Data Science - A practical Approach

Lorenz Feyen

Oct 13, 2021

CONTENTS

pdf version can be found here.

Part I

1. Introduction

CHAPTER

ONE

INTRODUCTION

this is an introduction

1.1 Structured vs Unstructured

When performing data preparation an important aspect is to consider with the type of data we are working with. In general there are 2 types of data, but you could consider a third.

1.1.1 Structured data

Structured data is data that adheres to a pre-defined data model and is therefore straightforward to analyze. This data model is the description of our data, each record has to be conform to the model. A table in a spreadsheet is a good example of the concept of structured data however often no data types are enforced, meaning a column can contain e.g. both numbers and text. Later we will see that a mixture of data types is often problematic therefor the need of a data model.

1.1.2 Unstructured data

In contrast to structured data, there is no apparent data model but this does not mean the data is unusable or cluttered. Usually it means either no data model has yet been applied or we are dealing with data that is difficult to confine in a model. A great example of this would be images, or more general (binary) files. These obviously are hard to sort yet often data structures also contain metadata from these files, with data describing things as when the file was uploaded, what is shown in the file, … In turn the metadata can be structured and a data model can be related to the unstructured data.

1.1.3 Semi-structured data

As an intermediate option, we have what is called semi-structured data. The reasoning behind this is that the concept of tables is not always applicable, in some occasions e.g. data lakes there is no complex structure present compared to a database. In a data lake files are stored similar to the folder structure in your computer, with no fancy infrastructure behind it, thus reducing operation costs. This implies that a data model can not be enforced and the data is stored in generic files.

1.2 Data Structures

There are several structures in which data can be stored and accessed, here we cover the 3 most important.

1.2.1 Data Lake

As mentioned earlier a data lake would be the most cost efficient method as it relies on the least infrastructure and can be serverless. The concept behind a data lake is straight-forward, the data is stored in simple files with a specific notation e.g. parquet, csv, xml,… What is important when designing a data lake would be partitioning, this can be achieved by using subfolders and saving parts of the data in different files. To make this more tangible, take a look at this symbolic example I provided. Instead of putting all data in one csv file, subfolder divide the data in Country, City and then the year. We could even further partition yet the data is here in daily frequency so that would create many small partitions. The difficulty for a data lake lies in the method of interacting, when adding new data one has to adhere to a agreed upon data model that is not enforced, meaning you could create incorrect data which then need to be cleaned. On the other hand efficiency of you data lake depends on good partitioning, as the order of divisioning of your folders. We could have also divided first on year and then on country and city. As a data scientist seeing the data lake might not be as common, as this is rather an engineering task, however using the concepts of a data lake in experimental projects can make a big difference.

1.2.2 Database

Another interesting data structure is the database, widely used for exceptional speeds and ease of use, yet costly in storage. Numerous implementations of servers using the SQL language are developed over the years with each their own dialect and advantages. The important take home message here is that you can easily perform queries on the database that pre-handles the data to retrieve the information you need. these operations include filtering, grouping categories, joining tables, ordering and much more, as SQL is a complete language on its own. As a data scientist these databases are much more common, so SQL is a good asset to learn!

1.2.3 Data Warehouse

A next step towards data analysis is the data warehouse, where a database is composed of the most pragmatic method of storing your data a data warehouse consist of multiple views on your data. Based upon the data of a dataset the data warehouse transforms this data into a new format that displays the data in a new way. Let me illustrate with with a simple example, we have a database with a table that contains the rentals of books from multiple libraries. This table has a few columns: a timestamp, the library, the action (rent, return, …), the client_id and the book_id. If you would want to know if a book is available this database is perfect for your needs as you just have to find the last event for that book and if its a return the book is (or should be) there. Now image we would want to know how many books are being rented per month this database is insufficient, yet our data warehouse might contain such a view! It is up to the data engineer/scientist to create a computation that displays the amount of books rented per month. If they also would like to subdivided it per category of books, you would need to incorporate another table of the database where information of the books is stored. More on these operations of a data warehouse will be seen in the data preprocessing chapter. One last remark about data warehousing, it is important to optimize between memory and computation. Tables in our data warehouse compared to database can be computed in place reducing memory costs yet increasing computation costs. If a visualization tool often queries a table in your warehouse it is favorable to create it as a table in your database.

1.3 OLTP and OLAP

From the previous section you might have deduced that a database and Data Warehouse serve 2 different purposes. These are denoted as OnLine Transaction Processing and OnLine Analytical Processing, as the names suggest these are used for transactional and analytical processes.

1.3.1 OLTP

For this method the database structure is optimal, let us review the example where we have libraries renting out books. Renting out a book would send a message to our OLTP system creating a new record stating that specific book is at this moment rented out from our library. OLTP handles day-to-day operational data that can be both written and read from our database.

1.3.2 OLAP

In the case we would like to analyse data from the libraries we would use the OLAP method, creating multi-dimensional views from our transactional data. Our dimensions would be the date (aggregated per month), the library and the category of book, the chapter of data preprocessing will use these operations practically. I could write a whole chapter on OLAP operations however they are well described in this wikipedia page.

Part II

2. Data Preparation

CHAPTER

TWO

INTRODUCTION

When performing data science, we often do not elaborate about the preparation that went into the dataset. It is considered tedious and irrelevant to the story of the analysis, however it is often the most important part of data analysis. Data Preparation is the metaphorical foundation of your construction, if you fail to prepare data, you prepare to fail your analysis.

> Good data beats a fancy algorithm

If you would perform an analysis and insert unprepared data, you will mostly be disappointed with the result.

2.1 why Data Preparation?

Aside from metaphors let us make the reasoning behind this step more tangile, to explain the relevance of this step, we partitioned the answer into a few key points.

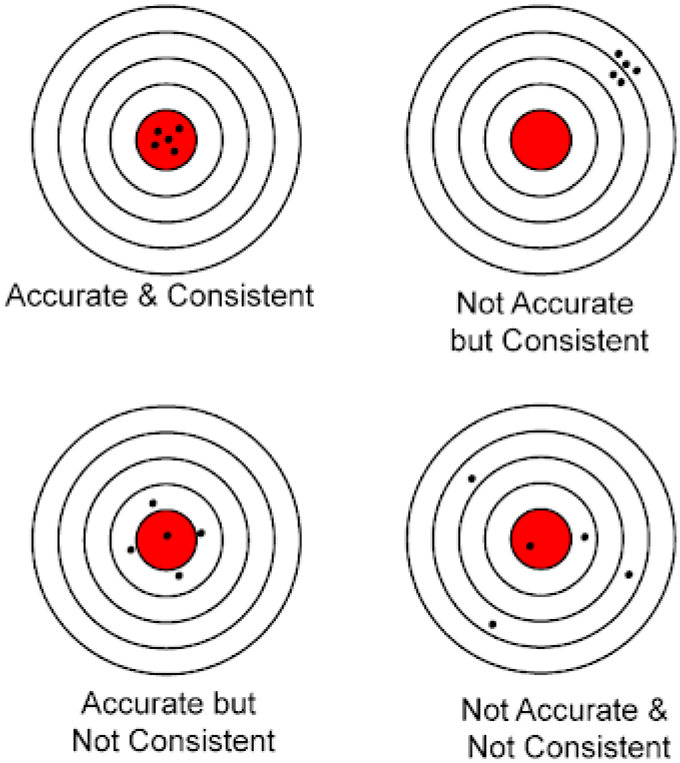
2.1.1 Accuracy

There is no excuse for incorrect data and accuracy is the most important attribute. Let us assume that we have a dataset where for some reason the result are not accurate. This would led us to an analysis where we conclude a result that contains a bias. An example would be a dataset of sold cars, where the listed price is that of the stock car without options. Options are not incorporated in the price and we are perhaps training an algorithm that predicts the stock price. If you as a data scientist fail to report/correct this, your predictions are making sense, but always underestimate!

2.1.2 Consistency

They usually say something such as 'consistency is key' and with data preparation that is likewise true. A dataset where we do not have consistent results will never converge towards a particular answer. Note however that it might not be a problem of consistency but rather you are missing crucial information. If we would have a dataset where local temperatures are logged, we would like to see a consistency each 24 hours. However we do see there are day to day fluctuations, so perhaps we need to keep track of cloud and rain data to make the dataset more complete. We could then see that the results are more consistent yet the possibility of outliers is still present. Equally possible would be that our temperature sensor is not sensitive enough or has large fluctuations in readings, it is the task of the data scientist to figure this out.

To get a visual about accuracy and consistency this picture might help:



2.1.3 Completeness

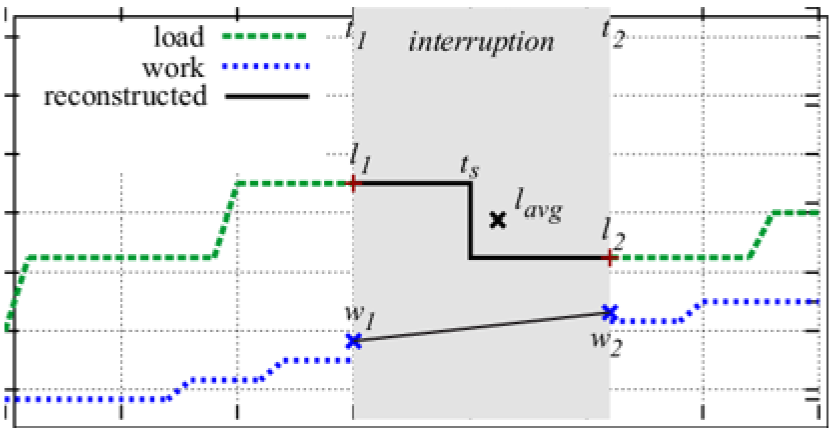
As hinted in the previous point, completeness is something you have to be aware of. Having 'complete' data is crucial for you narrative to give a correct answer, as you might otherwise lose detail. Note that you never will know if your data is complete as there might always be more data to mine. Yet you have to make a consideration between collecting more data and the effort required. This collecting can happen in multiple methods, as an example we use a survey where we asked several people 10 different questions, we could:

- gather new data, here our data grows 'longer' by asking the 10 question to more people. It might be that our sample of people were only students at a campus, so our data was not complete.
- gather new feature, by asking more questions to the same people (in case we could still find them). By doing this we get a better understanding of their opinion, again making our data more complete.
- fill missing values, by imputing the abstained questions with answers of similar records. When someone answered they did not want to answer we could figure out what they would have answered by looking at what persons answered that reply in a similar way.

2.1.4 Timeliness

For some datasets we are dealing with data that is time related. It can happen that data at specific timepoints is missing or delayed, resulting in a failure to use machine learning algorithms. A well-organised data pipeline utilises techniques of data preparation to circumvent these outages, usually this would be to retain the last successful datapoint. However in hindsight we could use more complex strategies to fill in these gaps or correct datetimes in our dataset,

In this example the data stream is interrupted and data preparation is there to handle these outages before we can perform analysis.



2.1.5 Believability

You could collect the most intricate dataset possible, but if the narrative that you are conducting contradicts itself, you will end up nowhere. During the process of data analytics it is important to apply a critical mind to what your dataset is telling you. Obviously this is not a reason to mask or mold the data so it agrees with your opinion. Rather you should be wary when conflicts happen and act accordingly, unfortunately it is impossible to write a generic tactic for this. As a data scientist your experience of the underlying subject should help create understanding of the topic, remember, gathering information from experts in the field is crucial here!

2.1.6 Interpretability

Another problem that might arise when you are diving deep into the data might be that you have created something no human could ever interpret. The Machine Learning algorithms outputs plausible and believable results, but it is impossible to understand the reasoning behind. For some this is perfectly acceptible, for some this is undesirable. It is your task as a data scientist to cater the wishes of the product operator and if they desire understanding as they would like to learn from the data driven process you need to unfold the process. Usually this comes down to which data transformations are used as some do produce an output that only makes mathematical sense.

2.1.7 In conclusion

There are multiple ways to deteriorate the quality of your data and raw formats of data often contain multiple. Before we can do anything with it these problems need to be resolved, if you fail to do so, the final output fails too.

2.2 Further reading

Towards Data Science

CHAPTER

THREE

MISSING DATA

In this notebook we will look at a few datasets where values from columns are missing. It is crucial for data science and machine learning to have a dataset where no values are missing as algorithms are usually not able to handle data with information missing.

