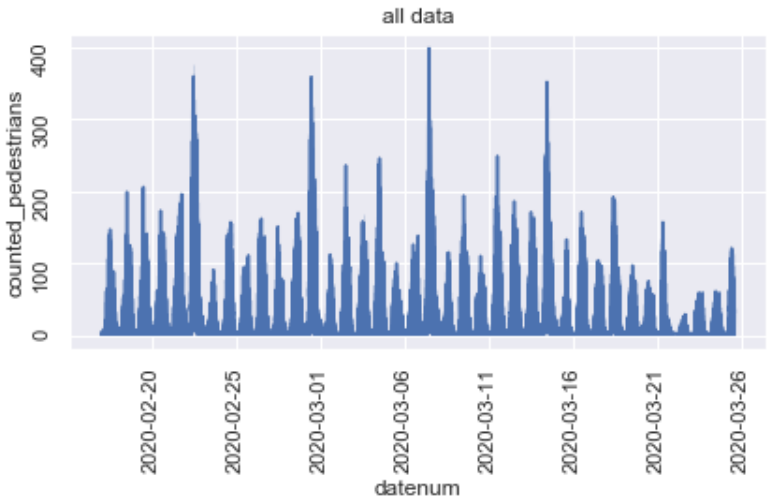


Figure 3.25: Pedestrians in Corona-Lockdown - Frequency

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

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

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

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

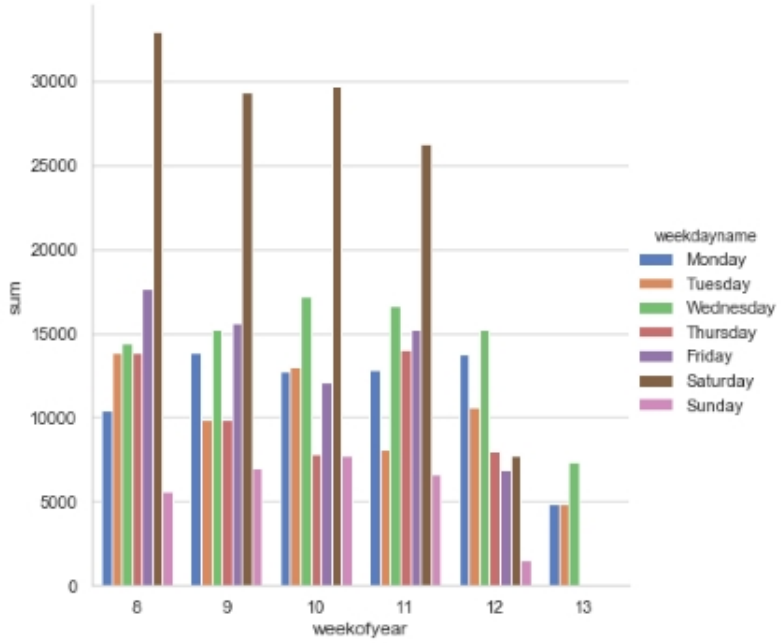


Figure 3.26: Pedestrians in Corona-Lockdown - Barplot “Ulm, Münsterplatz”

And here are some more graphics for München and Augsburg

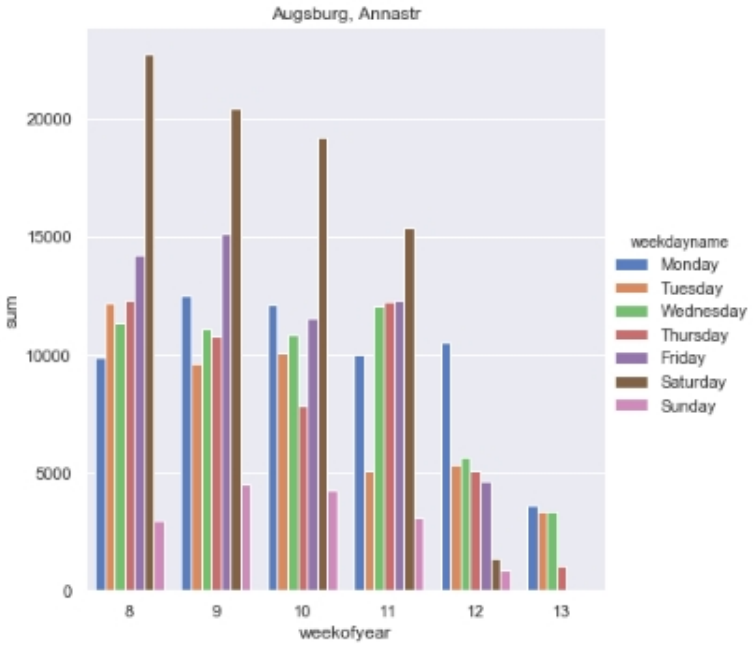


Figure 3.27: Pedestrians in Corona-Lockdown - Barplot “Augsburg”

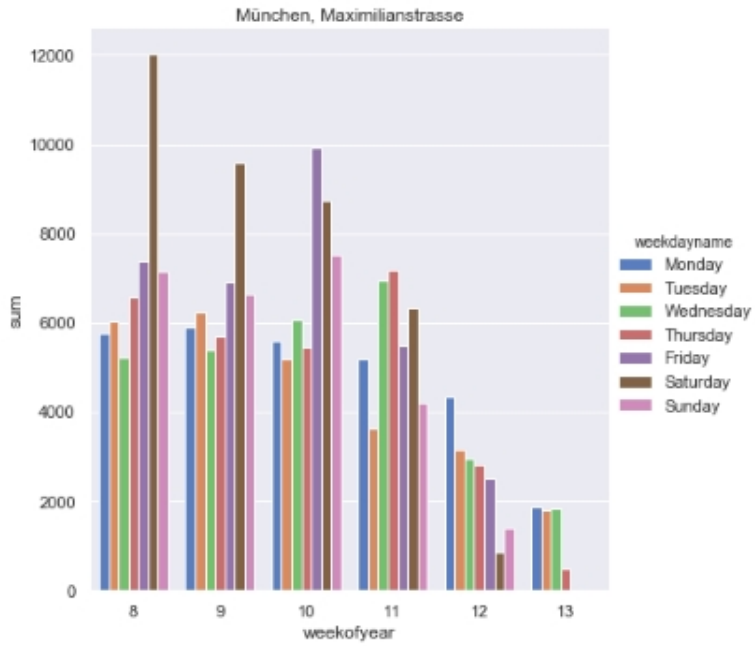


Figure 3.28: Pedestrians in Corona-Lockdown - Barplot “München, Maximilianstrasse”

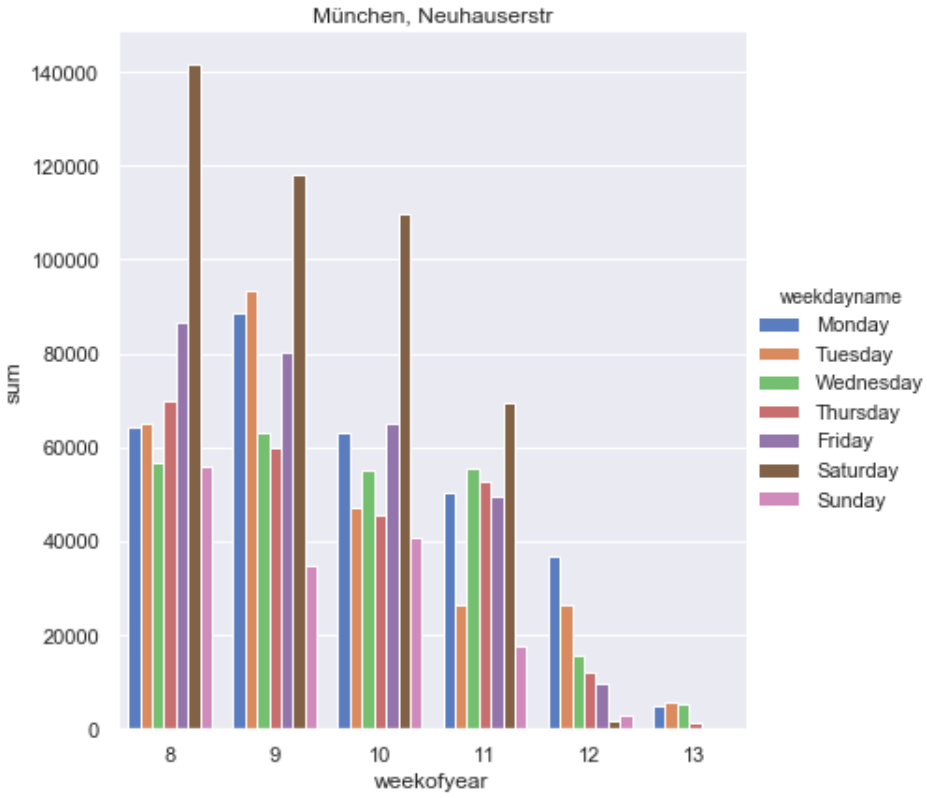


Figure 3.29: Pedestrians in Corona-Lockdown - Barplot “München, Neuhausstrasse”

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:

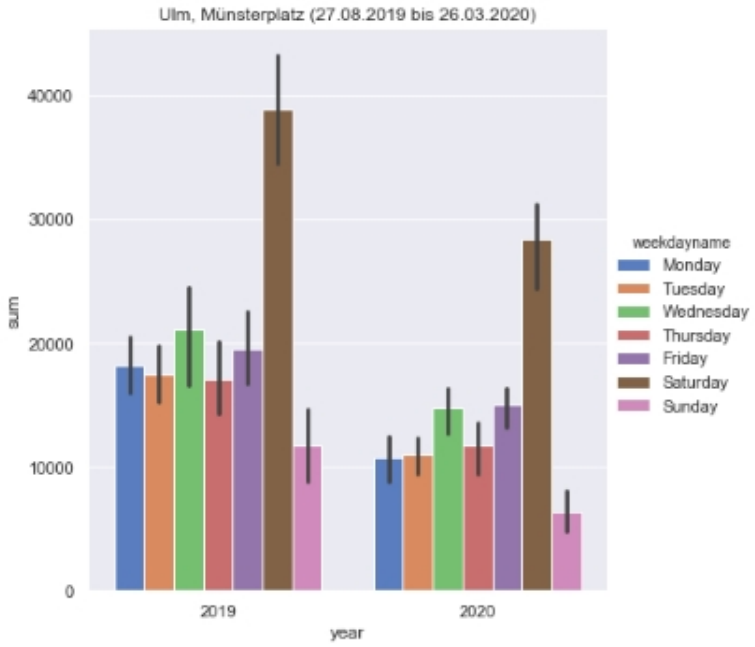


Figure 3.30: Pedestrians in Corona-Lockdown - Barplot “Ulm” 08/2019 to 03/2020

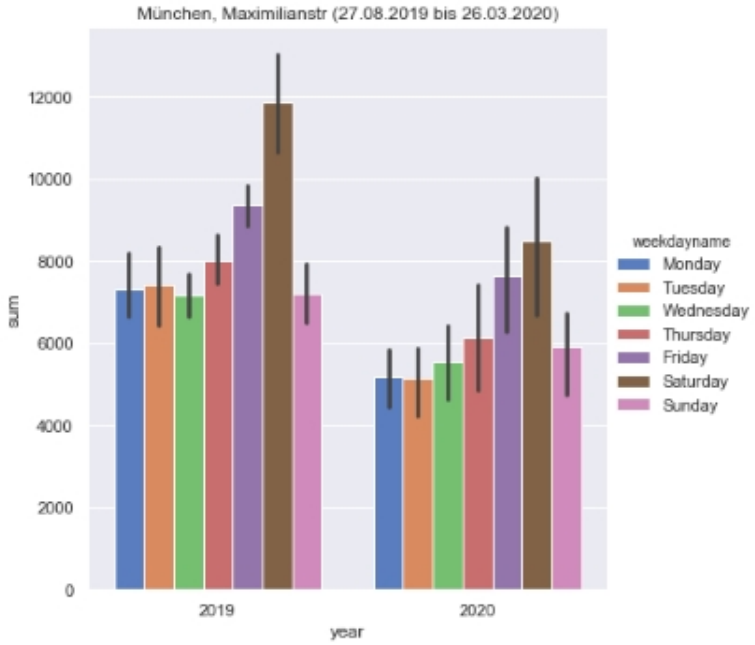


Figure 3.31: Pedestrians in Corona-Lockdown - Barplot “München, Maximilianstrasse” 08/2019 to 03/2020

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:

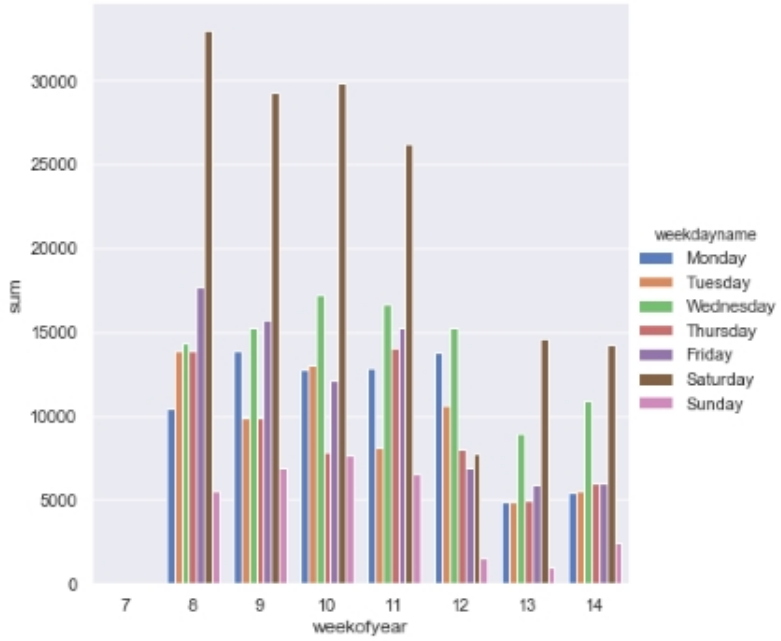


Figure 3.32: Pedestrians in Corona-Lockdown - Barplot "Ulm" April 2020

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/master/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

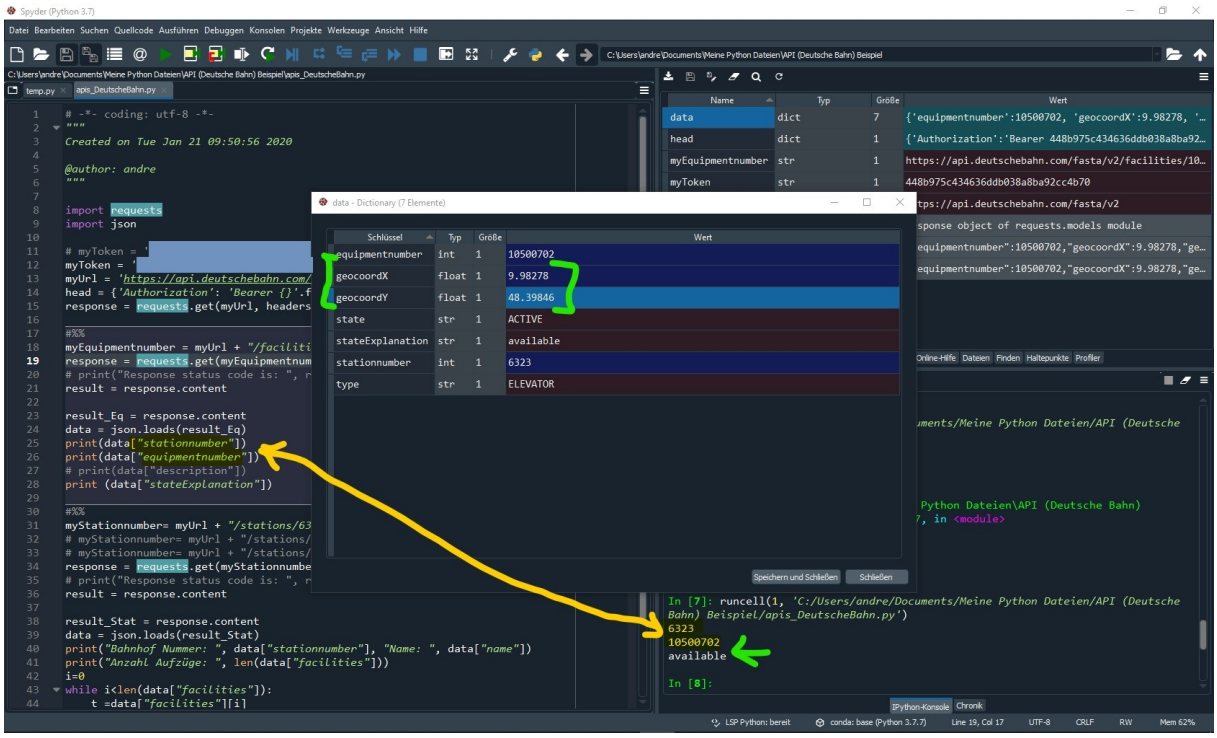


Figure 3.33: Deutsche Bahn - Elevators 1

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. 3.34, 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. 3.35.

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

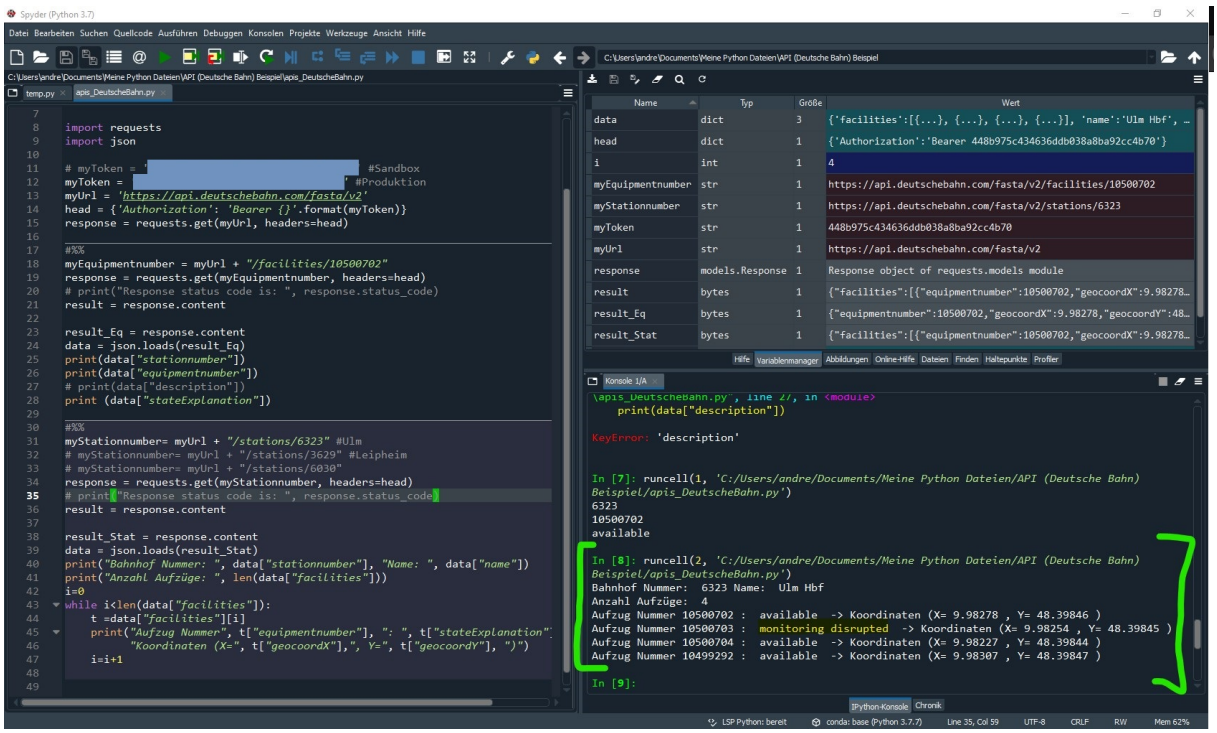


Figure 3.34: Deutsche Bahn - Elevators 2

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

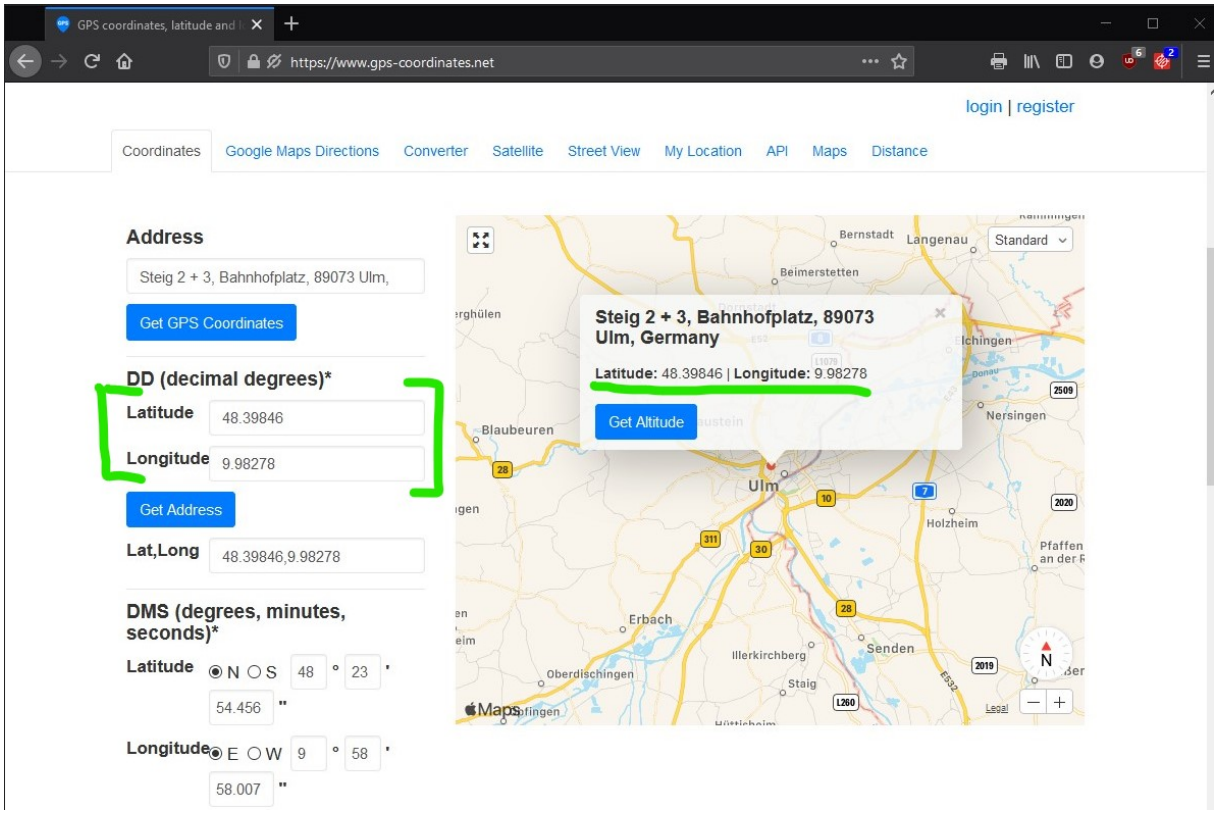


Figure 3.35: Deutsche Bahn - Latitude and Longitude

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 Introduction into Visualization of Big Data

I will start with a first introduction into visualization of Big Data now, because I think you are already able to do it now and come back to the Big Data topic later again:

I downloaded the list of all state of Vermont payments to vendors [18] (Open Data Commons License), which is a 298 MB huge csv file with 1.6 million lines (exactly 1’648’466 lines). Visualization of such huge datasets can be difficult with common tools like Excel. You can use Power-Query[19] or something

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

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

similar, but there are many benefits of using PySpark instead of Excel: the possibility to apply machine-learning algorithms from Spark Machine Learning Library[20] is only one to be mentioned.