For python, we will be using the pandas library to handle our dataset.

```
import pandas as pd
```

3.1 Kamyr digester

The first dataset we will be looking at is taken from a physical device equiped with numerous sensors, each timepoint (1 hour) these sensors are read out and the data is collected. Let's have a look at the general structure

```
kamyr_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c2_data_preparation/data/kamyr-digester.csv')
kamyr_df.head()
```

```
  Observation  Y-Kappa  ChipRate  BF-CMratio  BlowFlow  ChipLevel4   \
0    31-00:00    23.10    16.520     121.717  1177.607      169.805
1    31-01:00    27.60    16.810      79.022  1328.360      341.327
2    31-02:00    23.19    16.709      79.562  1329.407      239.161
3    31-03:00    23.60    16.478      81.011  1334.877      213.527
4    31-04:00    22.90    15.618      93.244  1334.168      243.131

   T-upperExt-2   T-lowerExt-2    UCZAA  WhiteFlow-4    ...  SteamFlow-4   \
0        358.282        329.545  1.443       599.253  ...        67.122
1        351.050        329.067  1.549       537.201  ...        60.012
2        350.022        329.260  1.600       549.611  ...        61.304
3        350.938        331.142  1.604       623.362  ...        68.496
4        351.640        332.709    NaN       638.672  ...        70.022

   Lower-HeatT-3  Upper-HeatT-3   ChipMass-4   WeakLiquorF   BlackFlow-2   \
0        329.432        303.099      175.964      1127.197      1319.039
1        330.823        304.879      163.202       665.975      1297.317
2        329.140        303.383      164.013       677.534      1327.072
3        328.875        302.254      181.487       767.853      1324.461
4        328.352        300.954      183.929       888.448      1343.424

   WeakWashF   SteamHeatF-3   T-Top-Chips-4   SulphidityL-4
0     257.325         54.612         252.077             NaN
1     241.182         46.603         251.406           29.11
```

(continues on next page)

(continued from previous page)

```
2      237.272          51.795          251.335             NaN
3      239.478          54.846          250.312           29.02
4      215.372          54.186          249.916           29.01

[5 rows x 23 columns]
```

Interesting, there seem to be 22 sensor values and 1 timestamp for each record. As mechanical devices are prone to noise and dropouts of sensors we would be foolish to assume no missing values are present.

```
kamyr_df.isna().sum().divide(len(kamyr_df)).round(4)*100
```

```
Observation         0.00
Y-Kappa             0.00
ChipRate            1.33
BF-CMratio          4.65
BlowFlow            4.32
ChipLevel4          0.33
T-upperExt-2        0.33
T-lowerExt-2        0.33
UCZAA               7.97
WhiteFlow-4         0.33
AAWhiteSt-4        46.84
AA-Wood-4           0.33
ChipMoisture-4      0.33
SteamFlow-4         0.33
Lower-HeatT-3       0.33
Upper-HeatT-3       0.33
ChipMass-4          0.33
WeakLiquorF         0.33
BlackFlow-2         0.33
WeakWashF           0.33
SteamHeatF-3        0.33
T-Top-Chips-4       0.33
SulphidityL-4      46.84
dtype: float64
```

As expected, the datapoint 'AAWhiteSt-4' even has 46% of data missing! It seems we only have 300 datapoints and presumably these missing values occur in different records our dataset will be decimated if we just drop all rows with missing values.

```
kamyr_df.shape
```

```
(301, 23)
```

```
kamyr_df.dropna().shape
```

```
(131, 23)
```

As we drop all rows with missing values, we are left with only 131 records. Whilst this might be good enough for some purposes, there are more viable options.

Perhaps we can first remove the column with the most missing values and then drop all remaining

```
kamyr_df.drop(columns=['AAWhiteSt-4 ','SulphidityL-4 ']).dropna().shape
```

```
(263, 21)
```

Significantly better, although we lost the information of 2 sensors we now have a complete dataset with 263 records. For purposes where those 2 sensors are irrelevant this is a viable option, keep in mind that this dataset is still 100% truthful, as we have not imputed any values.

Another option, where we retain all our records would be using the timely nature of our dataset, each record is a measurement with an interval of 1 hour. I have no knowledge of this dataset but one might make the assumption that the interval of 1 hour is taken as the state of the machine does not alter much in 1 hour. Therefore we could do what is called a forward fill, where we fill in the missing values with the same value of the sensor for the previous measurement.

This would solve nearly all nan values as there might be a problem where the first value is missing. This is shown below.

```
kamyr_df.fillna(method='ffill')['SulphidityL-4 ']
```

```
0         NaN
1       29.11
2       29.11
3       29.02
4       29.01
        ...
296     30.43
297     30.29
298     30.47
299     30.47
300     30.46
Name: SulphidityL-4 , Length: 301, dtype: float64
```

Although our dataset is not fully the truth, we can see that little to no changes occur in the sensor and using a forward fill is arguably the most suitable option.

3.2 Travel times

Another dataset from the same source contains a collection of recorded travel times and specific information about the travel itself as e.g.: the day of the week, where they were going, …

```
travel_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c2_data_preparation/data/travel-times.csv')
travel_df
```

```
          Date StartTime  DayOfWeek GoingTo  Distance  MaxSpeed  AvgSpeed  \
0     1/6/2012     16:37     Friday    Home     51.29     127.4      78.3
1     1/6/2012     08:20     Friday     GSK     51.63     130.3      81.8
2     1/4/2012     16:17  Wednesday    Home     51.27     127.4      82.0
3     1/4/2012     07:53  Wednesday     GSK     49.17     132.3      74.2
4     1/3/2012     18:57    Tuesday    Home     51.15     136.2      83.4
..         ...       ...        ...     ...       ...       ...       ...
200  7/18/2011     08:09     Monday     GSK     54.52     125.6      49.9
201  7/14/2011     08:03   Thursday     GSK     50.90     123.7      76.2
202  7/13/2011     17:08  Wednesday    Home     51.96     132.6      57.5
203  7/12/2011     17:51    Tuesday    Home     53.28     125.8      61.6
204  7/11/2011     16:56     Monday    Home     51.73     125.0      62.8

     AvgMovingSpeed FuelEconomy  TotalTime  MovingTime Take407All Comments
```

(continues on next page)

(continued from previous page)

```
0              84.8         NaN       39.3        36.3         No       NaN
1              88.9         NaN       37.9        34.9         No       NaN
2              85.8         NaN       37.5        35.9         No       NaN
3              82.9         NaN       39.8        35.6         No       NaN
4              88.1         NaN       36.8        34.8         No       NaN
..              ...         ...        ...         ...        ...       ...
200            82.4        7.89       65.5        39.7         No       NaN
201            95.1        7.89       40.1        32.1        Yes       NaN
202            76.7         NaN       54.2        40.6        Yes       NaN
203            87.6         NaN       51.9        36.5        Yes       NaN
204            92.5         NaN       49.5        33.6        Yes       NaN

[205 rows x 13 columns]
```

we have a total of 205 records and we can already see that the FuelEconomy column seems pretty bad, let's quantify that.

```
travel_df.isna().sum().divide(len(travel_df)).round(4)*100
```

```
Date                   0.00
StartTime              0.00
DayOfWeek              0.00
GoingTo                0.00
Distance               0.00
MaxSpeed               0.00
AvgSpeed               0.00
AvgMovingSpeed         0.00
FuelEconomy            8.29
TotalTime              0.00
MovingTime             0.00
Take407All             0.00
Comments              88.29
dtype: float64
```

In the end, it doesn't seem that bad, but there are comments and nearly none of them are filled in. Which in perspective is understandable. Let's see what the comments look like

```
travel_df[~travel_df.Comments.isna()].Comments
```

```
15                                    Put snow tires on
39                                           Heavy rain
49                                  Huge traffic backup
50        Pumped tires up: check fuel economy improved?
52                                   Backed up at Bronte
54                                   Backed up at Bronte
60                                                Rainy
78                                     Rain, rain, rain
91                                     Rain, rain, rain
92           Accident: backup from Hamilton to 407 ramp
110                                             Raining
132                               Back to school traffic?
133                      Took 407 all the way (to McMaster)
150                                Heavy volume on Derry
156                             Start early to run a batch
158      Accident at 403/highway 6; detour along Dundas
165                                        Detour taken
166                                       Must be Friday
```

(continues on next page)

(continued from previous page)

```
172                                 Medium amount of rain
174                                             New tires
182                                  Turn around on Derry
184                                           Empty roads
187                                Police slowdown on 403
189                             Accident blocked 407 exit
Name: Comments, dtype: object
```

As you would expect, these comments are text based. Now imagine we would like to run some Natural Language Processing (NLP) on these, it would be a pain to perform string operations on it when it is riddled with missing values.

Here a simple example where we select all records containing the word 'rain', with no avail.

```
travel_df[travel_df.Comments.str.lower().str.contains('rain')]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_6376/1298831137.py in <module>
----> 1 travel_df[travel_df.Comments.str.lower().str.contains('rain')]

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪frame.py in __getitem__(self, key)
   3446
   3447         # Do we have a (boolean) 1d indexer?
-> 3448         if com.is_bool_indexer(key):
   3449             return self._getitem_bool_array(key)
   3450

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪common.py in is_bool_indexer(key)
    137                     # Don't raise on e.g. ["A", "B", np.nan], see
    138                     #  test_loc_getitem_list_of_labels_categoricalindex_with_
↪na
--> 139                     raise ValueError(na_msg)
    140                 return False
    141             return True

ValueError: Cannot mask with non-boolean array containing NA / NaN values
```

The last line of the python error traceback gives us the reason it failed, because there were NaN values present.

Luckily the string variable has more or less it's on 'null' value, being an empty string, this way these operations are still possible, most of the comments will just contain nothing.

```
travel_df.Comments = travel_df.Comments.fillna('')
```

```
travel_df[travel_df.Comments.str.lower().str.contains('rain')]
```

```
           Date StartTime  DayOfWeek GoingTo  Distance  MaxSpeed  AvgSpeed  \
39   11/29/2011     07:23    Tuesday     GSK     51.74     112.2      55.3
60    11/9/2011     16:15  Wednesday    Home     51.28     121.4      65.9
78   10/25/2011     17:24    Tuesday    Home     52.87     123.5      65.1
91   10/12/2011     17:47  Wednesday    Home     51.40     114.4      59.7
110   9/27/2011     07:36    Tuesday     GSK     50.65     128.1      86.3
172    8/9/2011     08:15    Tuesday     GSK     49.08     134.8      60.5
```

(continues on next page)

(continued from previous page)

```
     AvgMovingSpeed FuelEconomy  TotalTime  MovingTime Take407All  \
39             61.0         NaN       56.2        50.9         No
60             71.8        9.35       46.7        42.1         No
78             72.4        8.97       48.7        43.8         No
91             65.8        8.75       51.7        46.9         No
110            88.6        8.31       35.2        34.3        Yes
172            67.2        8.54       48.7        43.8         No

                   Comments
39               Heavy rain
60                    Rainy
78         Rain, rain, rain
91         Rain, rain, rain
110                 Raining
172  Medium amount of rain
```

Fixed! now we can use the comments for analysis.

We still have to fix the FuelEconomy, let us take a look at the non NaN values

```
travel_df[~travel_df.FuelEconomy.isna()]
```

```
           Date StartTime  DayOfWeek GoingTo  Distance  MaxSpeed  AvgSpeed  \
6      1/2/2012     17:31     Monday    Home     51.37     123.2      82.9
7      1/2/2012     07:34     Monday     GSK     49.01     128.3      77.5
8    12/23/2011     08:01     Friday     GSK     52.91     130.3      80.9
9    12/22/2011     17:19   Thursday    Home     51.17     122.3      70.6
10   12/22/2011     08:16   Thursday     GSK     49.15     129.4      74.0
..          ...       ...        ...     ...       ...       ...       ...
197   7/20/2011     08:24  Wednesday     GSK     48.50     125.8      75.7
198   7/19/2011     17:17    Tuesday    Home     51.16     126.7      92.2
199   7/19/2011     08:11    Tuesday     GSK     50.96     124.3      82.3
200   7/18/2011     08:09     Monday     GSK     54.52     125.6      49.9
201   7/14/2011     08:03   Thursday     GSK     50.90     123.7      76.2

     AvgMovingSpeed FuelEconomy  TotalTime  MovingTime Take407All Comments
6              87.3           -       37.2        35.3         No
7              85.9           -       37.9        34.3         No
8              88.3        8.89       39.3        36.0         No
9              78.1        8.89       43.5        39.3         No
10             81.4        8.89       39.8        36.2         No
..              ...         ...        ...         ...        ...      ...
197            87.3        7.89       38.5        33.3        Yes
198           102.6        7.89       33.3        29.9        Yes
199            96.4        7.89       37.2        31.7        Yes
200            82.4        7.89       65.5        39.7         No
201            95.1        7.89       40.1        32.1        Yes

[188 rows x 13 columns]
```

It seems that aside NaN values there are also other intruders, a quick check on the data type (Dtype) reveils it is not recognised as a number!

```
travel_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Date            205 non-null    object
 1   StartTime       205 non-null    object
 2   DayOfWeek       205 non-null    object
 3   GoingTo         205 non-null    object
 4   Distance        205 non-null    float64
 5   MaxSpeed        205 non-null    float64
 6   AvgSpeed        205 non-null    float64
 7   AvgMovingSpeed  205 non-null    float64
 8   FuelEconomy     188 non-null    object
 9   TotalTime       205 non-null    float64
 10  MovingTime      205 non-null    float64
 11  Take407All      205 non-null    object
 12  Comments        205 non-null    object
dtypes: float64(6), object(7)
memory usage: 20.9+ KB
```

The column is noted as an object or string type, meaning that these numbers are given as '9.24' instead of 9.24 and numerical operations are not possible. We can cast them to numeric but have to warn pandas to coerce errors, meaning errors will be converted to NaN values. Later we'll handle the NaN's.

```
travel_df.FuelEconomy = pd.to_numeric(travel_df.FuelEconomy, errors='coerce')
travel_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 205 entries, 0 to 204
Data columns (total 13 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Date            205 non-null    object
 1   StartTime       205 non-null    object
 2   DayOfWeek       205 non-null    object
 3   GoingTo         205 non-null    object
 4   Distance        205 non-null    float64
 5   MaxSpeed        205 non-null    float64
 6   AvgSpeed        205 non-null    float64
 7   AvgMovingSpeed  205 non-null    float64
 8   FuelEconomy     186 non-null    float64
 9   TotalTime       205 non-null    float64
 10  MovingTime      205 non-null    float64
 11  Take407All      205 non-null    object
 12  Comments        205 non-null    object
dtypes: float64(7), object(6)
memory usage: 20.9+ KB
```

Wonderful, now the column is numerical and we can see 2 more missing values have popped up! We could easily drop these 19 records and have a complete dataset.

```
travel_df.dropna()
```

```
           Date StartTime  DayOfWeek GoingTo  Distance  MaxSpeed  AvgSpeed  \
8    12/23/2011     08:01     Friday     GSK     52.91     130.3      80.9
```

(continues on next page)