Once you are into the Spark environment you can easily aggregate, sort, group by what ever you want. I will show you in section sec. 4 how this works. 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)
```

The result are the following 10 lines of data, which can easily be moved into Pandas[21] or Excel where creating of bar-plots is very easy.

	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 3.36: 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.

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

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

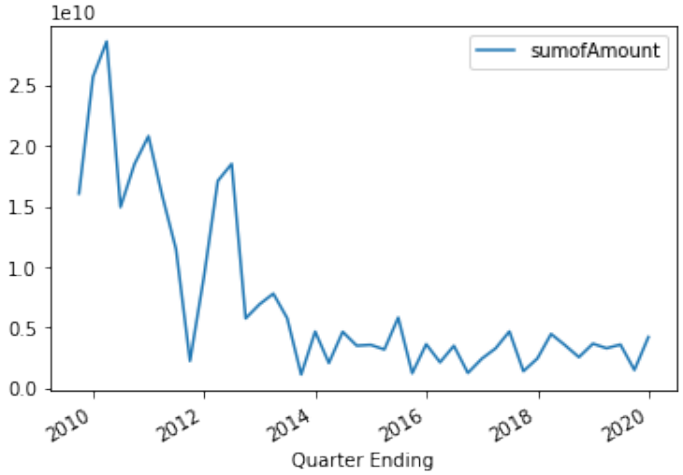


Figure 3.37: Big Data Visualization - Line Chart

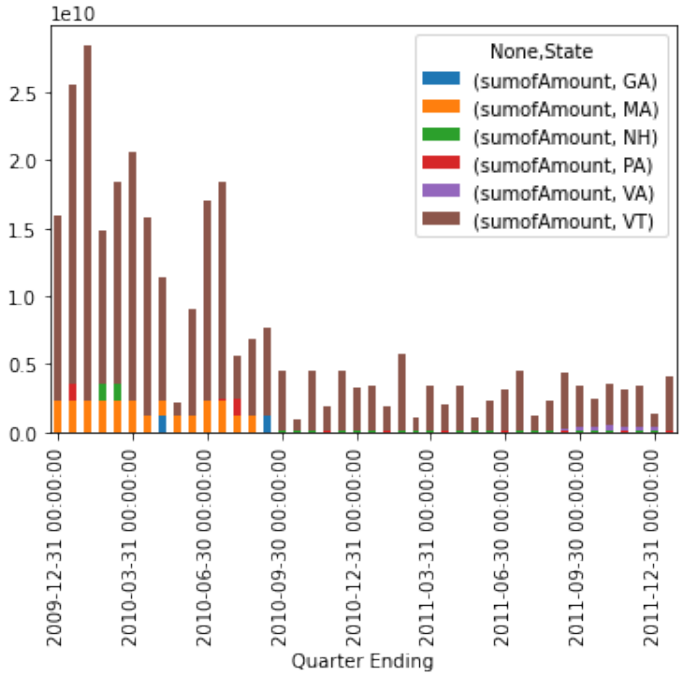


Figure 3.38: Big Data Visualization - Stacked Barplot

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.

3.2 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[22], 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 of Tidyverse[23] and Shiny-App[24] 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[25] reveals a strong increasing popularity for Streamlit):

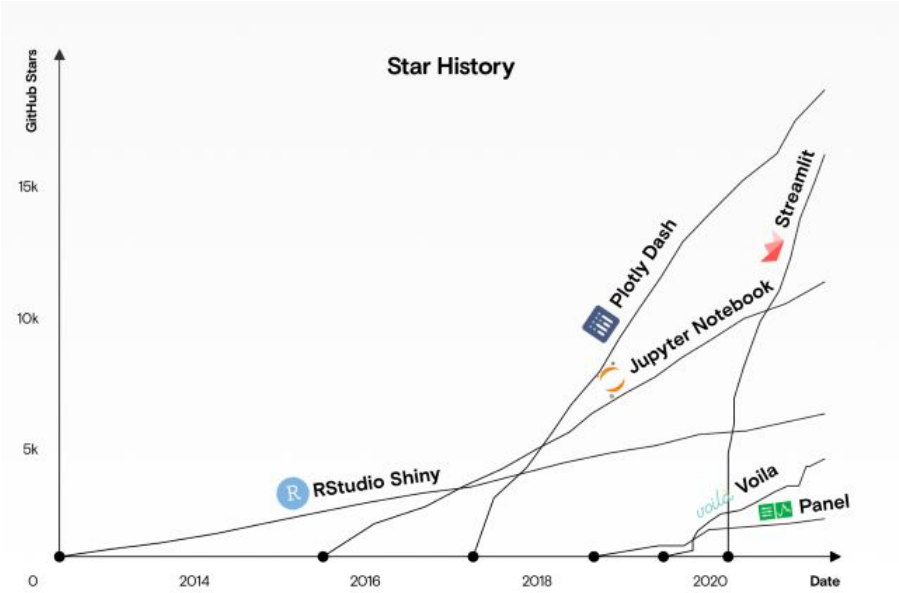


Figure 3.39: Data App - Comparison of Streamlit, RStudio-Shiny and others, Quelle: DataRevenue-Blog

I tested Streamlit[26] 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)

[22] Streamlit, https://www.streamlit.io/
[23] Tidyverse, https://www.tidyverse.org/
[24] Shiny-App, https://shiny.rstudio.com/
[25] Data-Revenue, https://www.datarevenue.com/de-blog/streamlit-vs-dash-vs-shiny-vs-voila-vs-flask-vs-jupyter
[26] Streamlit, https://www.streamlit.io/

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"[27] 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.

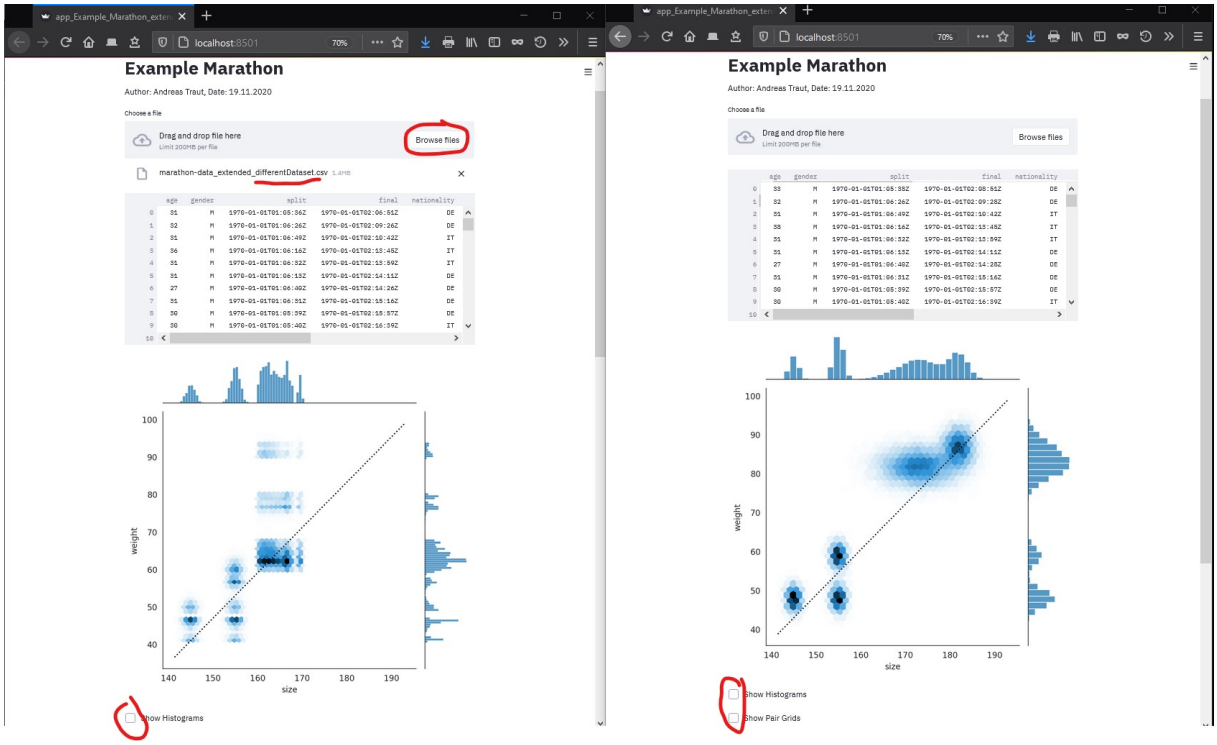


Figure 3.40: Data App - Marathon Example

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[28] and implemented a Streamlit app[29] in less than half an hour. This is why I love Streamlit: highly efficient, lovely design and easy to deploy.

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

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

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

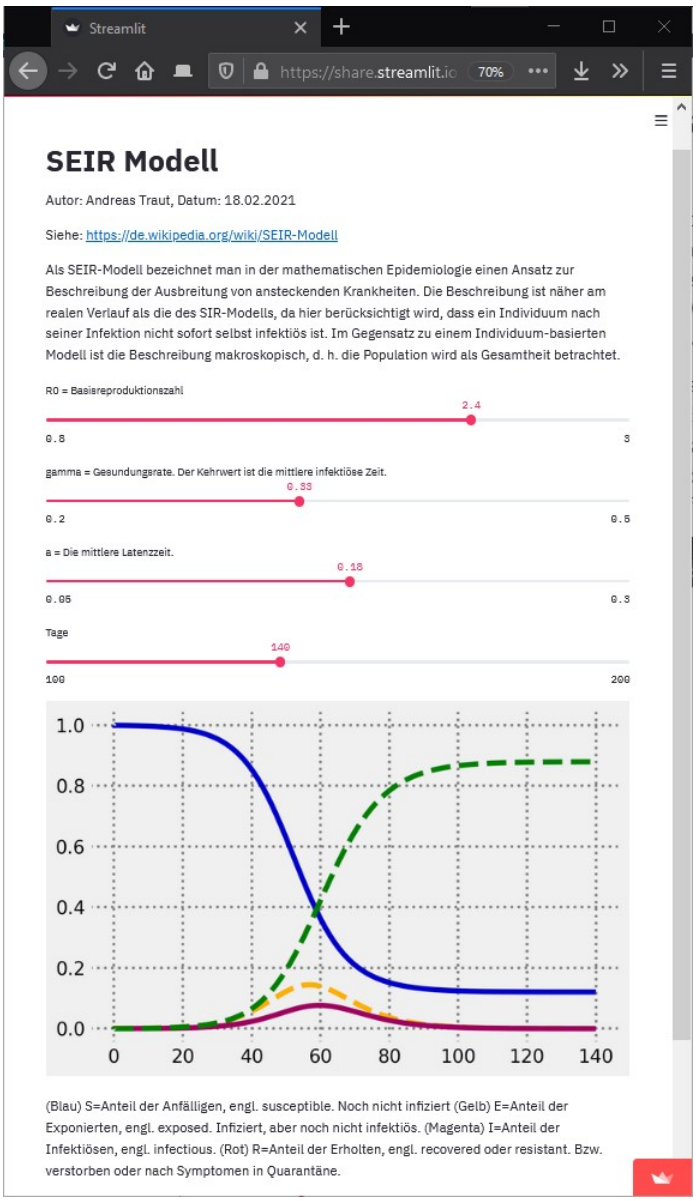


Figure 3.41: Data App - SEIR Model Example

3.3 Professional 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. Here are some examples and I recommend to read the documentation on their websites for getting a feeling about what these tools can do and maybe also their limitations.

3.3.1 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.

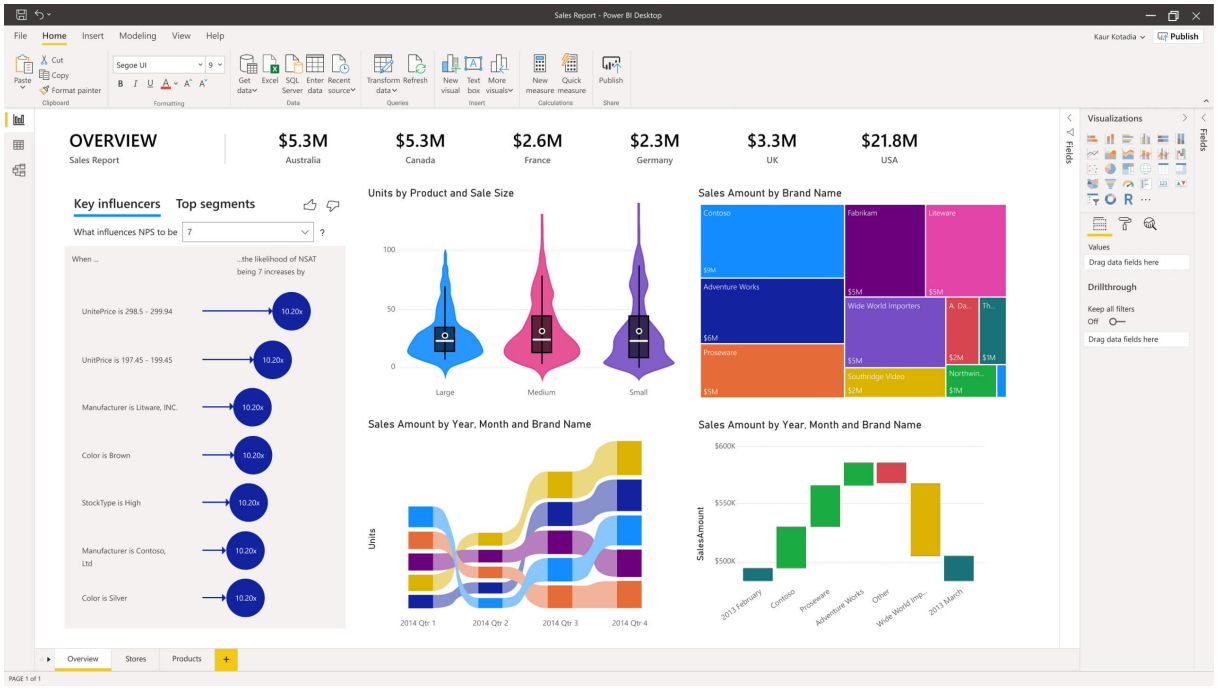


Figure 3.42: Professional BI - Power BI

Figure 3.43: Professional BI - Power BI Prices

3.3.2 Tableau

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

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

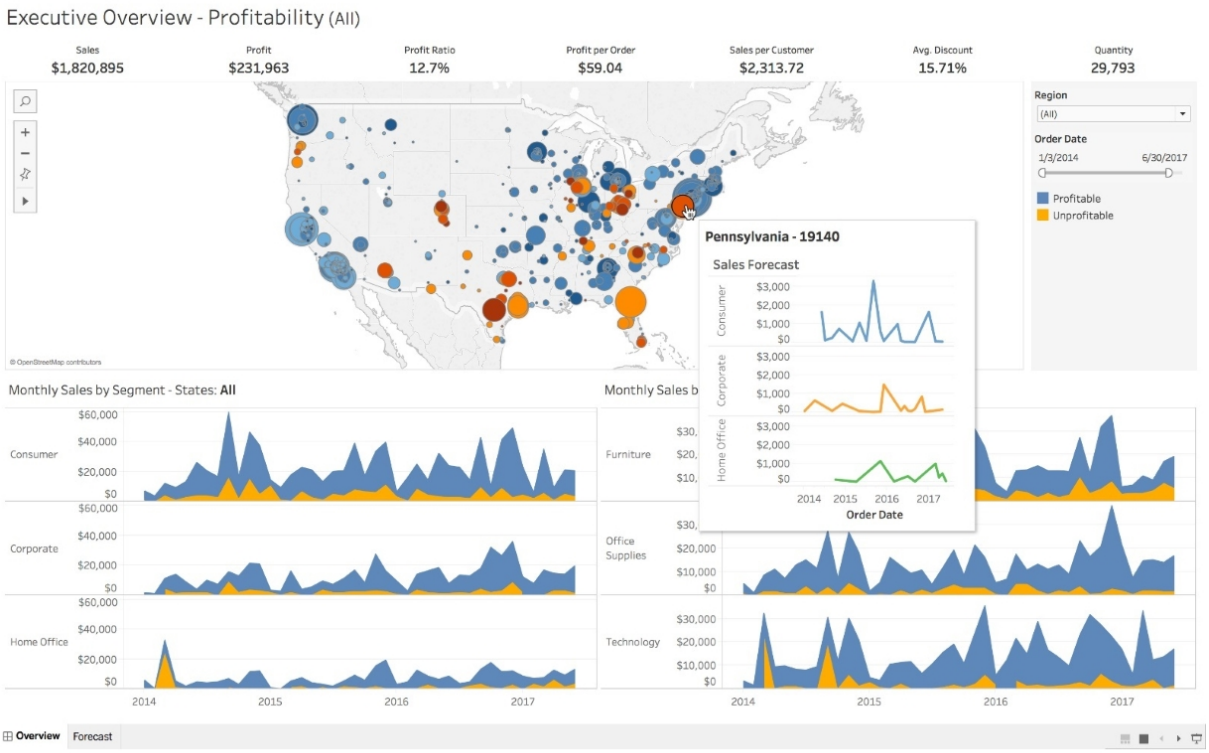


Figure 3.44: Professional BI - Tableau 1

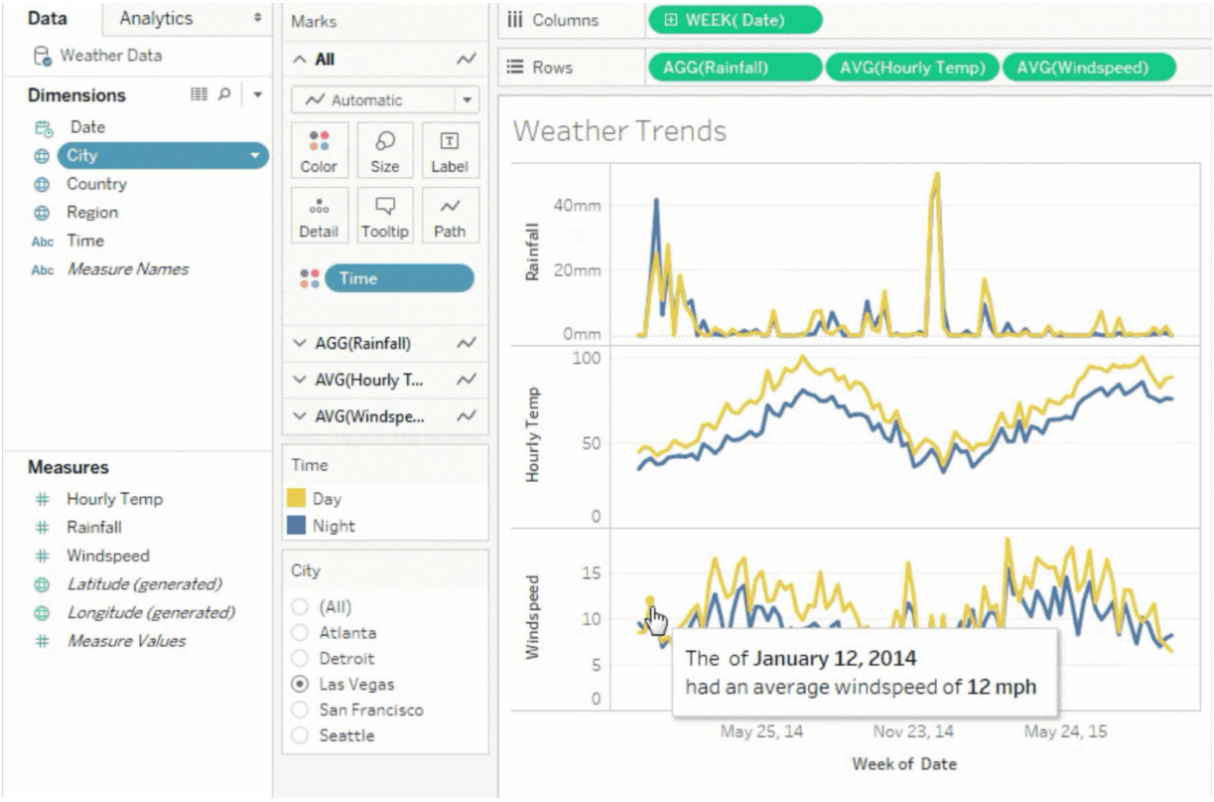


Figure 3.45: Professional BI - Tableau 2

Figure 3.46: Professional BI - Tableau Prices

3.3.3 QLink

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

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

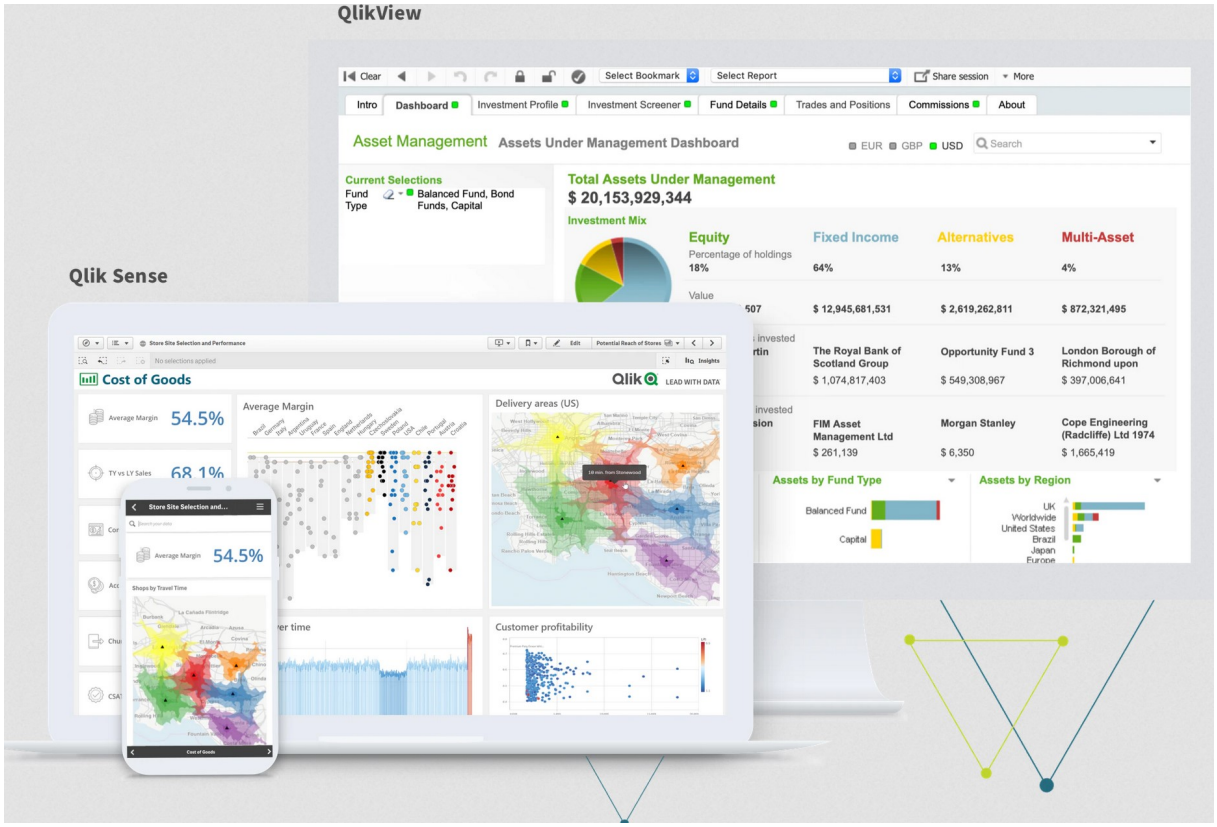


Figure 3.47: Professional BI - QLink

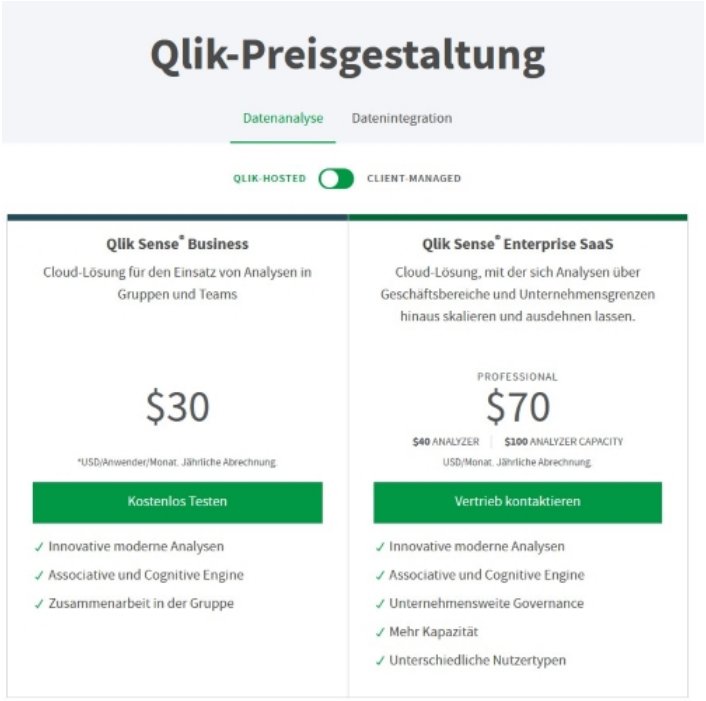


Figure 3.48: Professional BI - QLink Prices

4 Machine Learning with Python

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

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

[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

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	1980207	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	1760209	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:

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

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

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
```

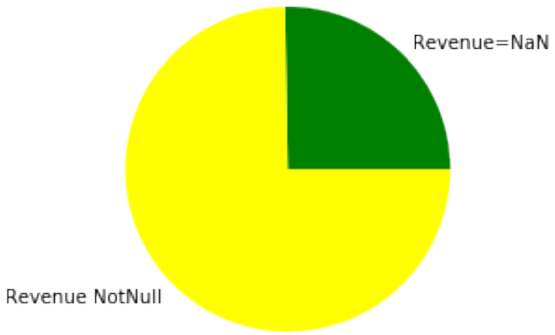


Figure 4.2: Movies Database - Plot Null and NotNull

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()
```

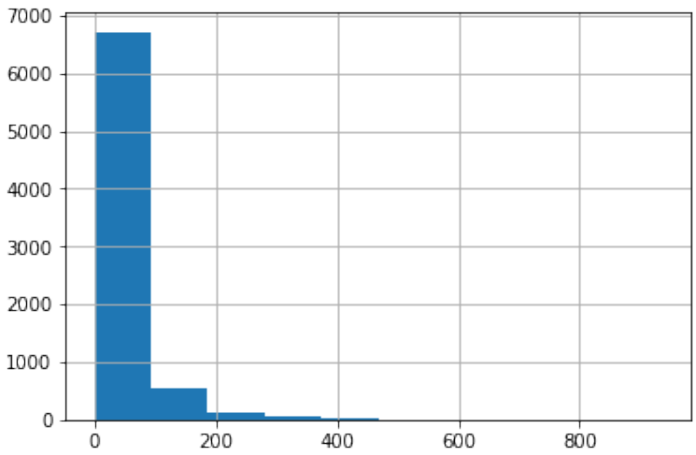


Figure 4.3: Movies Database - Histogram

```
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', color='gray')
fig.savefig('images/movies/movies_revenue_year_score.png')
```

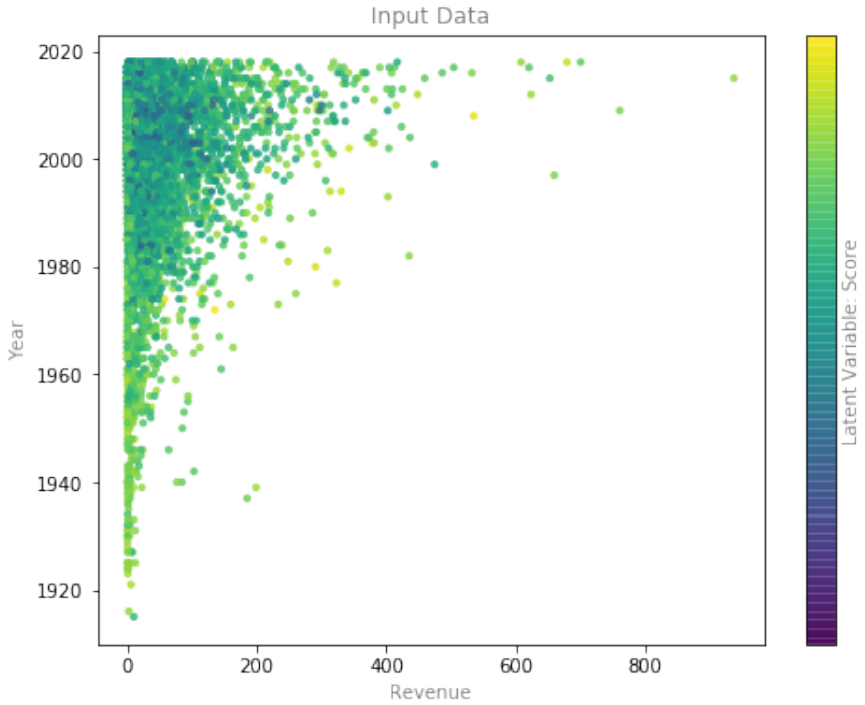


Figure 4.4: Movies Database - Scatterplot Revenue Year

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

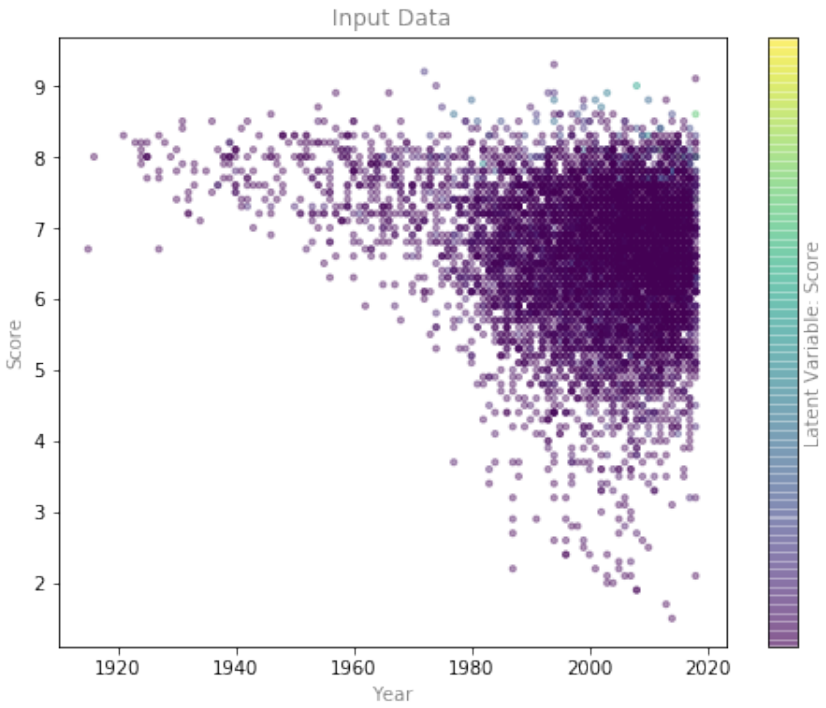


Figure 4.5: Movies Database - Scatterplot Year Score

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)`:

```
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 on the stratified sample by dividing `len(strat_test_set)`. The expectation is, that these are the same as above (otherwise we would have an error somewhere):

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

```
1    0.903653
2    0.069851
3    0.016058
4    0.010438
Name: RevCat, dtype: float64
```

Looks good so far: the numbers look very similar to the ones above. 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
```

Movies Database - Stratified Sample (Train, Test, NaNs)

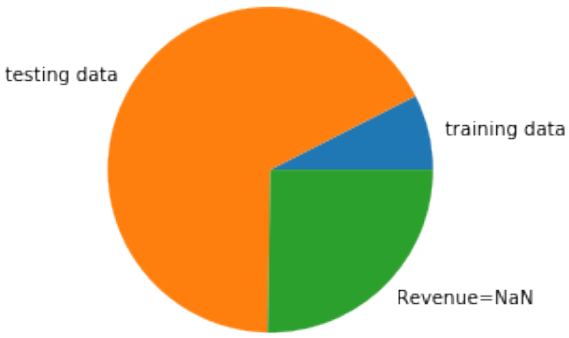


Figure 4.6: Movies Database - Testing vs Trainingdata vs NaN

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	19806.7		NaN	Action, Sci-Fi	18517	Don Taylor	103	A modern aircraft carrier is thrown back in ti…
59145915	The Chase	19945.8		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)
trainrmse
```

```
0.0
```

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
31233124	Victor Frankenstein	2015	6.0	36.0	Drama, Horror, Sci-Fi	45259	Paul McGuigan	110	Told from Igor's perspective, we see the troub…
30613062	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)
lin_rmse
```

```
54.68522284077671
```

```
movies_test_labels.mean()
```

```
36.2082738626228
```

```
movies_test_labels.std()
```

```
60.602745133524195
```

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)
```