(continued from previous page)

```
9    12/22/2011     17:19   Thursday    Home     51.17     122.3      70.6
10   12/22/2011     08:16   Thursday     GSK     49.15     129.4      74.0
11   12/21/2011     07:45  Wednesday     GSK     51.77     124.8      71.7
12   12/20/2011     16:05    Tuesday    Home     51.45     130.1      75.2
..          ...       ...        ...     ...       ...       ...       ...
197   7/20/2011     08:24  Wednesday     GSK     48.50     125.8      75.7
198   7/19/2011     17:17    Tuesday    Home     51.16     126.7      92.2
199   7/19/2011     08:11    Tuesday     GSK     50.96     124.3      82.3
200   7/18/2011     08:09     Monday     GSK     54.52     125.6      49.9
201   7/14/2011     08:03   Thursday     GSK     50.90     123.7      76.2

     AvgMovingSpeed  FuelEconomy  TotalTime  MovingTime Take407All Comments
8              88.3         8.89       39.3        36.0         No
9              78.1         8.89       43.5        39.3         No
10             81.4         8.89       39.8        36.2         No
11             78.9         8.89       43.3        39.4         No
12             82.7         8.89       41.1        37.3         No
..              ...          ...        ...         ...        ...      ...
197            87.3         7.89       38.5        33.3        Yes
198           102.6         7.89       33.3        29.9        Yes
199            96.4         7.89       37.2        31.7        Yes
200            82.4         7.89       65.5        39.7         No
201            95.1         7.89       40.1        32.1        Yes

[186 rows x 13 columns]
```

However im leaving them as an excercise for you to apply a technique we will see in the next part

3.3 Material properties

Another dataset from the same source contains the material properties from 30 samples, this time there is not timestamp as the samples are not related in time with each other.

```
material_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c2_data_preparation/data/raw-material-properties.csv')
material_df
```

```
    Sample  size1  size2  size3  density1  density2  density3
0   X12558  0.696   2.69   6.38      41.8     17.18      3.90
1   X14728  0.636   2.30   5.14      38.1     12.73      3.89
2   X15468  0.841   2.85   5.20      37.6     13.58      3.98
3   X21364  0.609   2.13   4.62      34.2     11.12      4.02
4   X23671  0.684   2.16   4.87      36.4     12.24      3.92
5   X24055  0.762   2.81   6.36      38.1     13.28      3.89
6   X24905  0.552   2.34   5.03      41.3     16.71      3.86
7   X25917  0.501   2.17   5.09       NaN       NaN       NaN
8   X27871  0.619   2.11   5.13       NaN       NaN       NaN
9   X28690  0.610   2.10   4.18      35.0     12.15      3.86
10  X31385  0.532   2.09   4.93       NaN       NaN       NaN
11  X31813  0.738   2.29   5.47       NaN       NaN       NaN
12  X32807  0.779   2.62   5.59       NaN       NaN       NaN
13  X33943  0.537   2.23   5.41      35.2     11.34      3.99
14  X35035  0.702   2.05   5.10      34.2     10.54      4.02
15  X39223  0.768   2.51   5.09      34.9     12.55      3.90
```

(continues on next page)

(continued from previous page)

```
16  X40503  0.714   2.56   6.03       35.6      12.20       4.02
17  X41400  0.621   2.42   5.10       38.7      14.27       3.98
18  X42988  0.726   2.11   4.69       37.1      13.14       3.98
19  X44749  0.698   2.36   5.40       36.6      12.16       4.01
20  X45295    NaN    NaN    NaN       38.1      13.34       3.89
21  X46965  0.759   2.47   4.83       38.7      14.83       3.89
22  X49666  0.535   2.13   5.23        NaN        NaN        NaN
23  X50678  0.716   2.29   5.45       37.3      13.70       3.92
24  X52894  0.635   2.08   4.94        NaN        NaN        NaN
25  X53925  0.598   2.12   4.69       37.9      13.45       3.78
26  X54254  0.700   2.47   5.22       38.8      14.72       3.92
27  X54272  0.957   2.96   7.37       36.2      13.38       4.20
28  X54394  0.759   2.66   5.36       35.2      12.19       3.98
29  X55408  0.661   2.10   4.27        NaN        NaN        NaN
30  X56952  0.646   2.38   4.51       40.1      15.68       3.86
31  X57095  0.662   2.34   4.71       35.0      12.37       3.90
32  X57128  0.749   2.43   5.16       37.3      13.04       3.92
33  X61870  0.598   2.21   4.90        NaN        NaN        NaN
34  X61888  0.619   2.59   5.81        NaN        NaN        NaN
35  X72736  0.693   2.05   5.02       39.6      15.55       3.94
```

let us quantify the amount of missing data

```
material_df.isna().sum().divide(len(material_df)).round(4)*100
```

```
Sample       0.00
size1        2.78
size2        2.78
size3        2.78
density1    27.78
density2    27.78
density3    27.78
dtype: float64
```

Unfortunately that is a lot of missing data, covered in all records, dropping here seems almost impossible if we want to keep a healthy amount of records.

Here it would be wise to go for a more elaborate method of imputation, I opted for the K-nearest neighbours method, which looks at the K most similar records in the dataset to make an educated guess on what the missing value could be, this because we can assume that records with similar data are also similar over all the properties (columns).

Im using the sklearn library for this, which has more imputation techniques such as MICE. More info can be found here

```
from sklearn.impute import KNNImputer
```

im creating an imputer object and specify that i want to use the 5 most similar records and weigh them by distance from the to imputed record, meaning closer neighbours are more important.

```
imputer = KNNImputer(n_neighbors=5, weights="distance")
```

As the imputer only takes numerical values I had to do some pandas magic and drop the first column, which I then added again. The result is a fully filled dataset, you can recognise the new values as they are not rounded.

```
pd.DataFrame(
    imputer.fit_transform(material_df.drop(columns=['Sample'])),
    columns=material_df.columns.drop('Sample')
)
```

```
      size1     size2     size3  density1   density2  density3
0   0.696000  2.690000  6.380000  41.800000  17.180000  3.900000
1   0.636000  2.300000  5.140000  38.100000  12.730000  3.890000
2   0.841000  2.850000  5.200000  37.600000  13.580000  3.980000
3   0.609000  2.130000  4.620000  34.200000  11.120000  4.020000
4   0.684000  2.160000  4.870000  36.400000  12.240000  3.920000
5   0.762000  2.810000  6.360000  38.100000  13.280000  3.890000
6   0.552000  2.340000  5.030000  41.300000  16.710000  3.860000
7   0.501000  2.170000  5.090000  38.495282  14.029399  3.931180
8   0.619000  2.110000  5.130000  37.405275  13.157346  3.943667
9   0.610000  2.100000  4.180000  35.000000  12.150000  3.860000
10  0.532000  2.090000  4.930000  37.811132  13.646072  3.908364
11  0.738000  2.290000  5.470000  37.088833  13.255412  3.941654
12  0.779000  2.620000  5.590000  36.540567  12.889902  3.970973
13  0.537000  2.230000  5.410000  35.200000  11.340000  3.990000
14  0.702000  2.050000  5.100000  34.200000  10.540000  4.020000
15  0.768000  2.510000  5.090000  34.900000  12.550000  3.900000
16  0.714000  2.560000  6.030000  35.600000  12.200000  4.020000
17  0.621000  2.420000  5.100000  38.700000  14.270000  3.980000
18  0.726000  2.110000  4.690000  37.100000  13.140000  3.980000
19  0.698000  2.360000  5.400000  36.600000  12.160000  4.010000
20  0.733097  2.653959  5.881504  38.100000  13.340000  3.890000
21  0.759000  2.470000  4.830000  38.700000  14.830000  3.890000
22  0.535000  2.130000  5.230000  37.391815  13.089536  3.944335
23  0.716000  2.290000  5.450000  37.300000  13.700000  3.920000
24  0.635000  2.080000  4.940000  37.254724  13.206262  3.933904
25  0.598000  2.120000  4.690000  37.900000  13.450000  3.780000
26  0.700000  2.470000  5.220000  38.800000  14.720000  3.920000
27  0.957000  2.960000  7.370000  36.200000  13.380000  4.200000
28  0.759000  2.660000  5.360000  35.200000  12.190000  3.980000
29  0.661000  2.100000  4.270000  36.172345  12.755632  3.887375
30  0.646000  2.380000  4.510000  40.100000  15.680000  3.860000
31  0.662000  2.340000  4.710000  35.000000  12.370000  3.900000
32  0.749000  2.430000  5.160000  37.300000  13.040000  3.920000
33  0.598000  2.210000  4.900000  37.865882  13.826029  3.887021
34  0.619000  2.590000  5.810000  35.932339  12.318210  3.989911
35  0.693000  2.050000  5.020000  39.600000  15.550000  3.940000
```

This concludes the part of missing values, perhaps you can try yourself and impute the missing values for the FuelEconomy using the SimpleImputer or even the IterativeImputer.

CHAPTER

FOUR

CONCATENATION AND DEDUPLICATION

In this notebook we are going to investigate the concepts of stitching data files (concatenation) and verifying the integrity of our data concercing duplicates

4.1 Concatenation

When dealing with large amounts of data, fractioning is often the only solution. Not only does this tidy up your data space, but it also benefits computation. Aside from that, appending new data to your data lake is independent of the historical data. However if you want to perform historical analysis this means you will need to perform additional operations.

In this notebook we have a setup of a very small data lake containing daily minimal temperatures. If you would look closely in the url you would see the following structure.

> data/temperature/australia/melbourne/1981.csv

This is a straight-forward but perfect example on how fragmentation works, in our data lake we have: temperatures data fractioned by country, city and year. As we are working with daily temperatures further fractioning would not be interesting, but you could fraction e.g. per month.

In the cells below, we read our both 1981 and 1982 data and concatenate them using python.

```
import pandas as pd
```

```
melbourne_1981_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-
↪science-practical-approach/main/src/c2_data_preparation/data/temperatures/australia/
↪melbourne/1981.csv')
```

```
melbourne_1982_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-
↪science-practical-approach/main/src/c2_data_preparation/data/temperatures/australia/
↪melbourne/1982.csv')
```

```
df = pd.concat(
    [
        melbourne_1981_df,
        melbourne_1982_df,
    ]
)
```

```
df
```

```
          Date  Temp
0   1981-01-01  20.7
1   1981-01-02  17.9
2   1981-01-03  18.8
3   1981-01-04  14.6
4   1981-01-05  15.8
..         ...   ...
360 1982-12-27  15.3
361 1982-12-28  16.3
362 1982-12-29  15.8
363 1982-12-30  17.7
364 1982-12-31  16.3

[730 rows x 2 columns]
```

And there you have it! we now have a dataframe containing both data from 1981 as 1982. Can you figure out what I calculated in the next cell? Do you think there might be a more 'clean' solution?

```
df[df.Date.str[5:7]== '01'].Temp.mean()
```

```
17.140322580645158
```

As an exercise I would ask you now to create a small python script that given a begin and end year (between 1981 and 1990) can automatically concatenate all the necessary data

```
for i in range(1982,1987):
    print(i)
```

```
1982
1983
1984
1985
1986
```

4.2 Deduplication

Another important aspect of data cleaning is the removal of duplicates. Here we fragment of a dataset from activity on a popular games platform. We can see which user has either bought or played specific games and how often. Unfortunately for some reason, entries might have duplicates which we have to deal with as otherwise users might have e.g. bought a game twice.

```
df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-practical-
↪approach/main/src/c2_data_preparation/data/steam.csv')
df
```

```
       user_id                                  game    action  freq
0     11373749              Sid Meier's Civilization IV  purchase   1.0
1     11373749              Sid Meier's Civilization IV      play   0.1
2     11373749              Sid Meier's Civilization IV  purchase   1.0
3     11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
4     11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
...        ...                                   ...       ...   ...
1834 112845094                                Arma 2  purchase   1.0
```

(continues on next page)

(continued from previous page)

```
1835  112845094                  Grand Theft Auto San Andreas  purchase   1.0
1836  112845094                    Grand Theft Auto Vice City  purchase   1.0
1837  112845094                    Grand Theft Auto Vice City  purchase   1.0
1838  112845094                          Grand Theft Auto III  purchase   1.0

[1839 rows x 4 columns]
```

We have a dataframe with 1839 interactions, you can see that the freq either notes the amount they bought (which always 1 as there is not use in buying it more) or the amount in hours they played.

Let us straightforward ask pandas to remove all rows that have an exact duplicate

```
df.drop_duplicates()
```

```
        user_id                                        game    action  freq
0      11373749                   Sid Meier's Civilization IV  purchase   1.0
1      11373749                   Sid Meier's Civilization IV      play   0.1
3      11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
5      11373749          Sid Meier's Civilization IV Warlords  purchase   1.0
7      56038151                       Tom Clancy's H.A.W.X. 2  purchase   1.0
...         ...                                         ...       ...   ...
1831  112845094                  Grand Theft Auto San Andreas  purchase   1.0
1832  112845094                  Grand Theft Auto San Andreas      play   0.2
1833  112845094                          Grand Theft Auto III  purchase   1.0
1834  112845094                                        Arma 2  purchase   1.0
1836  112845094                    Grand Theft Auto Vice City  purchase   1.0

[1132 rows x 4 columns]
```

Alright! this seemed to have dropped 707 rows from our dataset, but we would like to know more about those. Let's ask which rows the algorithm has dropped:

```
df[df.duplicated()]
```

```
        user_id                                        game    action  freq
2      11373749                   Sid Meier's Civilization IV  purchase   1.0
4      11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
6      11373749          Sid Meier's Civilization IV Warlords  purchase   1.0
10     56038151                  Grand Theft Auto San Andreas  purchase   1.0
12     56038151                    Grand Theft Auto Vice City  purchase   1.0
...         ...                                         ...       ...   ...
1827   39146470          Sid Meier's Civilization IV Warlords  purchase   1.0
1830   48666962                                      Crysis 2  purchase   1.0
1835  112845094                  Grand Theft Auto San Andreas  purchase   1.0
1837  112845094                    Grand Theft Auto Vice City  purchase   1.0
1838  112845094                          Grand Theft Auto III  purchase   1.0

[707 rows x 4 columns]
```

Here we can see the duplicates, no particular pattern seems to be present, we could just for curiosity count the games that are duplicated

```
df[df.duplicated()].game.value_counts()
```

```
Grand Theft Auto San Andreas                       172
Grand Theft Auto Vice City                         103
```

(continues on next page)

(continued from previous page)

```
Sid Meier's Civilization IV                       98
Grand Theft Auto III                              90
Sid Meier's Civilization IV Beyond the Sword      80
Sid Meier's Civilization IV Warlords              79
Sid Meier's Civilization IV Colonization          75
Crysis 2                                           7
Arma 2                                             1
Tom Clancy's H.A.W.X. 2                            1
TERA                                               1
Name: game, dtype: int64
```

It seems there are some games which are very prone to being duplicated, at this point we could go and ask the IT department why these games are acting weird.

Another thing im interested about is the perspective of a single gamer, here we took a single user_id and printed all his games

```
df[df.user_id == 11373749]
```

```
    user_id                                      game    action  freq
0  11373749                   Sid Meier's Civilization IV  purchase   1.0
1  11373749                   Sid Meier's Civilization IV      play   0.1
2  11373749                   Sid Meier's Civilization IV  purchase   1.0
3  11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
4  11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
5  11373749          Sid Meier's Civilization IV Warlords  purchase   1.0
6  11373749          Sid Meier's Civilization IV Warlords  purchase   1.0
```

Ah, you can see all of his three games are somehow duplicated in purchase, also it seems he only played one of them for only 0.1 hours. Looks like he fell to the bait of a tempting summer sale but didn't realise he had no time to actually play it.

Another thing I would like to mention here is that this dataset would make a fine recommender system as it contains user ids and hours played. Add game metadata (description) and reviews to the mix and your data preparation is done!

We can remove all duplicates now by overwriting our dataframe

```
df = df.drop_duplicates()
```

One thing still bothers me, as hours played can change over time it might be that different snapshots have produced different values, therefore more duplicates might be present with different hours_played.

Time to investigate this by using a subset of columns in the drop_duplicates algorithm

```
df.drop_duplicates(subset=['user_id', 'game', 'action'])
```

```
        user_id                                      game    action  freq
0      11373749                   Sid Meier's Civilization IV  purchase   1.0
1      11373749                   Sid Meier's Civilization IV      play   0.1
3      11373749  Sid Meier's Civilization IV Beyond the Sword  purchase   1.0
5      11373749          Sid Meier's Civilization IV Warlords  purchase   1.0
7      56038151                       Tom Clancy's H.A.W.X. 2  purchase   1.0
...         ...                                           ...       ...   ...
1831  112845094                    Grand Theft Auto San Andreas  purchase   1.0
1832  112845094                    Grand Theft Auto San Andreas      play   0.2
1833  112845094                            Grand Theft Auto III  purchase   1.0
1834  112845094                                        Arma 2  purchase   1.0
```

(continues on next page)

(continued from previous page)

```
1836  112845094                    Grand Theft Auto Vice City  purchase   1.0

[1120 rows x 4 columns]
```

Seems we have shaved off another 12 records, so our intuition was right, again lets see which the duplicates are:

```
df[df.duplicated(subset=['user_id', 'game', 'action'])]
```

```
        user_id                                         game action  freq
118   118664413                  Grand Theft Auto San Andreas   play   0.2
458    50769696                  Grand Theft Auto San Andreas   play   3.1
521    71411882                          Grand Theft Auto III   play   0.2
607    33865373                   Sid Meier's Civilization IV   play   2.0
898    71510748                  Grand Theft Auto San Andreas   play   0.2
908    28472068                    Grand Theft Auto Vice City   play   0.4
910    28472068                  Grand Theft Auto San Andreas   play   0.2
912    28472068                          Grand Theft Auto III   play   0.1
1506   59925638                      Tom Clancy's H.A.W.X. 2   play   0.3
1553  148362155                  Grand Theft Auto San Andreas   play  12.5
1709  176261926  Sid Meier's Civilization IV Beyond the Sword   play   0.4
1711  176261926                   Sid Meier's Civilization IV   play   0.2
```

As expected the duplicates are all in the 'play' action, to complete our view we extract the data of a single user

```
df[df.user_id==118664413]
```

```
       user_id                          game    action  freq
115  118664413  Grand Theft Auto San Andreas  purchase   1.0
116  118664413  Grand Theft Auto San Andreas      play   1.9
118  118664413  Grand Theft Auto San Andreas      play   0.2
```

It looks like we have a problem now, we know these are duplicates and should be removed, but which one? Personally I would argue here that we keep the highest value, as it is impossible to 'unplay' hours on the game. I will leave this as an exercise for you, but the solution is pretty tricky so i'll give a hint:

The algorithm always keeps the first record in case of duplicates, so you could sort the rows making sure the higher value is always encountered first, good luck!

CHAPTER

FIVE

OUTLIERS AND VALIDITY

When preparing data we have to be cautious with the accuracy of our set. Outliers and invalid data points are difficult to detect but should be handled with caution.

we start out by importing our most important library.

```
import pandas as pd
```

5.1 Silicon wafer thickness

Our first dataset contains information about the production of silicon wafers, each wafers thickness is measure on 9 different spots. More information on the dataset can be found here.

```
wafer_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c2_data_preparation/data/silicon-wafer-thickness.csv')
wafer_df.head()
```

```
      G1     G2     G3     G4     G5     G6     G7     G8     G9
0  0.175  0.188 -0.159  0.095  0.374 -0.238 -0.800  0.158 -0.211
1  0.102  0.075  0.141  0.180  0.138 -0.057 -0.075  0.072  0.072
2  0.607  0.711  0.879  0.765  0.592  0.187  0.431  0.345  0.187
3  0.774  0.823  0.619  0.370  0.725  0.439 -0.025 -0.259  0.496
4  0.504  0.644  0.845  0.681  0.502  0.151  0.404  0.296  0.260
```

we would like to investigate the distribution of measurements here, as we are early in this course using visualisation techniques would be too soon. This does not mean we can't use simple mathematics, introducing the InterQuartile Range. A reason for using IQR over standard deviation is that with IQR we do not assume a normal distribution. The IQR calculates the range between the bottom 'quart' or 25% and the top 25%, giving us an indication of the spread of our results, we calculate this IQR for each of the 9 measurements independently. For more info about IQR you can visit wikipedia.

```
iqr = wafer_df.quantile(0.75)-wafer_df.quantile(0.25)
iqr
```

```
G1    0.54425
G2    0.61000
G3    0.54075
G4    0.52475
G5    0.61175
G6    0.86750
G7    0.76175
```

(continues on next page)

(continued from previous page)

```
G8    0.87225
G9    0.86300
dtype: float64
```

you can see that the IQR spread for each measurement lays between 0.5 and 1 unit indicating that the 9 measurements of the wafer have a similar spread. With these IQR's we could calculate for each point relative to the spread of the measurement how far it is from the median.

```
relative_spread_df = (wafer_df-wafer_df.median())/iqr
relative_spread_df.head()
```

```
         G1        G2        G3        G4        G5        G6        G7  \
0 -0.011024 -0.077869 -0.819233 -0.367794  0.176543 -0.352738 -1.029865
1 -0.145154 -0.263115 -0.264448 -0.205812 -0.209236 -0.144092 -0.078110
2  0.782729  0.779508  1.100324  0.909004  0.532897  0.137176  0.586150
3  1.089573  0.963115  0.619510  0.156265  0.750306  0.427666 -0.012471
4  0.593477  0.669672  1.037448  0.748928  0.385779  0.095677  0.550706

         G8        G9
0 -0.130696 -0.254925
1 -0.229292  0.073001
2  0.083692  0.206257
3 -0.608770  0.564311
4  0.027515  0.290846
```

You can now see that some points are close to the median, whilst others are much higher, both positive as negative. By defining a threshold, we quantify what deviation has to be there to flag a reading as an outlier. The high outliers are seperated, note that only a single measurement of the 9 can trigger and render the total measurement as an outlier. Yet judging from the setup where we would want to find wafers with varying thickness that approach is desirable.

```
relative_spread_df[(relative_spread_df>2).any(axis='columns')]
```

```
            G1         G2         G3         G4         G5        G6  \
8     2.232430   2.009016   1.956542   1.589328   1.843890  1.544669
38   12.891135  12.827049  12.832178  13.913292  11.429506  9.500865
39    3.691318   3.981148   3.774387   4.081944   3.248059  3.729107
61    2.010106   2.153279   1.987980   1.863745   1.858602  1.274928
110   3.678457   2.841803   3.204808   3.180562   2.669391  0.518732
112   2.361047   2.086066   2.363384   2.107670   1.925623  1.238040
117   1.475425   1.043443   2.154415   2.582182   0.653862  1.823631
120   1.791456   1.484426   2.583449   1.440686   2.085819  0.990202
121   1.791456   1.484426   2.583449   1.440686   2.085819  0.990202
152   2.610932   2.102459   2.387425   2.549786   2.169187  1.730259
154  -0.529169  -0.538525  -0.404993  -0.331586  -0.552513  4.565994

            G7        G8        G9
8     1.233344  0.419604  1.582851
38   10.305875  9.927200  9.055620
39    3.304890  3.846374  3.149479
61    1.237283  0.825451  0.955968
110   0.700361  0.176555  0.727694
112   1.766328  0.890800  1.377752
117   1.581227  0.857552  1.188876
120   1.782081  1.034107  1.822711
121   1.782081  1.034107  1.822711
```

(continues on next page)

(continued from previous page)

```
152   2.241549  1.713958  1.592121
154  -0.051854 -0.382918 -0.536501
```

seems we have a few high outliers, you can clearly see the measurements are mostly all across the board high, but in some cases (e.g. id 154) only one measurement was an outlier. We can do the same for the low outliers.

```
relative_spread_df[(relative_spread_df<-2).any(axis='columns')]
```

```
           G1        G2        G3        G4        G5        G6        G7  \
54  -1.550758 -1.525410 -1.843736 -2.082897 -1.659174 -1.203458 -1.184772
56  -1.732660 -1.510656 -2.121128 -2.122916 -1.781774 -1.521614 -1.909419
59  -1.971520 -1.310656 -2.328248 -1.175798 -2.067838 -0.915274 -1.783394
64  -1.234727 -1.361475 -0.736015 -1.055741 -2.224765 -0.839193 -0.679357
65  -2.226918 -1.194262 -2.117429 -2.161029 -2.043318 -0.190202 -1.004923
102 -2.484153 -2.330328 -1.568192 -2.808957 -1.945239 -1.340634 -0.846078

           G8        G9
54  -1.650903 -1.245655
56  -1.782746 -1.159907
59  -1.304672 -1.514484
64  -0.865578 -0.663963
65  -0.270565 -0.794902
102 -1.691029 -0.887601
```

For a simple mathematical equation these result look promising, yet it can always be more sophisticated. Not going to deep into the subject we could perform some Machine Learning, using a unsupervised method. Here we use the sklearn library which contains the Isolation forest algorithm. More info about the algorithm here.

```
from sklearn.ensemble import IsolationForest
```

We first create the classifier and train (fit) it with the generic wafer data. Then for each record of the wafer data we make a prediction, if it thinks its an outlier, we keep them

```
clf = IsolationForest(random_state=0).fit(wafer_df)
wafer_df[clf.predict(wafer_df)==-1]
```

```
        G1     G2     G3     G4     G5     G6     G7     G8     G9
8    1.396  1.461  1.342  1.122  1.394  1.408  0.924  0.638  1.375
20  -0.558 -0.705 -0.526 -0.412 -0.753 -0.998 -0.270  0.598 -1.416
38   7.197  8.060  7.223  7.589  7.258  8.310  7.835  8.931  7.824
39   2.190  2.664  2.325  2.430  2.253  3.303  2.502  3.627  2.727
54  -0.663 -0.695 -0.713 -0.805 -0.749 -0.976 -0.918 -1.168 -1.066
56  -0.762 -0.686 -0.863 -0.826 -0.824 -1.252 -1.470 -1.283 -0.992
59  -0.892 -0.564 -0.975 -0.329 -0.999 -0.726 -1.374 -0.866 -1.298
61   1.275  1.549  1.359  1.266  1.403  1.174  0.927  0.992  0.834
65  -1.031 -0.493 -0.861 -0.846 -0.984 -0.097 -0.781  0.036 -0.677
102 -1.171 -1.186 -0.564 -1.186 -0.924 -1.095 -0.660 -1.203 -0.757
106 -0.659 -0.451 -0.692 -0.708 -0.595 -0.726 -1.031 -0.877 -1.080
110  2.183  1.969  2.017  1.957  1.899  0.518  0.518  0.426  0.637
112  1.466  1.508  1.562  1.394  1.444  1.142  1.330  1.049  1.198
117  0.984  0.872  1.449  1.643  0.666  1.650  1.189  1.020  1.035
120  1.156  1.141  1.681  1.044  1.542  0.927  1.342  1.174  1.582
121  1.156  1.141  1.681  1.044  1.542  0.927  1.342  1.174  1.582
152  1.602  1.518  1.575  1.626  1.593  1.569  1.692  1.767  1.383
```

Comparing the results with our IQR approach we see a lot of similarities, here the id 154 record did not show up as we already realised this was perhaps not a strong enough outlier. You could enhance our IQR technique by checking the amount of measurements that are above the threshold and respond accordingly, I will leave you a little hint.

```
(relative_spread_df>2).sum()
```

```
G1    7
G2    7
G3    8
G4    6
G5    6
G6    3
G7    3
G8    2
G9    2
dtype: int64
```