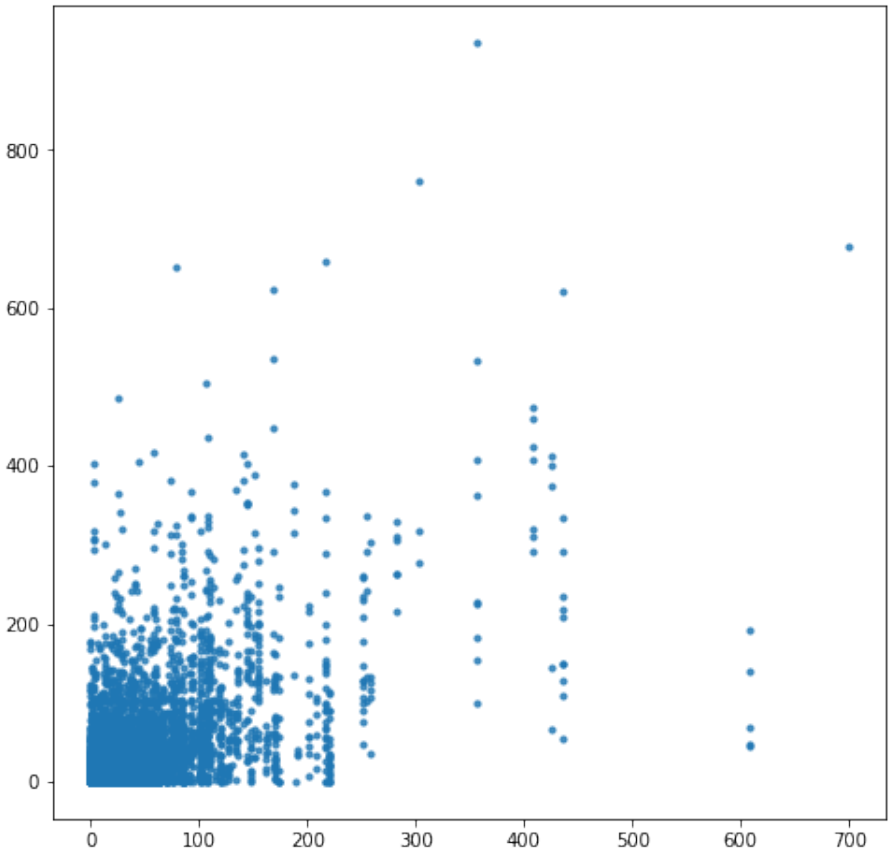


Figure 4.7: Movies Database - Plot True vs Predicted Value Side-by-Side Testdataset

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)
```

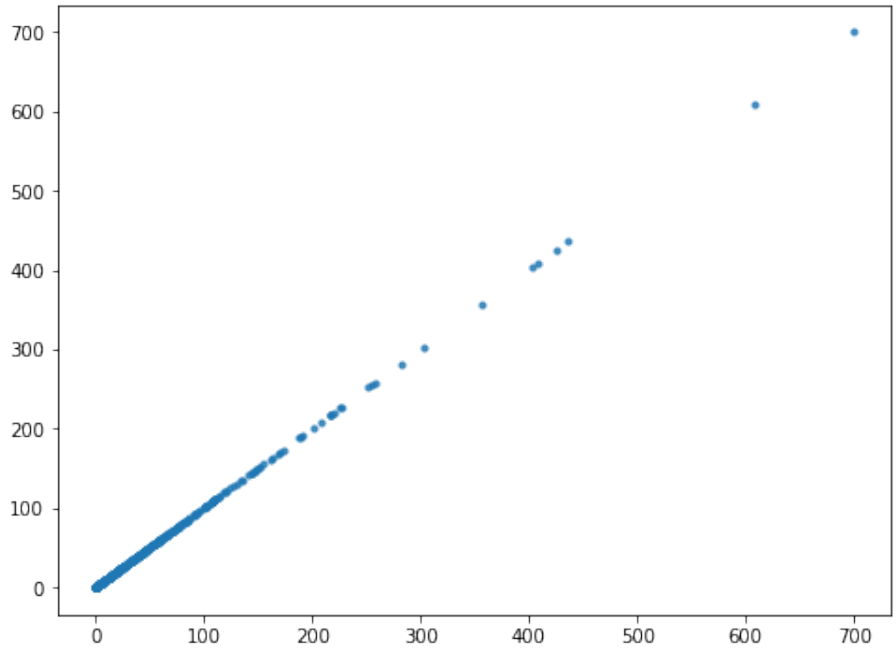


Figure 4.8: Movies Database - Plot True vs Predicted Value Side-by-Side Trainingdataset

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. **??** 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. **??**).

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*

[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/

train model”. After that we will do the *cross-validation*, *save the 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.

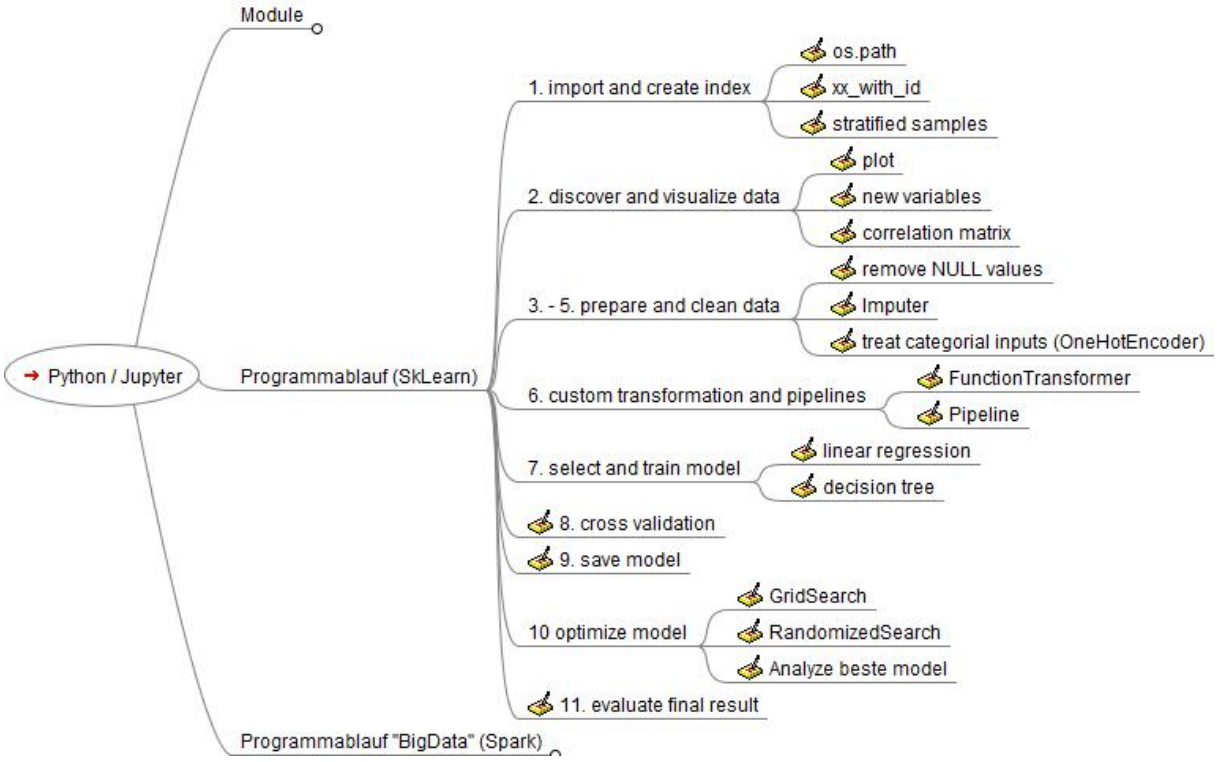


Figure 4.9: Mind Map - Scikit-Learn “Small Data”

```
#%% ######################################################################
# 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. 4.25) in order to learn from each of this two approaches. Finally I will put these two Mind Maps into one big (see fig. 4.30) 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
```

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

```
from six.moves import urllib
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

```
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")
```

```
158  # 1.1 Alternative 1: generate id with static data
159  housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
160  train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
161
```

Figure 4.10: Create Index - Alternative 1: Generate ID with Static Data

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)
```

```
173  # from sklearn.model_selection import StratifiedShuffleSplit
174  split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
175  for train_index, test_index in split.split(housing, housing["income_cat"]):
176      strat_train_set = housing.loc[train_index]
177      strat_test_set = housing.loc[test_index]
```

Figure 4.11: Create Index - Alternative 2: Generate Stratified Sampling

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. **??**. 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()])
```

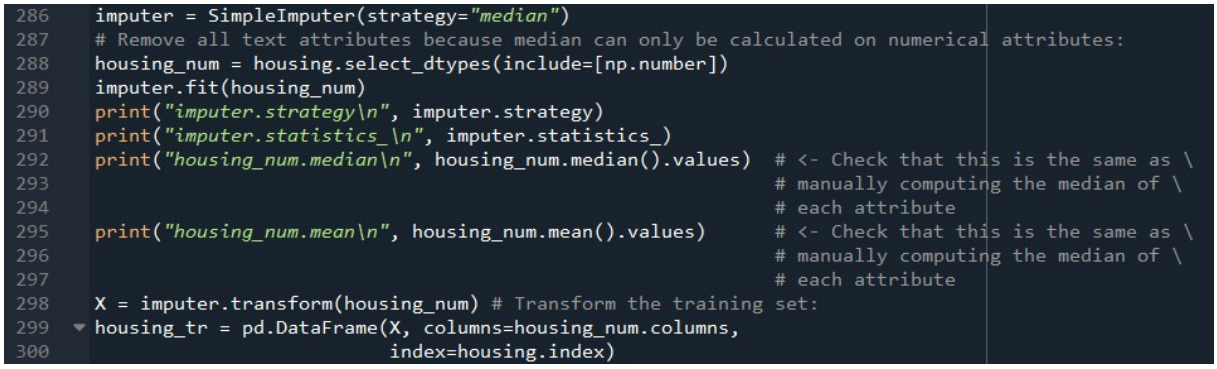
4.2.3.2 Remove all NULL-Values (3.2)

```
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)
```

```
266    sample_incomplete_rows = housing[housing.isnull().any(axis=1)].head()
267    # sample_incomplete_rows.dropna(subset=["total_bedrooms"])    # option 1
268    # sample_incomplete_rows.drop("total_bedrooms", axis=1)       # option 2
269
270    median = housing["total_bedrooms"].median()
271    sample_incomplete_rows["total_bedrooms"].fillna(median, inplace=True) # option 3
```

4.2.4 Use “Imputer” to Clean NaNs (4)

```
imputer = SimpleImputer(strategy="median")
# Remove all text attributes because median can only be calculated on
 ↪  numerical attributes:
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:
X = imputer.transform(housing_num) # Transform the training set:
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing.index)
housing_tr.loc[sample_incomplete_rows.index.values]
```



```
imputer = SimpleImputer(strategy="median")
# Remove all text attributes because median can only be calculated on numerical attributes:
housing_num = housing.select_dtypes(include=[np.number])
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
X = imputer.transform(housing_num) # Transform the training set:
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing.index)
```

4.2.5 Treat “Categorial” Inputs (5)

```
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]

# from sklearn.preprocessing import FunctionTransformer

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())
```

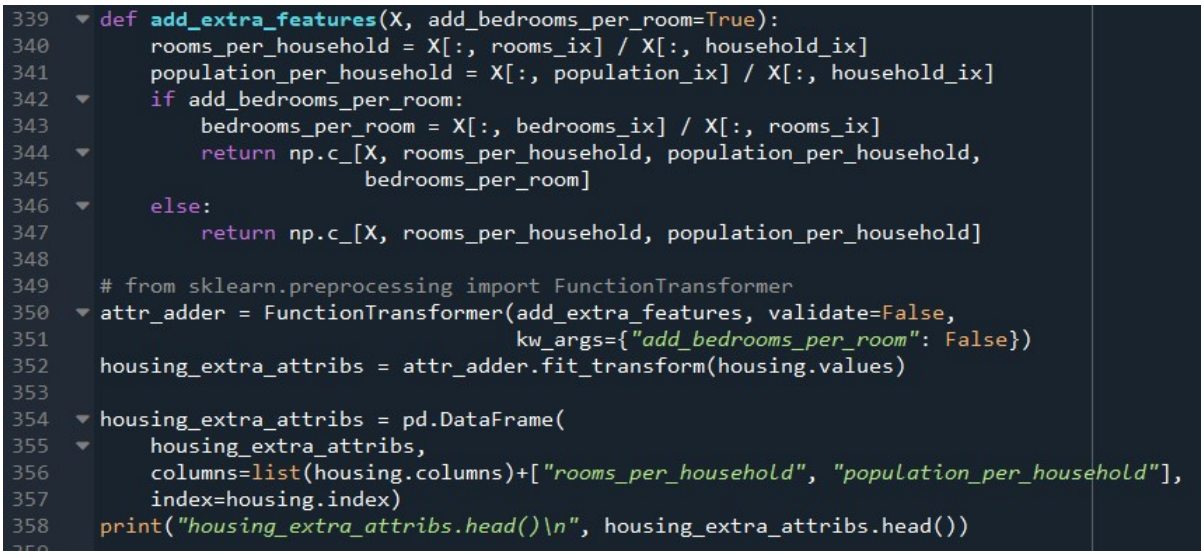


Figure 4.12: Small Data - Custom Transformer (6.1)

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)
```

```
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)
```

Figure 4.13: Small Data - Pipelines (6.2)

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)
```

```
403  # from sklearn.linear_model import LinearRegression
404  lin_reg = LinearRegression()
405  lin_reg.fit(housing_prepared, housing_labels)
406  # let's try the full preprocessing pipeline on a few training instances
407  some_data = housing.iloc[:1]
408  some_labels = housing_labels.iloc[:1]
409  some_data_prepared = full_pipeline.transform(some_data)
410  print("Predictions:\n", lin_reg.predict(some_data_prepared))
411  print("Labels:\n", list(some_labels)) # Compare against the actual values:
412
413  # from sklearn.metrics import mean_squared_error
414  housing_predictions = lin_reg.predict(housing_prepared)
415  lin_mse = mean_squared_error(housing_labels, housing_predictions)
416  lin_rmse = np.sqrt(lin_mse)
417  print("lin_rmse\n", lin_rmse)
```

Figure 4.14: Small Data - LinearRegression Model (7.1)

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)
```

```
421  # from sklearn.tree import DecisionTreeRegressor
422  tree_reg = DecisionTreeRegressor(random_state=42)
423  tree_reg.fit(housing_prepared, housing_labels)
424  housing_predictions = tree_reg.predict(housing_prepared)
425
426  tree_mse = mean_squared_error(housing_labels, housing_predictions)
427  tree_rmse = np.sqrt(tree_mse)
428  print("tree_rmse\n", tree_rmse)
```

Figure 4.15: Small Data - DecisionTreeRegressor Model (7.2)

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())
```

```
440    # from sklearn.model_selection import cross_val_score
441    scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
442                             scoring="neg_mean_squared_error", cv=10)
```

Figure 4.16: Small Data - Cross-Validation for DecisionTreeRegressor (8.1)