5.2 Distillation column

As an exercise you can try the same technique to this dataset and see what you would find, good luck! Be mindful that you do not incorporate the date as a variable in your outlier algorithm.

```
distil_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c2_data_preparation/data/distillation-tower.csv')
distil_df
```

```
           Date     Temp1    FlowC1     Temp2    TempC1     Temp3    TempC2  \
0    2000-08-21  139.9857  432.0636  377.8119  100.2204  492.1353  490.1459
1    2000-08-23  131.0470  487.4029  371.3060  100.2297  482.2100  480.3128
2    2000-08-26  118.2666  437.3516  378.4483  100.3084  488.7266  487.0040
3    2000-08-29  118.1769  481.8314  378.0028   95.5766  493.1481  491.1137
4    2000-08-30  120.7891  412.6471  377.8871   92.9052  490.2486  488.6641
..          ...       ...       ...       ...       ...       ...       ...
248  2003-01-26  130.8138  212.6385  341.5964  121.4354  468.3401  467.0299
249  2003-01-28  128.9673  225.1412  349.8965  118.8604  479.7665  478.4652
250  2003-01-31  130.5328  223.5965  345.9366  120.4027  474.5378  473.1145
251  2003-02-03  128.5248  213.5613  343.4950  119.6989  469.3802  467.9954
252  2003-02-04  131.0491  217.4117  346.1960  119.0825  474.6599  473.0381

       TempC3     Temp4  PressureC1  ...     Temp10  FlowC3   FlowC4   Temp11  \
0    180.5578  187.4331    215.0627  ...   513.9653  8.6279  10.5988  30.8983
1    172.6575  179.5089    205.0999  ...   504.5145  8.7662  10.7560  31.9099
2    165.9400  172.9262    205.0304  ...   508.9997  8.5319  10.5737  29.9165
3    167.2085  174.2338    205.2561  ...   514.1794  8.6260  10.6695  30.6229
4    167.0326  173.9681    205.0883  ...   511.0948  8.5939  10.4922  29.4977
..        ...       ...         ...  ...        ...     ...      ...      ...
248  174.7639  180.7649    229.7393  ...   479.0290  5.5590   6.4470  16.4131
249  176.2176  182.3646    230.5049  ...   491.2362  5.6342   6.4360  17.2385
250  176.3310  182.2578    230.6638  ...   485.8786  5.4810   6.3575  16.9866
251  174.6435  180.5093    230.5226  ...   480.2879  5.4727   6.4175  16.6778
252  177.1088  183.1810    225.6420  ...   486.0253  5.4597   6.3291  16.8766

       Temp12  InvTemp1  InvTemp2  InvTemp3  InvPressure1  VapourPressure
0    489.9900    2.0409    2.6468    2.1681        4.3524         32.5026
1    480.2888    2.0821    2.6932    2.2207        4.5497         34.8598
```

(continues on next page)

(continued from previous page)

```
2    486.6190    2.0550    2.6424    2.1796         4.5511         32.1666
3    491.1304    2.0361    2.6455    2.1620         4.5464         30.4064
4    487.6475    2.0507    2.6463    2.1704         4.5499         30.9238
..        ...       ...       ...       ...            ...             ...
248  466.3347    2.1444    2.9274    2.2127         4.0911         38.8507
249  477.8816    2.0926    2.8580    2.1620         4.0783         34.2653
250  472.3176    2.1172    2.8907    2.1855         4.0756         36.5717
251  467.0001    2.1413    2.9113    2.2090         4.0780         38.1054
252  472.2701    2.1174    2.8885    2.1844         4.1608         35.6298

[253 rows x 28 columns]
```

CHAPTER

SIX

STRING OPERATIONS

CHAPTER

SEVEN

DATETIME OPERATIONS

When our dataset contains time-related data, datetime operations are a great asset to our data science toolkit. For this exercise we obtain a public covid dataset containing A LOT of information on infection cases, deaths, tests and vaccinations.

Let's start by importing the data, as the dataset is about 60MB at the time of writing, this might take some time. Perhaps you could think of a method to make this more efficient, do we always need all of the data?

More info about the data can be found here

```
import pandas as pd
```

```
covid_df = pd.read_csv('https://raw.githubusercontent.com/owid/covid-19-data/master/
↪public/data/owid-covid-data.csv', on_bad_lines='skip')
covid_df.head()
```

```
  iso_code continent     location        date  total_cases  new_cases  \
0      AFG      Asia  Afghanistan  2020-02-24          5.0        5.0
1      AFG      Asia  Afghanistan  2020-02-25          5.0        0.0
2      AFG      Asia  Afghanistan  2020-02-26          5.0        0.0
3      AFG      Asia  Afghanistan  2020-02-27          5.0        0.0
4      AFG      Asia  Afghanistan  2020-02-28          5.0        0.0

   new_cases_smoothed  total_deaths  new_deaths  new_deaths_smoothed  ...  \
0                 NaN           NaN         NaN                  NaN  ...
1                 NaN           NaN         NaN                  NaN  ...
2                 NaN           NaN         NaN                  NaN  ...
3                 NaN           NaN         NaN                  NaN  ...
4                 NaN           NaN         NaN                  NaN  ...

   female_smokers  male_smokers  handwashing_facilities  \
0             NaN           NaN                  37.746
1             NaN           NaN                  37.746
2             NaN           NaN                  37.746
3             NaN           NaN                  37.746
4             NaN           NaN                  37.746

   hospital_beds_per_thousand  life_expectancy  human_development_index  \
0                         0.5            64.83                    0.511
1                         0.5            64.83                    0.511
2                         0.5            64.83                    0.511
3                         0.5            64.83                    0.511
4                         0.5            64.83                    0.511

   excess_mortality_cumulative_absolute  excess_mortality_cumulative  \
```

(continues on next page)

(continued from previous page)

```
0                                    NaN                            NaN
1                                    NaN                            NaN
2                                    NaN                            NaN
3                                    NaN                            NaN
4                                    NaN                            NaN

   excess_mortality  excess_mortality_cumulative_per_million
0               NaN                                      NaN
1               NaN                                      NaN
2               NaN                                      NaN
3               NaN                                      NaN
4               NaN                                      NaN

[5 rows x 65 columns]
```

As mentioned a lot of information is present here, about 65 columns. yet for this exercise my main objective is the 'date' column. If we would print out the data types using the info method, we can see that the date is recognized as an 'object' stating that it is an ordinary string, not a datetime.

```
covid_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 121744 entries, 0 to 121743
Data columns (total 65 columns):
 #   Column                                   Non-Null Count   Dtype
---  ------                                   --------------   -----
 0   iso_code                                 121744 non-null  object
 1   continent                                116202 non-null  object
 2   location                                 121744 non-null  object
 3   date                                     121744 non-null  object
 4   total_cases                              115518 non-null  float64
 5   new_cases                                115515 non-null  float64
 6   new_cases_smoothed                       114500 non-null  float64
 7   total_deaths                             104708 non-null  float64
 8   new_deaths                               104863 non-null  float64
 9   new_deaths_smoothed                      114500 non-null  float64
 10  total_cases_per_million                  114910 non-null  float64
 11  new_cases_per_million                    114907 non-null  float64
 12  new_cases_smoothed_per_million           113897 non-null  float64
 13  total_deaths_per_million                 104113 non-null  float64
 14  new_deaths_per_million                   104268 non-null  float64
 15  new_deaths_smoothed_per_million          113897 non-null  float64
 16  reproduction_rate                        98318 non-null   float64
 17  icu_patients                             14443 non-null   float64
 18  icu_patients_per_million                 14443 non-null   float64
 19  hosp_patients                            16504 non-null   float64
 20  hosp_patients_per_million                16504 non-null   float64
 21  weekly_icu_admissions                    1268 non-null    float64
 22  weekly_icu_admissions_per_million        1268 non-null    float64
 23  weekly_hosp_admissions                   2088 non-null    float64
 24  weekly_hosp_admissions_per_million       2088 non-null    float64
 25  new_tests                                52248 non-null   float64
 26  total_tests                              52352 non-null   float64
 27  total_tests_per_thousand                 52352 non-null   float64
 28  new_tests_per_thousand                   52248 non-null   float64
 29  new_tests_smoothed                       62816 non-null   float64
```

(continues on next page)

(continued from previous page)

```
 30  new_tests_smoothed_per_thousand          62816 non-null   float64
 31  positive_rate                            58959 non-null   float64
 32  tests_per_case                           58319 non-null   float64
 33  tests_units                              64746 non-null   object
 34  total_vaccinations                       28115 non-null   float64
 35  people_vaccinated                        26746 non-null   float64
 36  people_fully_vaccinated                  23714 non-null   float64
 37  total_boosters                           3057 non-null    float64
 38  new_vaccinations                         23298 non-null   float64
 39  new_vaccinations_smoothed                50221 non-null   float64
 40  total_vaccinations_per_hundred           28115 non-null   float64
 41  people_vaccinated_per_hundred            26746 non-null   float64
 42  people_fully_vaccinated_per_hundred      23714 non-null   float64
 43  total_boosters_per_hundred               3057 non-null    float64
 44  new_vaccinations_smoothed_per_million    50221 non-null   float64
 45  stringency_index                         101767 non-null  float64
 46  population                               120880 non-null  float64
 47  population_density                       112501 non-null  float64
 48  median_age                               107423 non-null  float64
 49  aged_65_older                            106229 non-null  float64
 50  aged_70_older                            106834 non-null  float64
 51  gdp_per_capita                           108055 non-null  float64
 52  extreme_poverty                          72482 non-null   float64
 53  cardiovasc_death_rate                    107695 non-null  float64
 54  diabetes_prevalence                      111063 non-null  float64
 55  female_smokers                           84078 non-null   float64
 56  male_smokers                             82858 non-null   float64
 57  handwashing_facilities                   54111 non-null   float64
 58  hospital_beds_per_thousand               97911 non-null   float64
 59  life_expectancy                          115458 non-null  float64
 60  human_development_index                  107790 non-null  float64
 61  excess_mortality_cumulative_absolute     4317 non-null    float64
 62  excess_mortality_cumulative              4317 non-null    float64
 63  excess_mortality                         4317 non-null    float64
 64  excess_mortality_cumulative_per_million  4317 non-null    float64
dtypes: float64(60), object(5)
memory usage: 60.4+ MB
```

We would like to change that, as we can only perform datetime operations if pandas recognises the datetime format used. Good for us, pandas has a method to automatically infer the date format, we do that now.

```
covid_df.date = pd.to_datetime(covid_df.date)
covid_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 121744 entries, 0 to 121743
Data columns (total 65 columns):
 #   Column                                   Non-Null Count   Dtype
---  ------                                   --------------   -----
 0   iso_code                                 121744 non-null  object
 1   continent                                116202 non-null  object
 2   location                                 121744 non-null  object
 3   date                                     121744 non-null  datetime64[ns]
 4   total_cases                              115518 non-null  float64
 5   new_cases                                115515 non-null  float64
 6   new_cases_smoothed                       114500 non-null  float64
```

(continues on next page)

(continued from previous page)

```
 7   total_deaths                              104708 non-null  float64
 8   new_deaths                                104863 non-null  float64
 9   new_deaths_smoothed                       114500 non-null  float64
 10  total_cases_per_million                   114910 non-null  float64
 11  new_cases_per_million                     114907 non-null  float64
 12  new_cases_smoothed_per_million            113897 non-null  float64
 13  total_deaths_per_million                  104113 non-null  float64
 14  new_deaths_per_million                    104268 non-null  float64
 15  new_deaths_smoothed_per_million           113897 non-null  float64
 16  reproduction_rate                         98318 non-null   float64
 17  icu_patients                              14443 non-null   float64
 18  icu_patients_per_million                  14443 non-null   float64
 19  hosp_patients                             16504 non-null   float64
 20  hosp_patients_per_million                 16504 non-null   float64
 21  weekly_icu_admissions                     1268 non-null    float64
 22  weekly_icu_admissions_per_million         1268 non-null    float64
 23  weekly_hosp_admissions                    2088 non-null    float64
 24  weekly_hosp_admissions_per_million        2088 non-null    float64
 25  new_tests                                 52248 non-null   float64
 26  total_tests                               52352 non-null   float64
 27  total_tests_per_thousand                  52352 non-null   float64
 28  new_tests_per_thousand                    52248 non-null   float64
 29  new_tests_smoothed                        62816 non-null   float64
 30  new_tests_smoothed_per_thousand           62816 non-null   float64
 31  positive_rate                             58959 non-null   float64
 32  tests_per_case                            58319 non-null   float64
 33  tests_units                               64746 non-null   object
 34  total_vaccinations                        28115 non-null   float64
 35  people_vaccinated                         26746 non-null   float64
 36  people_fully_vaccinated                   23714 non-null   float64
 37  total_boosters                            3057 non-null    float64
 38  new_vaccinations                          23298 non-null   float64
 39  new_vaccinations_smoothed                 50221 non-null   float64
 40  total_vaccinations_per_hundred            28115 non-null   float64
 41  people_vaccinated_per_hundred             26746 non-null   float64
 42  people_fully_vaccinated_per_hundred       23714 non-null   float64
 43  total_boosters_per_hundred                3057 non-null    float64
 44  new_vaccinations_smoothed_per_million     50221 non-null   float64
 45  stringency_index                          101767 non-null  float64
 46  population                                120880 non-null  float64
 47  population_density                        112501 non-null  float64
 48  median_age                                107423 non-null  float64
 49  aged_65_older                             106229 non-null  float64
 50  aged_70_older                             106834 non-null  float64
 51  gdp_per_capita                            108055 non-null  float64
 52  extreme_poverty                           72482 non-null   float64
 53  cardiovasc_death_rate                     107695 non-null  float64
 54  diabetes_prevalence                       111063 non-null  float64
 55  female_smokers                            84078 non-null   float64
 56  male_smokers                              82858 non-null   float64
 57  handwashing_facilities                    54111 non-null   float64
 58  hospital_beds_per_thousand                97911 non-null   float64
 59  life_expectancy                           115458 non-null  float64
 60  human_development_index                   107790 non-null  float64
 61  excess_mortality_cumulative_absolute      4317 non-null    float64
 62  excess_mortality_cumulative               4317 non-null    float64
 63  excess_mortality                          4317 non-null    float64
```

(continues on next page)

(continued from previous page)

```
 64  excess_mortality_cumulative_per_million  4317 non-null    float64
dtypes: datetime64[ns](1), float64(60), object(4)
memory usage: 60.4+ MB
```

now we are ready to perform datetime operations, however we can see that dates are appearing multiple times, this because we have records for multiple countries. I live in Belgium, so decided to isolate that subsection of the data. If they had used a data lake and partitioned into countries, reading out the data would have been much more efficient, but efficiency is not something I would expect from government as a Belgian.

```
covid_belgium_df = covid_df[covid_df.location=='Belgium'].set_index('date')
covid_belgium_df.head()
```

```
           iso_code continent location  total_cases  new_cases  \
date
2020-02-04      BEL    Europe  Belgium          1.0        1.0
2020-02-05      BEL    Europe  Belgium          1.0        0.0
2020-02-06      BEL    Europe  Belgium          1.0        0.0
2020-02-07      BEL    Europe  Belgium          1.0        0.0
2020-02-08      BEL    Europe  Belgium          1.0        0.0

            new_cases_smoothed  total_deaths  new_deaths  new_deaths_smoothed  \
date
2020-02-04                 NaN           NaN         NaN                  NaN
2020-02-05                 NaN           NaN         NaN                  NaN
2020-02-06                 NaN           NaN         NaN                  NaN
2020-02-07                 NaN           NaN         NaN                  NaN
2020-02-08                 NaN           NaN         NaN                  NaN

            total_cases_per_million  ...  female_smokers  male_smokers  \
date                                 ...
2020-02-04                    0.086  ...            25.1          31.4
2020-02-05                    0.086  ...            25.1          31.4
2020-02-06                    0.086  ...            25.1          31.4
2020-02-07                    0.086  ...            25.1          31.4
2020-02-08                    0.086  ...            25.1          31.4

            handwashing_facilities  hospital_beds_per_thousand  \
date
2020-02-04                     NaN                        5.64
2020-02-05                     NaN                        5.64
2020-02-06                     NaN                        5.64
2020-02-07                     NaN                        5.64
2020-02-08                     NaN                        5.64

            life_expectancy  human_development_index  \
date
2020-02-04            81.63                    0.931
2020-02-05            81.63                    0.931
2020-02-06            81.63                    0.931
2020-02-07            81.63                    0.931
2020-02-08            81.63                    0.931

            excess_mortality_cumulative_absolute  excess_mortality_cumulative  \
date
2020-02-04                                   NaN                          NaN
2020-02-05                                   NaN                          NaN
```

(continues on next page)

(continued from previous page)

```
2020-02-06                                        NaN                             NaN
2020-02-07                                        NaN                             NaN
2020-02-08                                        NaN                             NaN

            excess_mortality  excess_mortality_cumulative_per_million
date
2020-02-04               NaN                                      NaN
2020-02-05               NaN                                      NaN
2020-02-06               NaN                                      NaN
2020-02-07               NaN                                      NaN
2020-02-08               NaN                                      NaN

[5 rows x 64 columns]
```

Now that we have our dataset containing only Belgium I would like to emphasize another aspect, for features such as population density we would not expect a 'head count' to differ each day, and as we can see this number is steady over the whole line (results may vary for those who execute this in the future).

```
covid_belgium_df.population.value_counts()
```

```
11632334.0    611
Name: population, dtype: int64
```

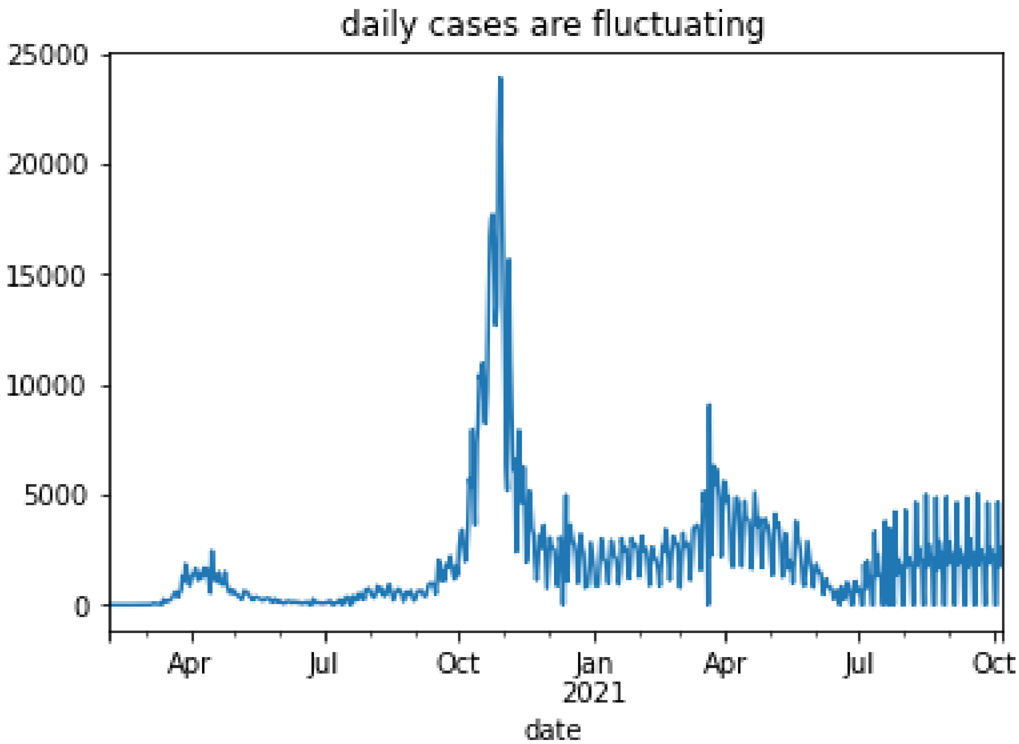
we only have a single value (in my case 11.6M) that is repeated over the whole dataset, would this look optimal to you? How would you perhaps approach this to improve data management? If you would like to go hands-on I left you a blank cell to experiment.

Optimalizations aside, we can not do that which we came for! Datetime operations, the first thing that I have in mind is that due to weekends, the cases might fluctuate a lot per day, so it is not optimal to view it on a daily basis.

First we create a simple line plot with the raw daily cases, then we perform a weekly sum to create a more smooth version of the new cases.

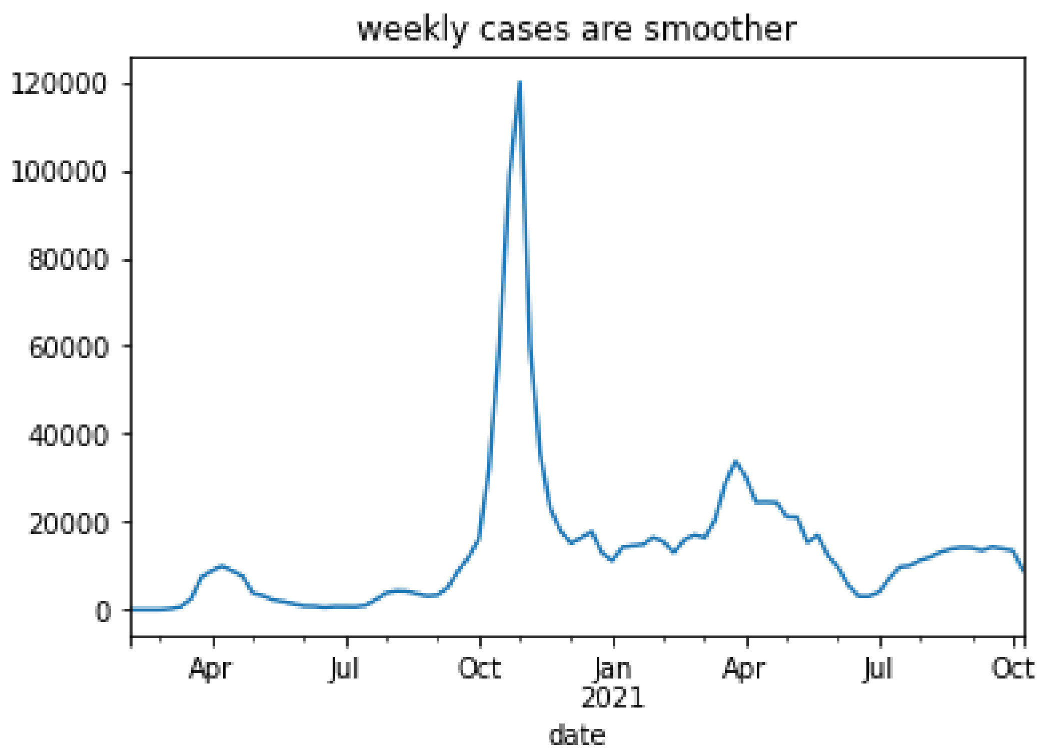
```
covid_belgium_df['new_cases'].plot(title='daily cases are fluctuating')
```

```
<AxesSubplot:title={'center':'daily cases are fluctuating'}, xlabel='date'>
```




```
weekly_cases_df = covid_belgium_df['new_cases'].resample('W').sum()
weekly_cases_df.plot(title='weekly cases are smoother')
```

```
<AxesSubplot:title={'center':'weekly cases are smoother'}, xlabel='date'>
```



That looks great! Those who inspected carefully saw that the x-axis was correclty identified as datetimes and that the y-axis for weekly sums have a much higher range.

In a next example we would like to have the relative changes from week to week, this can be done using the shift operation.

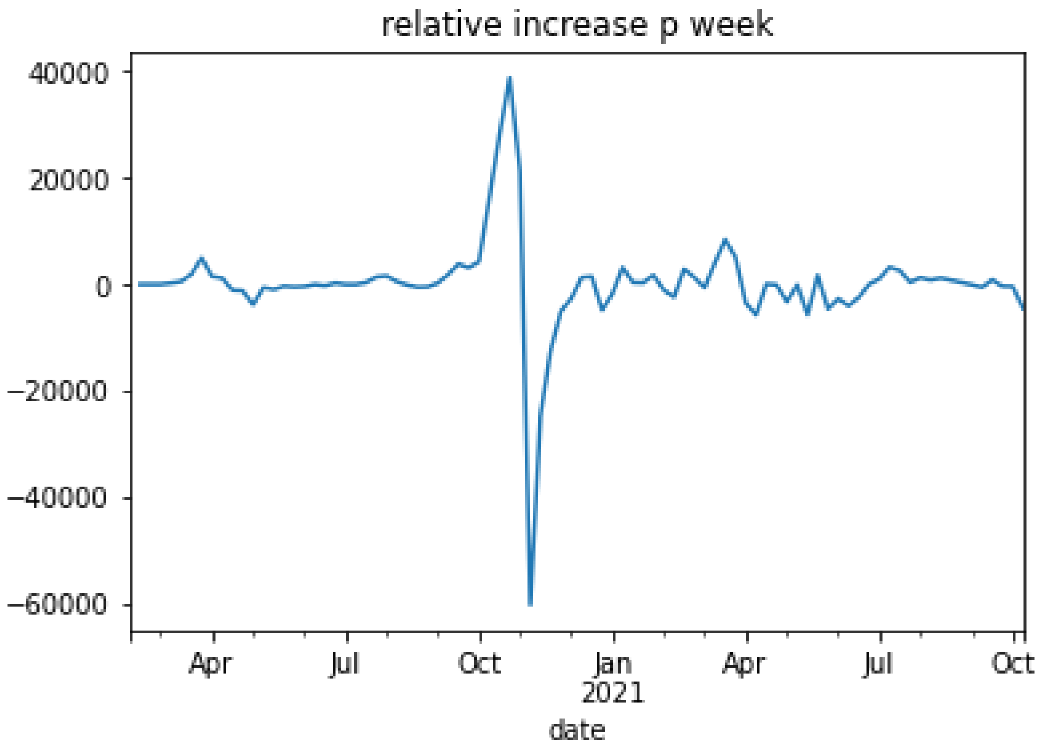
```
weekly_cases_df.shift(1)
```

```
date
2020-02-09        NaN
2020-02-16        1.0
2020-02-23        0.0
2020-03-01        0.0
2020-03-08        1.0
                 ...
2021-09-12    14099.0
2021-09-19    13508.0
2021-09-26    14298.0
2021-10-03    13909.0
2021-10-10    13474.0
Freq: W-SUN, Name: new_cases, Length: 88, dtype: float64
```

This method shifted our data by 1 week forwards, this way we can subtract these results from our original data creating a relative increase (this_week_cases - last_week_cases).

```
(weekly_cases_df-weekly_cases_df.shift(1)).plot(title='relative increase p week')
```

```
<AxesSubplot:title={'center':'relative increase p week'}, xlabel='date'>
```



Another powerfull asset of datetimes is that we can utilize the concepts of days, weeks, months and years. In Belgium they speak about a phenomenon called 'the weekend effect' where a lot of reports are delayed and therefore Sundays have less cases whereas Mondays have more.

Do we see that in our data? let us seperate the Sundays and Mondays and take a mean!

```
print('mean deaths on Monday')
covid_belgium_df.loc[covid_belgium_df.index.dayofweek==0,"new_deaths"].mean()
```

```
mean deaths on Monday
```

```
39.02439024390244
```

```
print('mean deaths on Sunday')
covid_belgium_df.loc[covid_belgium_df.index.dayofweek==6,"new_deaths"].mean()
```

```
mean deaths on Sunday
```

```
36.646341463414636
```

It seems indeed that more people are reported to pass away no a Monday than on a Sunday, it would be optimal to verify this with statistics, but for now we keep it simple.

As a last example I would like to use slicing of our dataset to demonstrate we can also take a subset of our data and handle this, here we took the months of dec2020-jan2021 for belgium and calculated the total deaths.

```
covid_belgium_df.loc['2020-12-01':'2021-01-31'].new_deaths.sum()
```

```
4447.0
```

Now let's compare this to our neighbours, the Netherlands and France, we do exactly the same operations by selecting exaclty the same time window.

```
covid_netherlands_df = covid_df[covid_df.location=='Netherlands'].set_index('date')
covid_netherlands_df.loc['2020-12-01':'2021-01-31'].new_deaths.sum()
```

```
4655.0
```

```
covid_france_df = covid_df[covid_df.location=='France'].set_index('date')
covid_france_df.loc['2020-12-01':'2021-01-31'].new_deaths.sum()
```

```
23382.0
```

You can see that Belgium has the lowest of total deaths in that time interval, so you could assume we performed the best! However this approach is a bit simplified as there are not as many Belgians as French and Dutch. Could you perhaps think if an improvement to create a better understanding?

CHAPTER EIGHT

CATEGORICAL ENCODING

Often we deal with categorical data and this kind of data is something computer algorithms are not able to understand. On the other hand long categorical features might take up unnecessary memory in our dataset, so converting to a categorical feature is optimal.

```
import pandas as pd
```

8.1 Raw Material Charaterization

In this dataset, we have a few numerical features describing characteristics of our material, next to that we also have an Outcome feature describing the state of our material in a category.