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)
```

```
454    lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
455                                 scoring="neg_mean_squared_error", cv=10)
```

Figure 4.17: Small Data - Cross-Validation for LinearRegression (8.2)

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)
```

```
463   # from sklearn.ensemble import RandomForestRegressor
464   forest_reg = RandomForestRegressor(n_estimators=10, random_state=42)
465   forest_reg.fit(housing_prepared, housing_labels)
466
467   housing_predictions = forest_reg.predict(housing_prepared)
468   forest_mse = mean_squared_error(housing_labels, housing_predictions)
469   forest_rmse = np.sqrt(forest_mse)
470   print(forest_rmse)
471   # from sklearn.model_selection import cross_val_score
472   forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
473                                   scoring="neg_mean_squared_error", cv=10)
474   forest_rmse_scores = np.sqrt(-forest_scores)
475   display_scores(forest_rmse_scores)
```

Figure 4.18: Small Data - Cross-Validation for RandomForestRegressor (8.3)

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)
```

```
481   from sklearn.ensemble import ExtraTreesRegressor
482   extratree_reg = ExtraTreesRegressor(n_estimators=10, random_state=42)
483   extratree_reg.fit(housing_prepared, housing_labels)
484
485   housing_predictions = extratree_reg.predict(housing_prepared)
486   extratree_mse = mean_squared_error(housing_labels, housing_predictions)
487   extratree_rmse = np.sqrt(extratree_mse)
488   print(extratree_rmse)
489   extratree_scores = cross_val_score(extratree_reg, housing_prepared,
490                                      housing_labels,
491                                      scoring = "neg_mean_squared_error", cv=10)
492   extratree_rmse_scores = np.sqrt(-extratree_scores)
493   display_scores(extratree_rmse_scores)
```

Figure 4.19: Small Data - Cross-Validation for ExtraTreesRegressor (8.4)

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)
```

```
# from sklearn.model_selection import GridSearchCV
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)
```

Figure 4.20: Small Data - Optimize Model GridSearchCV on RandomForestRegressor (10.1.1)

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 4.21: 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)
```

```
# from sklearn.model_selection import RandomizedSearchCV
# from scipy.stats import randint
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 4.22: 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 4.23: 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)))
      )
```

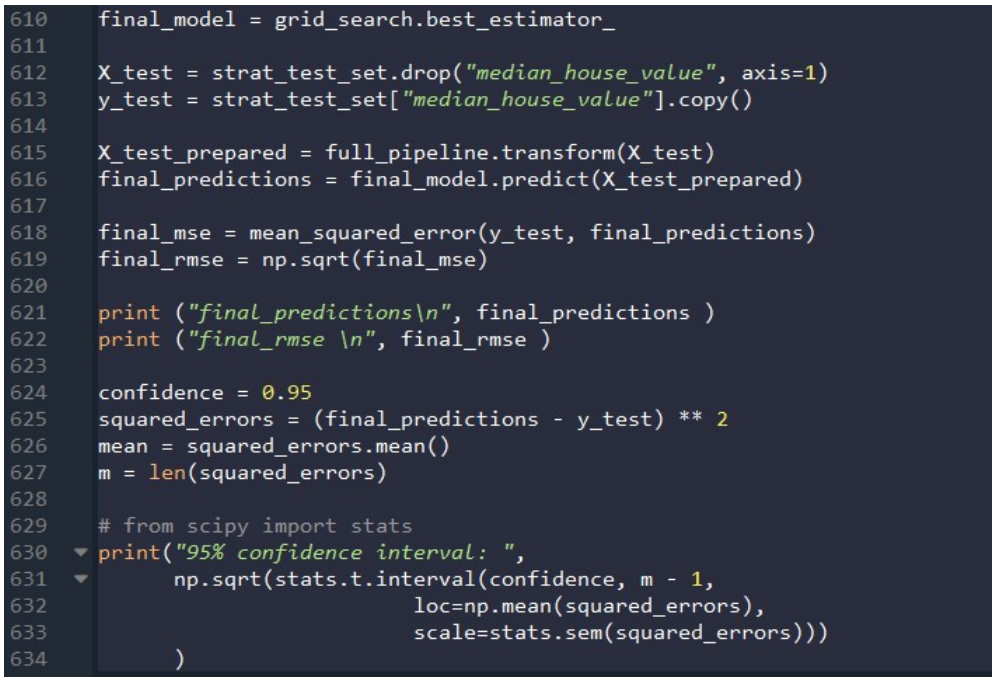


Figure 4.24: Small Data - Evaluate final model on test dataset (11)

4.3 “Big Data” Machine Learning using Spark ML Library

This will be an example for a “Big-Data” environment and uses the “Apache MLib” scalable machine learning library. Various tutorials, documentation, “code-fragments” and guidelines can be found in the internet **for free** (at least for your private use). The best is in my opinion the official documentation. A few more helpful sources are the following Github repositories:

- tirthajyoti/Spark-with-Python (MIT licence)
- Apress/learn-pyspark (Freeware License)
- mahmoudparsian/pyspark-tutorial (Apache License v2.0)

Concerning the topic **“Big Data”** 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 differenciate “standard” machine learning from a “scalable” environment is not easy. I recommend a separate training! Some steps are a bit similar to “scikit-learn” (e.g. data-cleaning, preprocessing), but the technical environment for running the code is different and also the code itself is different.

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

Let’s start with the structure, which I put into a mind map (you can download it from this repository). I aligned the structure to the SkLearn mind map above in order to learn from each of this two approaches.

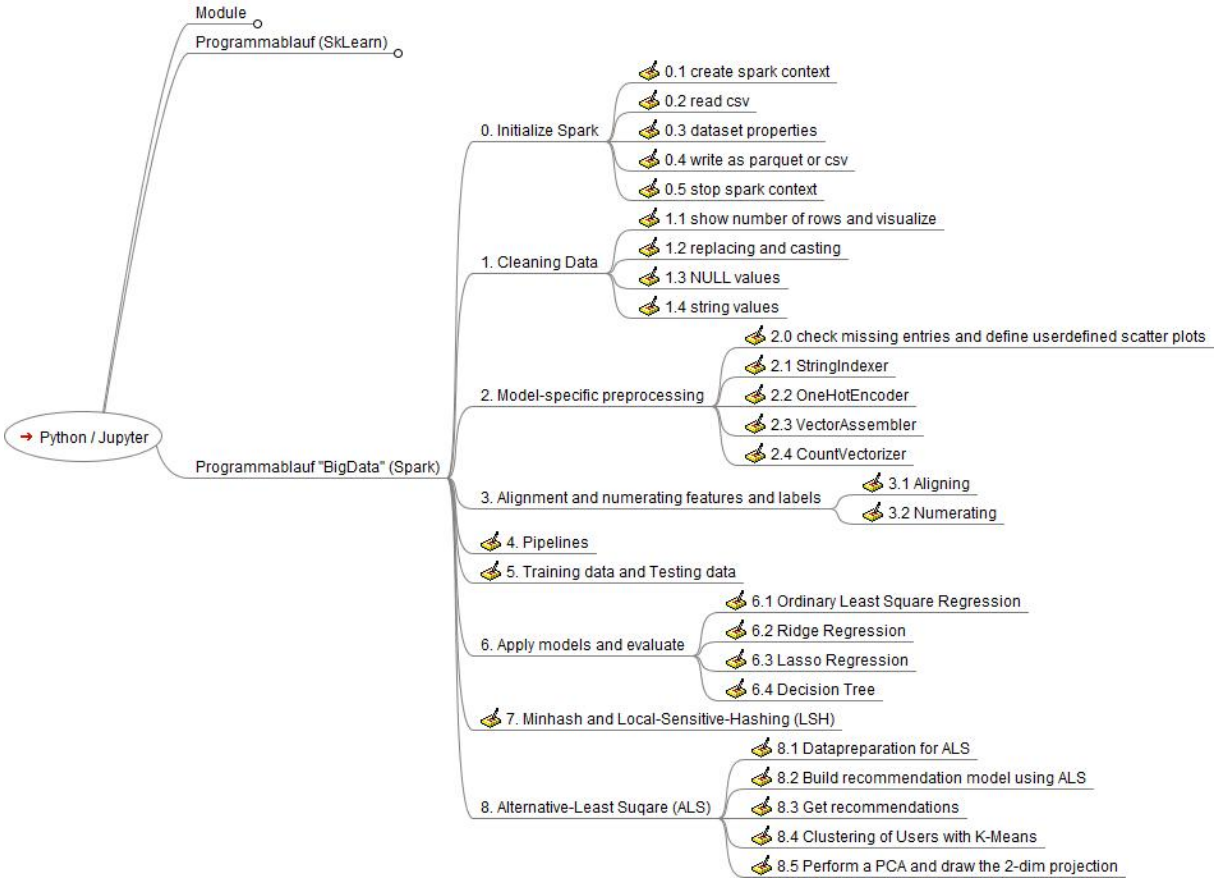


Figure 4.25: Mind Map - Apache Spark ML “Big Data”

There are different ways to approach the Apache Spark and Hadoop environment: you can install it on your own computer (which I found rather difficult because of lack of userfriendly and easy understandable documentation). Or you can dive into a Cloud environment, like e.g. Microsoft Azure or Amazon EWS or Google Cloud and try to get a virtual machine up and running for your purposes. Have a look at my documentation, where I shared my experiences, which I had with Microsoft Azure here.

For the following explanation I decided to use Docker. 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.”* Learn from the Docker-Curriculum how it works. I found an 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 following repository here. After having installed the Docker application you will need to pull my “machine-learning-pyspark” image to your computer:

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

Then open Windows Powershell and type the following:

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

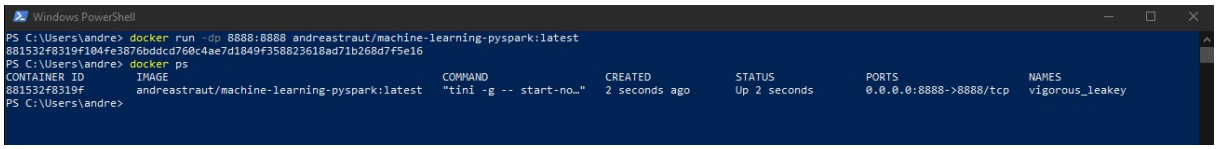


Figure 4.26: Big Data - Run Docker

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

Figure 4.27: Big Data - Docker Dashbord

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

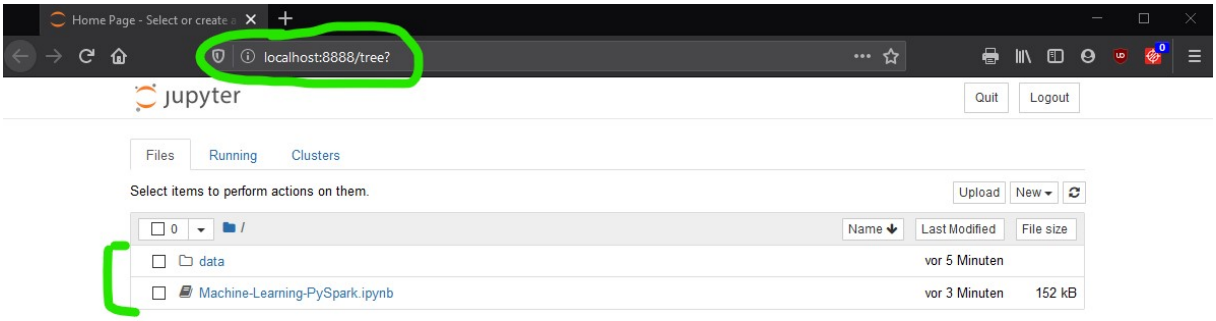
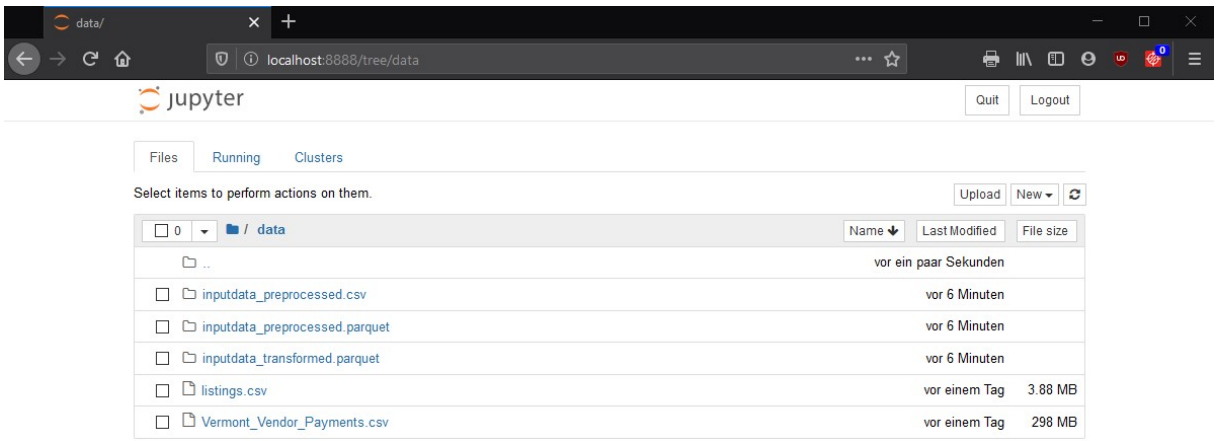


Figure 4.28: Big Data - Docker Localhost

The folder “data” contains the datasets. If you would like to do further analysis or produce alternate visualisations of the Airbnb-data, you can download them from here. It is available below under a Creative Commons CC0 1.0 Universal (CC0 1.0) “Public Domain Dedication” license. The data for the Vermont-Vendor-Payments can be downloaded from here and are available under the Open Data Commons Open Database License.



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

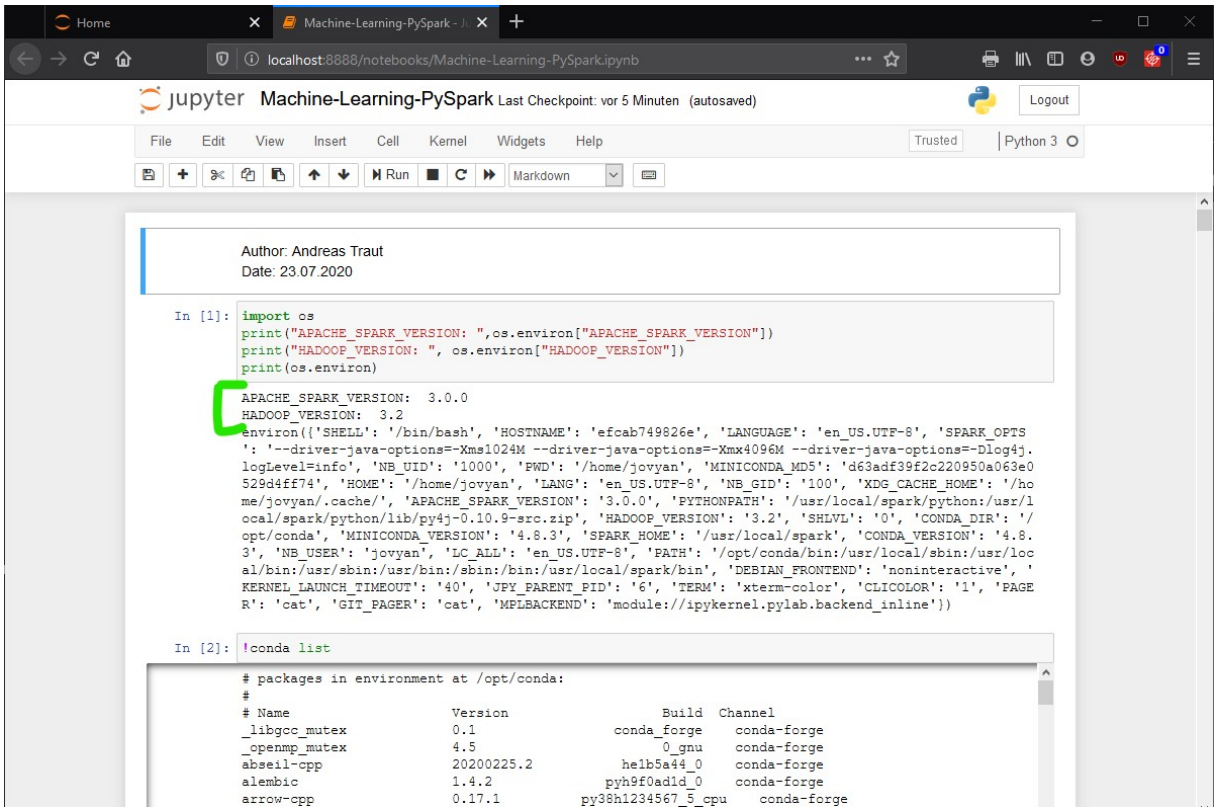


Figure 4.29: Big Data - Docker Jupyter Notebook

4.3.0.1 Initialize Spark

Initializing a Spark sessions works and reading a CSV file can by done with the following commands (see more documentation here and also have a look at a "Get Started Guide"):