Let's have a look at the data

```
raw_material_df = pd.read_csv('./data/raw-material-characterization.csv')
raw_material_df.head()
```

```
  Lot number   Outcome  Size5  Size10  Size15    TGA   DSC   TMA
0       B370  Adequate   13.8     9.2    41.2  787.3  18.0  65.0
1       B880  Adequate   11.2     5.8    27.6  772.2  17.7  68.8
2       B452  Adequate    9.9     5.8    28.3  602.3  18.3  50.7
3       B287  Adequate   10.4     4.0    24.7  677.9  17.7  56.5
4       B576  Adequate   12.3     9.3    22.0  593.5  19.5  52.0
```

So we can see that the outcome is indeed a text field with a human interpretable value. The different values are:

```
raw_material_df.Outcome.unique()
```

```
array(['Adequate', 'Poor'], dtype=object)
```

Image that we would like to get all records where the Outcome is less than adequate, using strings this is not possible as the computer does not understand relations of Adequate and Poor when they are denoted as text.

```
raw_material_df[raw_material_df.Outcome<'Adequate']
```

```
Empty DataFrame
Columns: [Lot number, Outcome, Size5, Size10, Size15, TGA, DSC, TMA]
Index: []
```

To overcome this we can change the type of the feature from 'object' (string) to 'category' let us look at the data types of our data

```
raw_material_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24 entries, 0 to 23
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Lot number  24 non-null     object
 1   Outcome     24 non-null     object
 2   Size5       24 non-null     float64
 3   Size10      24 non-null     float64
 4   Size15      24 non-null     float64
 5   TGA         24 non-null     float64
 6   DSC         24 non-null     float64
 7   TMA         24 non-null     float64
dtypes: float64(6), object(2)
memory usage: 1.6+ KB
```

Now we can change that of Outcome to category using the astype method

```
raw_material_df.Outcome = raw_material_df.Outcome.astype('category')
raw_material_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 24 entries, 0 to 23
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Lot number  24 non-null     object
 1   Outcome     24 non-null     category
 2   Size5       24 non-null     float64
 3   Size10      24 non-null     float64
 4   Size15      24 non-null     float64
 5   TGA         24 non-null     float64
 6   DSC         24 non-null     float64
 7   TMA         24 non-null     float64
dtypes: category(1), float64(6), object(1)
memory usage: 1.6+ KB
```

Our feature might be of categorical nature now, however we still have to define it is an ordinal category and has an order.

```
raw_material_df.Outcome = raw_material_df.Outcome.cat.as_ordered().cat.reorder_
↪categories(['Poor', 'Adequate'])
```

If we retry to effort to only take the records where the Outcome is less than Adequate, we now get an outcome! Since we only have 2 categories this is a bit unfortunate, but you should get the idea behind it.

```
raw_material_df[raw_material_df.Outcome<'Adequate']
```

```
   Lot number Outcome  Size5  Size10  Size15    TGA   DSC   TMA
5        B914    Poor   13.7     7.8    27.0  597.9  18.1  49.8
6        B404    Poor   15.5    10.7    34.3  668.5  19.6  55.7
7        B694    Poor   15.4    10.7    35.9  602.8  19.2  53.6
8        B875    Poor   14.9    11.3    41.0  614.6  18.5  50.0
10       B517    Poor   16.1    11.6    39.2  682.8  17.5  56.4
```

(continues on next page)

(continued from previous page)

```
13        B430    Poor    12.9      9.7     36.3  642.4  19.1  55.0
21        B745    Poor    10.2      5.8     24.7  575.9  18.5  46.2
```

Let's take this a step further, since computer algorithms still have no idea what the numerical relation is between Adequate and Poor, we could use a Label Encoder for that.

```
from sklearn.preprocessing import LabelEncoder
```

the label encoder is inputted with the Outcome feature and recognises 2 types, it chooses a numerical value for each while fitting.

```
le = LabelEncoder()
le.fit(raw_material_df.Outcome)
```

```
LabelEncoder()
```

After fitting we can use this encoder to transform our dataset!

```
raw_material_df['outcome_label'] = le.transform(raw_material_df.Outcome)
raw_material_df.head()
```

```
  Lot number   Outcome  Size5  Size10  Size15    TGA   DSC   TMA  \
0       B370  Adequate   13.8     9.2    41.2  787.3  18.0  65.0
1       B880  Adequate   11.2     5.8    27.6  772.2  17.7  68.8
2       B452  Adequate    9.9     5.8    28.3  602.3  18.3  50.7
3       B287  Adequate   10.4     4.0    24.7  677.9  17.7  56.5
4       B576  Adequate   12.3     9.3    22.0  593.5  19.5  52.0

   outcome_label
0              0
1              0
2              0
3              0
4              0
```

It seems something unfortunate has happened, the encoder gave the Adequate an outcome label of 0, which is lower than the label of Poor (1), this might be bad if we would like to give a score as our outcome.

There is in pandas another method of mapping a label to a category albeit less automated, as you would have to know the categories in your feature.

```
raw_material_df.outcome_label = raw_material_df.Outcome.map({'Poor': 0, 'Adequate':1})
raw_material_df.head()
```

```
  Lot number   Outcome  Size5  Size10  Size15    TGA   DSC   TMA outcome_label
0       B370  Adequate   13.8     9.2    41.2  787.3  18.0  65.0             1
1       B880  Adequate   11.2     5.8    27.6  772.2  17.7  68.8             1
2       B452  Adequate    9.9     5.8    28.3  602.3  18.3  50.7             1
3       B287  Adequate   10.4     4.0    24.7  677.9  17.7  56.5             1
4       B576  Adequate   12.3     9.3    22.0  593.5  19.5  52.0             1
```

Yes! This did the trick, now we can use that outcome label to predict an outcome for future samples.

CHAPTER

NINE

RESTAURANT TIPS

Now we are going to look at a dataset of tips, here a restaurant tracked the table bills and tips for a few days in the week whilst also noting the gender, smoking habit and time of day. This led to a small yet very interesting dataset, let's have a look!

```
tips_df = pd.read_csv('https://raw.githubusercontent.com/mwaskom/seaborn-data/master/
↪tips.csv')
tips_df
```

```
     total_bill   tip     sex smoker   day    time  size
0         16.99  1.01  Female     No   Sun  Dinner     2
1         10.34  1.66    Male     No   Sun  Dinner     3
2         21.01  3.50    Male     No   Sun  Dinner     3
3         23.68  3.31    Male     No   Sun  Dinner     2
4         24.59  3.61  Female     No   Sun  Dinner     4
..          ...   ...     ...    ...   ...     ...   ...
239       29.03  5.92    Male     No   Sat  Dinner     3
240       27.18  2.00  Female    Yes   Sat  Dinner     2
241       22.67  2.00    Male    Yes   Sat  Dinner     2
242       17.82  1.75    Male     No   Sat  Dinner     2
243       18.78  3.00  Female     No  Thur  Dinner     2

[244 rows x 7 columns]
```

We can see here that we have a lot of categorical variables: gender, smoker, the day and the time. In later sections we will see how we can aggregate on these categorical variables. Now however we would like to process them for a machine learning exercise, where we need numbers not text. For the features smoker and day, you could argue there is a clear numbering between them, smoking is 1 if the person was smoking and e.g. Sun relates to 7 as it is the seventh day of the week.

But for the gender this is different, we can't really say that women are 1 and Men are 0 or vice versa (although in this binary case it might work). The same theory applies for time, if we would say that breakfast, lunch and dinner equal to 0, 1 and 2 this would give our algorithm a bad impression as it would think dinner is twice lunch…

We use One Hot Encoding for this, the idea is that for each value of the feature we create a new column.

```
from sklearn.preprocessing import OneHotEncoder
```

First we create our encoder, then we give it the day column to learn and see which values of categories there are.

```
ohe = OneHotEncoder()
ohe.fit(tips_df[['day']])
```

```
OneHotEncoder()
```

Now we can perform an encoding, here we insert the day column and it returns a matrix with 4 columns corresponding to the 4 days in our feature.

```
ohe.transform(tips_df[['day']]).todense()
```

```
matrix([[0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
        [0., 0., 1., 0.],
```

(continues on next page)

(continued from previous page)

```
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
```

(continues on next page)

(continued from previous page)

```
       [0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
```

(continues on next page)

(continued from previous page)

```
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 0., 1.]])
```

As this is a rather mathematical approach for this simple problem I prefer to use the pandas approach for this, which is the get_dummies method. The outcome is much more pleasing yet completely the same.

```
pd.get_dummies(tips_df.day)
```

```
     Fri  Sat  Sun  Thur
0      0    0    1     0
1      0    0    1     0
2      0    0    1     0
3      0    0    1     0
4      0    0    1     0
..   ...  ...  ...   ...
239    0    1    0     0
240    0    1    0     0
241    0    1    0     0
242    0    1    0     0
243    0    0    0     1

[244 rows x 4 columns]
```

As an exercise you could create a script that given a specific feature (e.g. day):

- extracts that feature
- creates dummies
- concattenates it to the dataframe

Good luck!

CHAPTER TEN

SCALING AND NORMALIZATION

In this notebook we are going to look into 2 rather mathematical concepts, Scaling and Normalization. The idea is that we can scale the values and shape the distribution of feature in our dataset.

As an example we take a dataset containing samples from a low density polyethylene production process, containing several numerical features such as temperatures, Forces, Pressure,…

The idea is that by using Scaling and normalization, the 'range of motion' for these sensors is equal and we can compare the fluxtuations not only inbetween records, but also inbetween sensors.

```
import pandas as pd
```

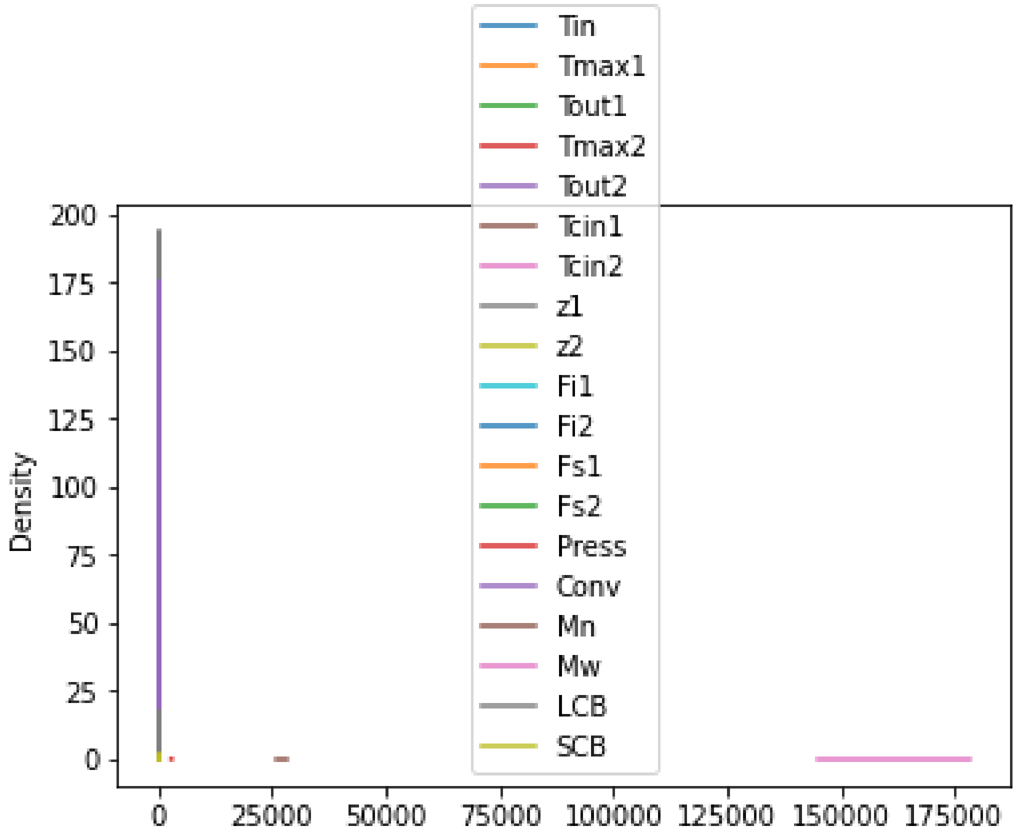
```
ldpe_df = pd.read_csv('https://openmv.net/file/LDPE.csv').drop(columns=['Unnamed: 0'])
ldpe_df.head()
```

```
      Tin   Tmax1   Tout1   Tmax2   Tout2   Tcin1   Tcin2     z1     z2  \
0  208.17  296.35  233.81  283.41  239.05  117.14  117.20  0.029  0.581
1  207.26  298.26  230.88  287.55  242.55  116.39  117.23  0.028  0.574
2  205.30  296.57  235.38  284.35  245.19  117.33  118.42  0.031  0.578
3  209.29  294.11  225.61  283.31  242.04  116.15  117.94  0.030  0.581
4  206.76  295.13  230.26  283.74  244.92  116.75  118.49  0.030  0.579

      Fi1     Fi2     Fs1     Fs2  Press    Conv     Mn      Mw    LCB    SCB
0  0.4507  0.4518  666.42  248.95   3021  0.1322  27379  160326  0.781  26.11
1  0.4765  0.5091  658.61  246.36   3033  0.1365  27043  165044  0.819  26.29
2  0.4744  0.4505  666.51  244.65   3004  0.1335  27344  165621  0.801  26.13
3  0.4429  0.4516  667.31  242.28   2980  0.1300  27502  160497  0.778  25.92
4  0.4394  0.4414  670.83  244.31   2997  0.1316  27518  165713  0.786  26.02
```

We can see that our features clearly have different ranges, but lets try to visualise these ranges using a density plot

```
ldpe_df.plot(kind='density')
```

```
<AxesSubplot:ylabel='Density'>
```