```
In [3]: import pyspark
        from pyspark.sql import SparkSession
        from pyspark.sql import functions as F
```

4.3.0.1.1 Create Spark Context and Spark Session

```
In [4]: sc = pyspark.SparkContext(appName='Spark Modelling Context')
```

```
In [5]: spark = SparkSession.builder \
                .appName('Spark Modelling Session') \
                .config('spark.executor.memory','5g') \
                .config('spark.executor.cores','4') \
                .getOrCreate()
```

4.3.0.1.2 Read CSV

```
In [6]: 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)
```

4.3.0.1.3 Dataset Properties and some Select, Group and Aggregate Methods After then the data-cleaning and data preparation (eliminating of null values, visualization techniques) work pretty similar to the "Small data" (Sklearn) approach.

4.3.0.1.4 Write as Parquet or CSV If you want to persist (=save) your intermediate you can do it as follows:

Persisting the preprocessed data

In [22]:
```
data.select(*data.columns[:]).write.format("parquet") \
.save("data/inputdata_preprocessed.parquet", mode='overwrite')

data.select(*data.columns[:]).write.csv('data/inputdata_preprocessed.csv', mode='overwrite', header=
```

In [23]:
```
filename = "data/inputdata_preprocessed.parquet"
data = spark.read.parquet(filename)
data.show(5)
```

```
+----+--------------------+-------+----------+--------------------+--------------------+--------+-
--------+---------------+-----+--------------+-----------------+-----------+-----------------+----
-------------------------+----------------+
|  id|                name|host_id| host_name| neighbourhood_group|       neighbourhood|latitude|l
ongitude|      room_type|price|minimum_nights|number_of_reviews|last_review|reviews_per_month|calc
ulated_host_listings_count|availability_365|
+----+--------------------+-------+----------+--------------------+--------------------+--------+-
--------+---------------+-----+--------------+-----------------+-----------+-----------------+----
-------------------------+----------------+
|3176|Fabulous Flat in ...|   3718|    Britta|              Pankow|Prenzlauer Berg S...|  52.535|
13.41758|Entire home/apt| 90.0|            62|              145| 2019-06-27|             1.11|
1.0|             140|
|3309|BerlinSpot Schöne...|   4108|      Jana|Tempelhof - Schön...|     Schöneberg-Nord|52.49885|
13.34906|   Private room| 28.0|             7|               27| 2019-05-31|             0.34|
1.0|             320|
|6883|Stylish East Side...|  16149|   Steffen|Friedrichshain-Kr...|Frankfurter Allee...|52.51171|
13.45477|Entire home/apt|125.0|             3|              133| 2020-02-16|             1.08|
1.0|               0|
|7071|BrightRoom with s...|  17391|BrightRoom|              Pankow|      Helmholtzplatz|52.54316|
13.41509|   Private room| 33.0|             1|              292| 2020-03-06|             2.27|
2.0|              45|
|9991|Geourgeous flat -...|  33852|   Philipp|              Pankow|Prenzlauer Berg S...|52.53303|
13.41605|Entire home/apt|180.0|             6|                8| 2020-01-04|             0.14|
1.0|               8|
+----+--------------------+-------+----------+--------------------+--------------------+--------+-
--------+---------------+-----+--------------+-----------------+-----------+-----------------+----
-------------------------+----------------+
only showing top 5 rows
```

4.3.0.1.5 Read Parquet

See jupyter notebook.

4.3.0.1.6 How to stop a Spark Session and Spark Context

See jupyter notebook.

4.3.0.2 Cleaning the data

4.3.0.2.1 Show number of rows and columns and do some visualizations

4.3.0.2.2 Replacing and Casting

4.3.0.2.3 Null-Values

4.3.0.2.4 String Values

4.3.0.3 Model-specific preprocessing

4.3.0.3.1 Check missing entries and define userdefined scatter plot

4.3.0.3.2 StringIndexer

I included some examples of how features can be extracted, transformed and selected in the Jupyter-Notebook (see more documentation here). Just to mention a few here: the "StringIndexer", "OneHotEncoder" and "VectorAssembler" work as follows:

```
In [25]: data.select('neighbourhood_group').distinct().count()
Out[25]: 38

In [26]: from pyspark.ml.feature import StringIndexer
         neighbourhood_indexer = StringIndexer(inputCol='neighbourhood_group', outputCol='neighbourhood_group_
         neighbourhood_indexer_model = neighbourhood_indexer.fit(data)
         data = neighbourhood_indexer_model.transform(data)

In [27]: data.groupby('neighbourhood_group').agg(F.collect_set('neighbourhood_group_index').alias('neighbourh
```

```
+--------------------+------------------------+
| neighbourhood_group|neighbourhood_group_index|
+--------------------+------------------------+
|Friedrichshain-Kr...|                   [0.0]|
|               Mitte|                   [1.0]|
|              Pankow|                   [2.0]|
|           Neukölln|                   [3.0]|
|Charlottenburg-Wilm.|                   [4.0]|
|Tempelhof - Schön...|                   [5.0]|
|         Lichtenberg|                   [6.0]|
|  Treptow - Köpenick|                   [7.0]|
|Steglitz - Zehlen...|                   [8.0]|
|       Reinickendorf|                   [9.0]|
|Marzahn - Hellers...|                  [10.0]|
|             Spandau|                  [11.0]|
| Downtown Apartments|                  [12.0]|
|Downtown Apartmen...|                  [13.0]|
|     Neue Kantstraße|                  [14.0]|
|      Alexanderplatz|                  [15.0]|
|Prenzlauer Berg Nord|                  [16.0]|
|     Alt-Lichtenberg|                  [17.0]|
|           Barstraße|                  [18.0]|
|Blankenfelde/Nied...|                  [19.0]|
|    Brunnenstr. Nord|                  [20.0]|
|Frankfurter Allee...|                  [21.0]|
|           Grunewald|                  [22.0]|
|      Kurfürstendamm|                  [23.0]|
|                Lisa|                  [24.0]|
```

4.3.0.3.3 OntHotEncoder

```
In [28]: from pyspark.ml.feature import OneHotEncoder
         one_hot_encoder = OneHotEncoder(
             inputCol = 'neighbourhood_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)
```

```
In [29]: from pyspark.ml.feature import VectorAssembler
         data = data.withColumn('number_of_reviews', data['number_of_reviews'].cast('double'))
         data.select('number_of_reviews').show()
```

```
+-----------------+
|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|
|            188.0|
|             31.0|
|             74.0|
|            296.0|
|             39.0|
+-----------------+
only showing top 20 rows
```

4.3.0.3.4 VectorAssembler

4.3.0.3.5 CountVectorizer

4.3.0.4 Aligning and numerating Features and Labels

4.3.0.4.1 Aligning

4.3.0.4.2 Numerating

4.3.0.5 Pipelines

4.3.0.6 Training data and Testing data

4.3.0.7 Apply models and evaluate

4.3.0.7.1 Ordinary Least Square Regression After having extracted, transformed and selected features you will want to apply some models, which are documented here, for example the "OLS Regression":

```
In [38]: from pyspark.ml.regression import LinearRegression
         lr = LinearRegression(featuresCol='num_features', labelCol='label',maxIter=1000, fitIntercept=True)

In [39]: lr_model = lr.fit(data_train)
         lr_model.coefficients

Out[39]: DenseVector([-0.4689])

In [40]: pred = lr_model.transform(data_test)
```

4.3.0.7.2 Ridge Regression

4.3.0.7.3 Lasso Regression

4.3.0.7.4 Decision Tree

4.3.0.8 Minhash und Local-Sensitive-Hashing (LSH)

see example: https://github.com/AndreasTraut/Deep_learning_explorations

4.3.0.9 Alternative-Least-Square (ALS)

4.3.0.9.1 Datapreparation for ALS

4.3.0.9.2 Build the recommendation model using alternating least squares (ALS)

4.3.0.9.3 Get recommendations

4.3.0.9.4 Clustering of Users with K-Means see example: https://hub.docker.com/repository/docker/andreastrau
learning-pyspark

4.3.0.9.5 Perform a PCA and draw the 2-dim projection

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:

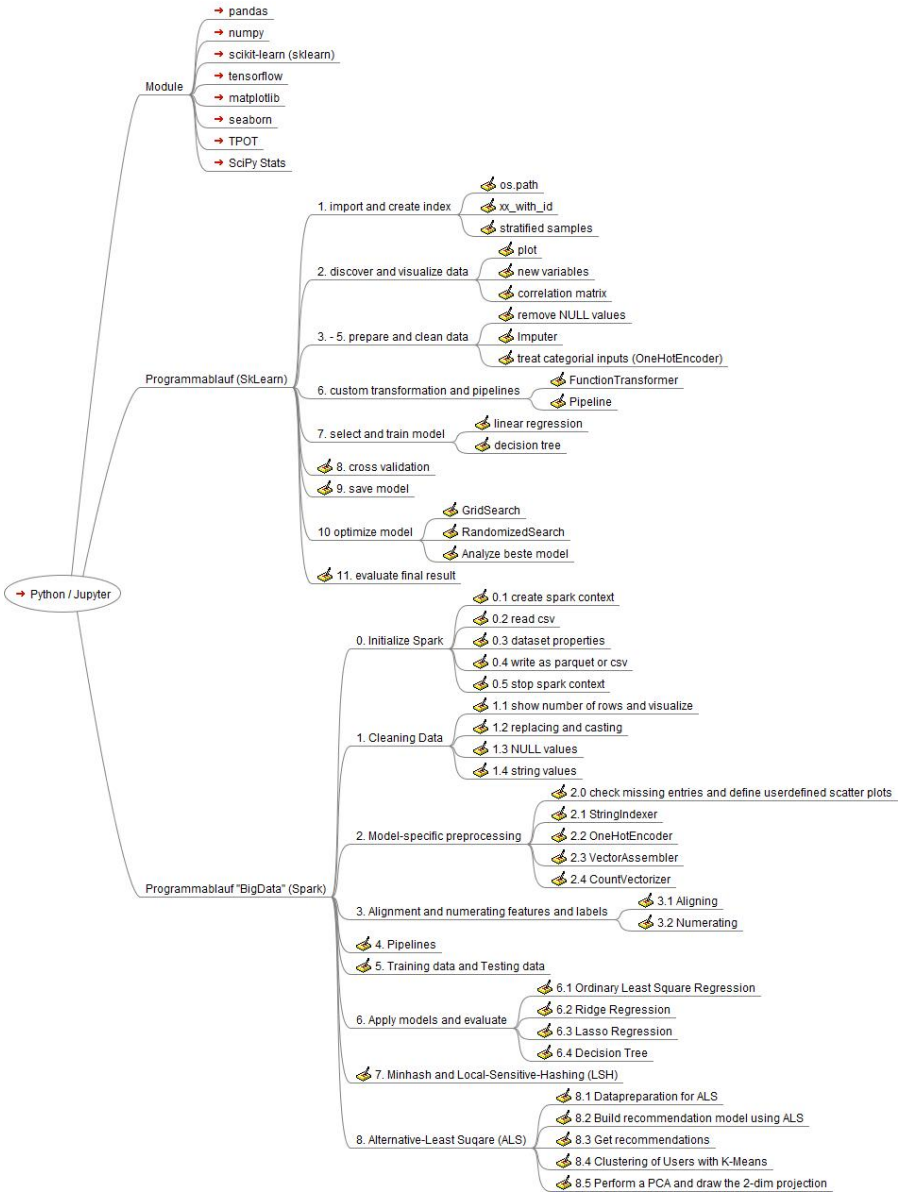


Figure 4.30: Mind Map - Scikit-Learn and Apache SparkML

4.5 Digression (Excurs) to Big Data Visualization and K-Means Clustering Algorithm and Map-Reduce

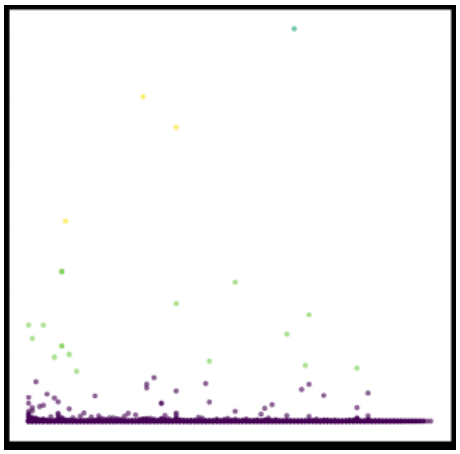
4.5.1 Big Data Visualization

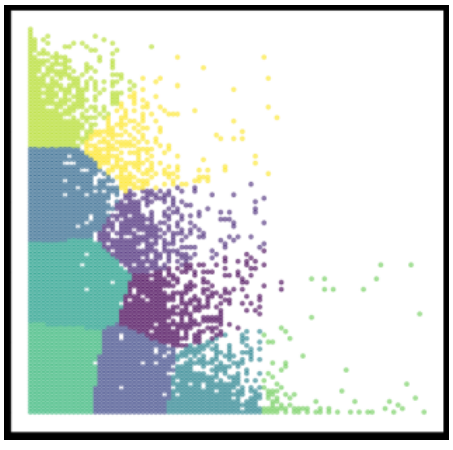
You will see a Jupyter-Notebook (which contains the Machine-Learning Code) and a folder named “data” (which contains the raw-data and preprocessed data). As you can see: I also worked on a 298 MB big csv-file (“Vermont_Vendor_Payments.csv”), which I couldn’t open in Excel, because of

the huge size. This file contains a list of all state of Vermont payments to vendors (Open Data Commons License) and has more than 1.6 million lines (exactly 1'648'466 lines). I already mentioned in my repository "Visualization-of-Data-with-Python", that the **visualization of big datasets** can be difficult when using "standard" office tools, like Excel. If you are not able to open such csv-files in Excel you have to find other solutions. One is to use PySpark which I will show you here. Another solution would have been to use the Excel built-in connection, PowerQuery or something similar, maybe Access or whatever, which is not the topic here, because we also want to be able to apply machine-learning algorithms from the Spark Machine Learning Library. And there are more benefits of using PySpark instead of Excel: it can handle distributed processing, it's a lot faster, you can use pipelines, it can read many file systems (not only csv), it can process real-time data.

4.5.2 K-Means Clustering Algorithm

Additionally I worked on this dataset to show how the K-Means Clustering Algorithm can be applied by using the Spark Marchine-Learning Libary (see more documentation here). I will show how the "Vermont Vendor Payments" dataset can be clustered. In the images below every color represents a differents cluster:





4.5.3 Map-Reduce

This is a programming model for generating big data sets with parallel distributed algorithm on a cluster. Map-Reduce is very important for Big Data and therefore I added some Jupyter-Notebooks to better understand how it works. Learn the basis of the *Map-Reduce* programming model from here and then have a look into my jupyter notebook for details. I used the very popular "Word Count" example in order to explain Map-Reduce in detail.

In another application of Map-Reduce I found the very popular **term frequency–inverse document frequency** (short **TF-idf**) very interesting (see Wikipedia). 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" 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 or to classify documents.

Have a look into my notebook here to learn more about Big Data Visualization, K-Means Clustering Algorithm, Map-Reduce and TF-idf.

4.6 Future learnings and coding & data sources

For all of these topics various tutorials, documentation, coding examples and guidelines can be found in the internet **for free**! The Open Source Community is an incredible treasure trove and enrichment that positively drives many digital developments: Scikit-Learn, Apache Spark, Spyder, GitHub, Tensorflow and also Firefox, Signal, Threema, Corona-Warnapp… to be mentionned. 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 documentation and a few more Github repositories like e.g. tirthajyoti/Spark-with-Python (MIT Licence), Apress/learn-pyspark (Freeware License), mahmoudparsian/pyspark-tutorial (Apache License v2.0). What I will do here in my repository is nothing more than putting it together so that it works for my problem (which can be challenging as well sometimes). Adapting it for your needs should be easier from this point on.

Data:

If you would like to do further analysis or produce alternate visualizations of the Airbnb-data, you can download them from here. It is available below under a Creative Commons 1.0 Universal "Public Domain Dedication" license. The data for the Vermont-Vendor-Payments can be downloaded from here and are available under the Open Data Commons Open Database License. The movies database doesn't even mention a license and is from Kaggle. There you find a lot of more datasets and also coding examples for your studies.

5 Use cases of artificial intelligence in industry

In this text I would like to explain in an easily understandable way what is *meant by "artificial intelligence in industry" (AI)*, 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 AI in an easy-to-understand way and describe some areas where AI is already being successfully applied. I will also look at some special features, such as 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.

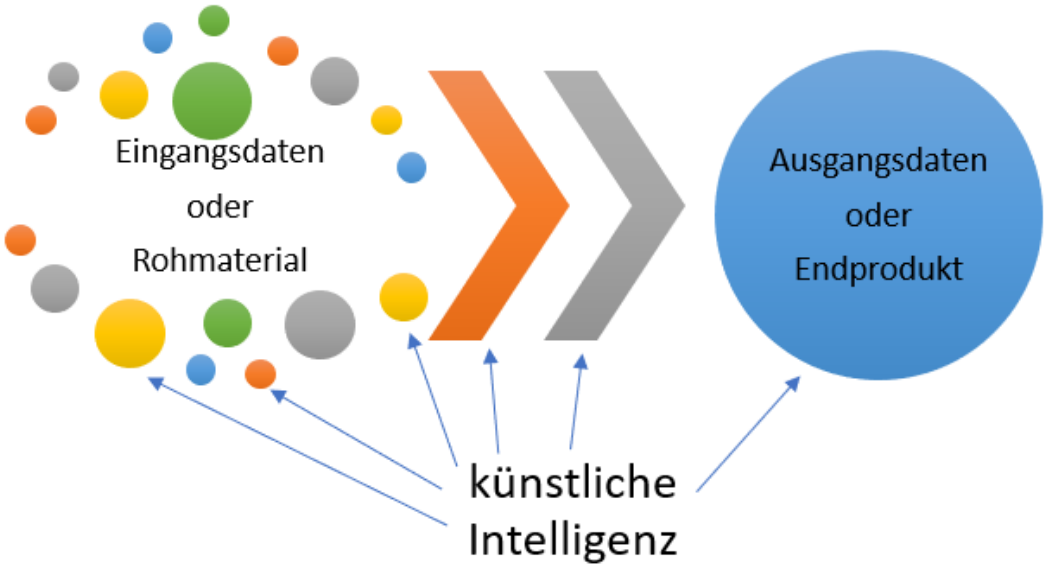
In the *third part,* I show how AI techniques can be implemented in the Python programming language when dealing with the topic of image recognition and provide my programming code in the process.

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.

5.1 1. AI explained in an easy to understand way

My diagram shows **"input data"** (or raw material) on the left and **"output data"** (or end products) on the right. In between, **processes run** (represented by the two orange/grey arrows). During the runtime of these processes, the intermediate results are usually logged and stored by means of log files (which is also data).



5.1.1 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, historised with a time stamp and stored in log files. These processes can take place regionally at different locations or organisationally in different company units. In addition, there are many other examples. I would like to mention a few 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 optimised 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 utilised. [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

[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/

to be made to the products so that they can be produced in series cost-effectively and without errors. [3]

- **Quality control**: Images of the products are generated using sensors, X-rays or high-resolution cameras. Artificial intelligence can then use image recognition algorithms to detect defects in the products and sort them out. [4][5]

I have linked further articles in the footnotes. The list could go on, but this selection should cover the most important areas.

5.1.2 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 rates the product. For example, a person 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.

5.1.3 Is artificial intelligence really intelligent or just a very clever algorithm?

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

[3] Intel, Artificial intelligence reduces costs and accelerates time to market, https://www.intel.com/content/dam/www/public/us/en/documents/papers/artificial-intelligence-reduces-costs-and-accelerates-time-to-market-paper.pdf

[4] Fraunhofer Institute, AI-based visual quality control, https://www.iais.fraunhofer.de/de/geschaeftsfelder/Computer-Vision/visuelle-qualitaetskontrolle.html

[5] Elektronik Praxis, AI in quality control: when no detail should be overlooked, https://www.elektronikpraxis.vogel.de/ki-in-der-qualitaetskontrolle-wenn-kein-detail-uebersehen-werden-darf-a-860403/

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 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.

5.2 2. how has BMW benefited from AI?

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/):

CBR Exclusive: How BMW optimised supply chain big data with Teradata

JAMES NUNNS EDITOR
10TH NOVEMBER 2016

+ INCREASE / DECREASE TEXT SIZE −

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.

5.3 3. how can AI techniques be implemented in concrete terms?

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.

5.3.1 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', 0.7614468
  ('n03777754', 'modem', 0.08100146
  ('n03272010', 'electric_guitar',
  ('n03492542', 'hard_disc', 0.0156
  ('n04070727', 'refrigerator', 0.0
```

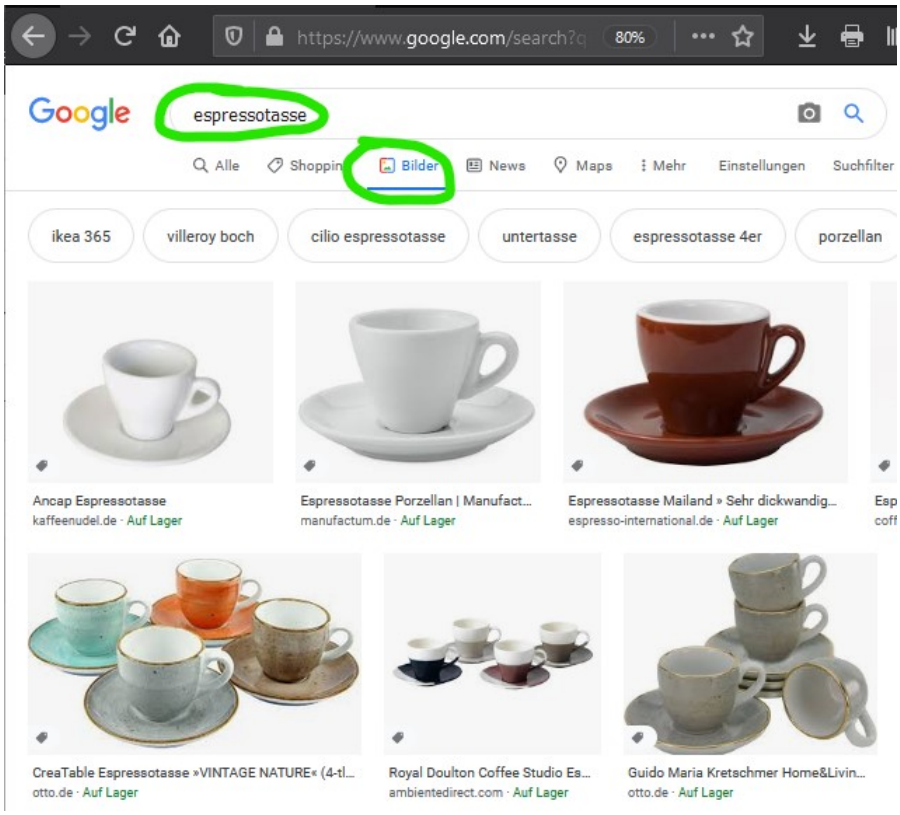
5.3.2 Which groups can be formed?

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[8]. Texts can also be grouped. In the case of a document or contract, we may be looking for similar texts.

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

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

Grouping similar things is a frequently discussed problem. We all know the useful Google function to show similar images. You enter a term (e.g. espresso cup) in the search bar and similar images are displayed:



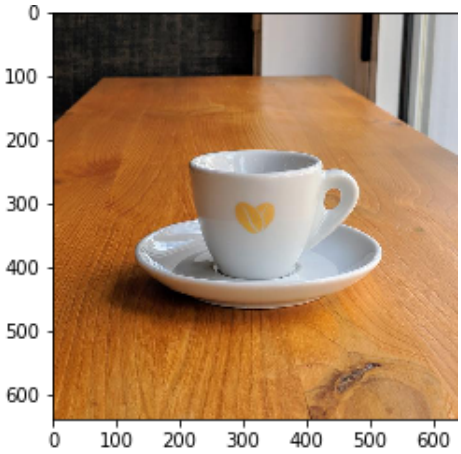
Two questions arise when implementing the programme code. The first question is:

How do you compare two pictures? Or two texts? It's not easy, but this problem has been analysed 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).[9] 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.

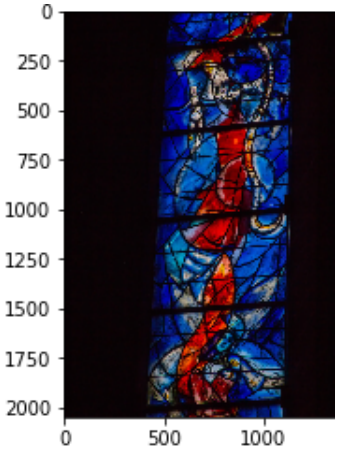
[9] Towards Datascience, Locality Sensitive HashingAn effective way of reducing the dimensionality of your data, https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

The result of my work was: I applied the LSH algorithm to over 9000 images (each about 300*300 pixels), including my collection of church windows and espresso cups (I like to drink espresso and started photographing the cups at some point). On my own computer, this grouping took a few minutes and was only necessary once. After that, I downloaded a completely new picture of an espresso cup from the internet:

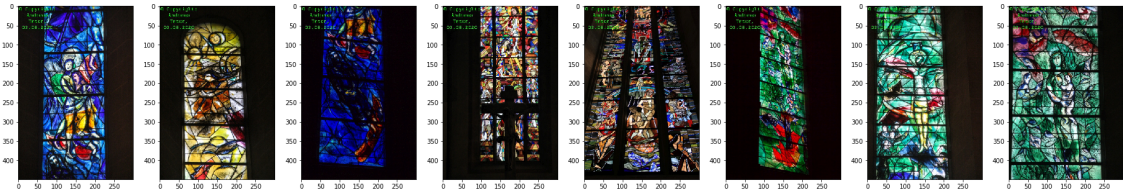


I gave this picture to the programme and then used it to search for similar pictures from my picture collection. After a few seconds, the programme showed me these 15 pictures:

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 programme 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.

5.4 4. what recommendations for implementing AI techniques would I give you?

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 organisational 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 programmes 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