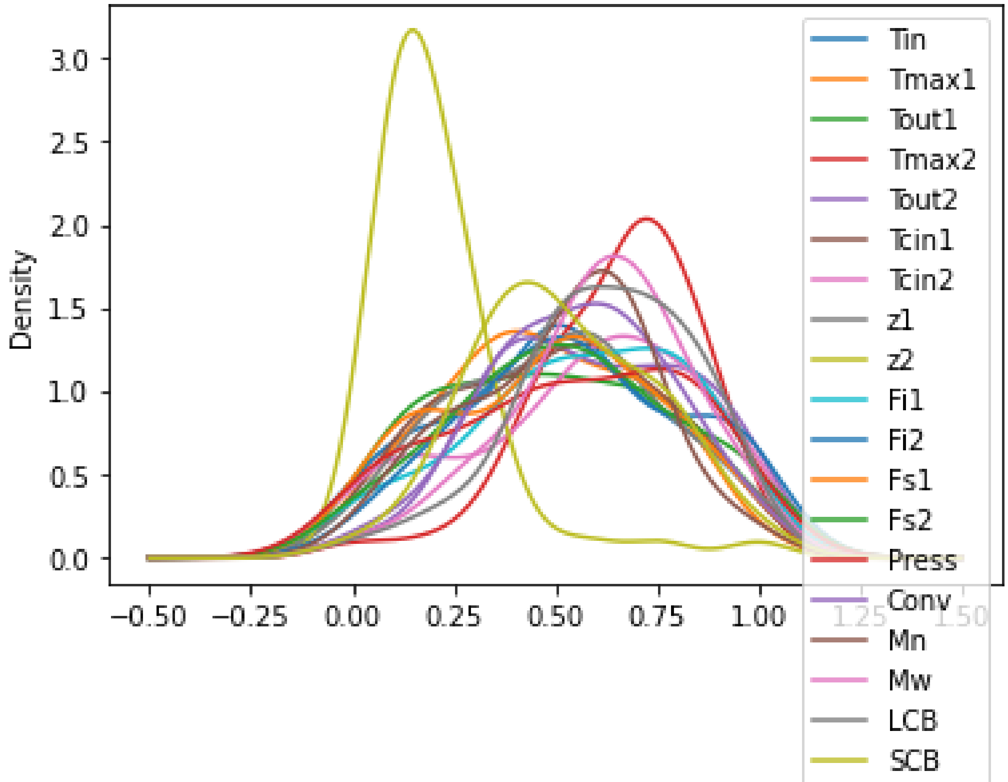


Ouch, this is clearly not working! Because the 'Mw' feature is in the range of 150k-175k our plot is so dilluted the rest are pinned to 0. We can use the sklearn library to perform a min max scaling, this scaling will shift the distribution of each feature between 0 and 1, but that can also be adjusted.

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
scaler.fit(ldpe_df)
pd.DataFrame(scaler.transform(ldpe_df), columns=ldpe_df.columns).plot(kind='density')
```

```
<AxesSubplot:ylabel='Density'>
```



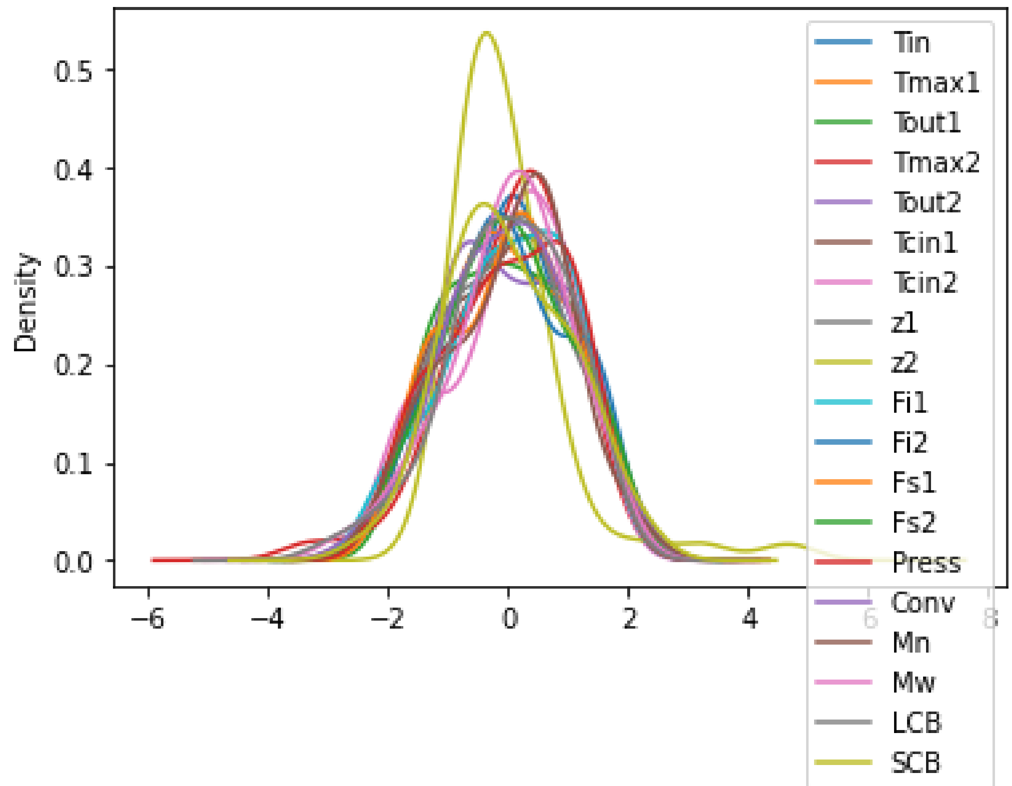
That makes a lot more sense, you can now see all of the distribution at once. Also there seems to be one (yellow) feature that has some outliers perhaps something weird is going on there…

Taking it a step further we could also alter the distributions by using a standard scaler instead of a min max scaler, redistributing the values mathematically into a normal distribution.

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
scaler.fit(ldpe_df)
pd.DataFrame(scaler.transform(ldpe_df), columns=ldpe_df.columns).plot(kind='density')
```

```
<AxesSubplot:ylabel='Density'>
```



You can see it had some trouble fitting our special feature into the normal distribution but it did work out in the end. With this we are ready to perform machine learning algorithms on this data, but first why not try and figure out where those outliers are I mentioned earlier?

CHAPTER

ELEVEN

BINNING AND RANKING

When dealing with numerical data the trouble can sometimes be that numbers can have a wide variety.

Here we apply 2 methods to deal with that, binning and ranking. With binning we change the numerical feature into a categorical/ordinal feature. Ranking is used when our numerical feature contains a non normal distribution that fails to be normalized.

For this example we use a food consumption dataset, where european countries are listed and the relative percentage of each country is given that consumes the type of food, e.g. a value of 67 means that 67% of that country eats that type of food.

```
import pandas as pd
pd.set_option('display.max_columns', None)
```

```
food_df = pd.read_csv('https://openmv.net/file/food-consumption.csv')
food_df
```

```
        Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
0       Germany           90              49   88       19.0      57.0
1         Italy           82              10   60        2.0      55.0
2        France           88              42   63        4.0      76.0
3       Holland           96              62   98       32.0      62.0
4       Belgium           94              38   48       11.0      74.0
5    Luxembourg           97              61   86       28.0      79.0
6       England           27              86   99       22.0      91.0
7      Portugal           72              26   77        2.0      22.0
8       Austria           55              31   61       15.0      29.0
9   Switzerland           73              72   85       25.0      31.0
10       Sweden           97              13   93       31.0       NaN
11      Denmark           96              17   92       35.0      66.0
12       Norway           92              17   83       13.0      62.0
13      Finland           98              12   84       20.0      64.0
14        Spain           70              40   40        NaN      62.0
15      Ireland           30              52   99       11.0      80.0

    Powder soup  Tin soup  Potatoes  Frozen fish  Frozen veggies  Apples  \
0            51        19        21           27              21      81
1            41         3         2            4               2      67
2            53        11        23           11               5      87
3            67        43         7           14              14      83
4            37        23         9           13              12      76
5            73        12         7           26              23      85
6            55        76        17           20              24      76
7            34         1         5           20               3      22
```

(continues on next page)

(continued from previous page)

```
8            33          1         5           15              11      49
9            69         10        17           19              15      79
10           43         43        39           54              45      56
11           32         17        11           51              42      81
12           51          4        17           30              15      61
13           27         10         8           18              12      50
14           43          2        14           23               7      59
15           75         18         2            5               3      57

    Oranges  Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
0        75            44   71      22      91         85         74     30.0
1        71             9   46      80      66         24         94      5.0
2        84            40   45      88      94         47         36     57.0
3        89            61   81      15      31         97         13     53.0
4        76            42   57      29      84         80         83     20.0
5        94            83   20      91      94         94         84     31.0
6        68            89   91      11      95         94         57     11.0
7        51             8   16      89      65         78         92      6.0
8        42            14   41      51      51         72         28     13.0
9        70            46   61      64      82         48         61     48.0
10       78            53   75       9      68         32         48      2.0
11       72            50   64      11      92         91         30     11.0
12       72            34   51      11      63         94         28      2.0
13       57            22   37      15      96         94         17      NaN
14       77            30   38      86      44         51         91     16.0
15       52            46   89       5      97         25         31      3.0

    Crisp bread
0            26
1            18
2             3
3            15
4             5
5            24
6            28
7             9
8            11
9            30
10           93
11           34
12           62
13           64
14           13
15            9
```

Here you could do some data validity, where we check if all values are between 0 and 100, or we check for missing values. I will leave that up to you

11.1 Binning

the first thing we want to do is seperate the countries based on their coffee consumption, instead of creating arbitrary values we can perform a quantitative cut. This means we create a number of equally sized groups using the qcut function, we give them the labels low, medium and high.

```
food_df['bin_coffee'] = pd.qcut(food_df['Real coffee'], q=3, labels=['low', 'medium',
↪'high'])
food_df
```

```
        Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
0       Germany           90              49   88       19.0      57.0
1         Italy           82              10   60        2.0      55.0
2        France           88              42   63        4.0      76.0
3       Holland           96              62   98       32.0      62.0
4       Belgium           94              38   48       11.0      74.0
5    Luxembourg           97              61   86       28.0      79.0
6       England           27              86   99       22.0      91.0
7      Portugal           72              26   77        2.0      22.0
8       Austria           55              31   61       15.0      29.0
9   Switzerland           73              72   85       25.0      31.0
10       Sweden           97              13   93       31.0       NaN
11      Denmark           96              17   92       35.0      66.0
12       Norway           92              17   83       13.0      62.0
13      Finland           98              12   84       20.0      64.0
14        Spain           70              40   40        NaN      62.0
15      Ireland           30              52   99       11.0      80.0

    Powder soup  Tin soup  Potatoes  Frozen fish  Frozen veggies  Apples  \
0            51        19        21           27              21      81
1            41         3         2            4               2      67
2            53        11        23           11               5      87
3            67        43         7           14              14      83
4            37        23         9           13              12      76
5            73        12         7           26              23      85
6            55        76        17           20              24      76
7            34         1         5           20               3      22
8            33         1         5           15              11      49
9            69        10        17           19              15      79
10           43        43        39           54              45      56
11           32        17        11           51              42      81
12           51         4        17           30              15      61
13           27        10         8           18              12      50
14           43         2        14           23               7      59
15           75        18         2            5               3      57

    Oranges  Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
0        75            44   71      22      91         85         74     30.0
1        71             9   46      80      66         24         94      5.0
2        84            40   45      88      94         47         36     57.0
3        89            61   81      15      31         97         13     53.0
4        76            42   57      29      84         80         83     20.0
5        94            83   20      91      94         94         84     31.0
6        68            89   91      11      95         94         57     11.0
7        51             8   16      89      65         78         92      6.0
8        42            14   41      51      51         72         28     13.0
9        70            46   61      64      82         48         61     48.0
```

(continues on next page)

(continued from previous page)

```
10       78             53  75       9      68          32          48      2.0
11       72             50  64      11      92          91          30     11.0
12       72             34  51      11      63          94          28      2.0
13       57             22  37      15      96          94          17      NaN
14       77             30  38      86      44          51          91     16.0
15       52             46  89       5      97          25          31      3.0

    Crisp bread bin_coffee
0            26     medium
1            18     medium
2             3     medium
3            15       high
4             5     medium
5            24       high
6            28        low
7             9        low
8            11        low
9            30        low
10           93       high
11           34       high
12           62     medium
13           64       high
14           13        low
15            9        low
```

a new column has appeared at the end of our dataframe, containing the labels of our binning, countries with low coffee consumption are put in the low category and vice versa. Now we can seperate the countries with low coffee consumption from the rest

```
food_df[food_df.bin_coffee == 'low']
```

```
        Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
6       England           27              86   99       22.0      91.0
7      Portugal           72              26   77        2.0      22.0
8       Austria           55              31   61       15.0      29.0
9   Switzerland           73              72   85       25.0      31.0
14        Spain           70              40   40        NaN      62.0
15      Ireland           30              52   99       11.0      80.0

    Powder soup  Tin soup  Potatoes  Frozen fish  Frozen veggies  Apples  \
6            55        76        17           20              24      76
7            34         1         5           20               3      22
8            33         1         5           15              11      49
9            69        10        17           19              15      79
14           43         2        14           23               7      59
15           75        18         2            5               3      57

    Oranges  Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
6        68            89   91      11      95         94         57     11.0
7        51             8   16      89      65         78         92      6.0
8        42            14   41      51      51         72         28     13.0
9        70            46   61      64      82         48         61     48.0
14       77            30   38      86      44         51         91     16.0
15       52            46   89       5      97         25         31      3.0

    Crisp bread bin_coffee
```

(continues on next page)

(continued from previous page)

```
6             28         low
7              9         low
8             11         low
9             30         low
14            13         low
15             9         low
```

You can already see the England/Ireland stereotype here, note that those are the only 2 with really low coffee consumption, the others are only in this low binning because we requested equally spaced bins in our qcut function. using the cut function would result in a different outcome. Perhaps you could try that out?

I tried to think of some metric to quantify the status of coffee drinkers, since we also have the instant coffee consumption we could create a metric where we subtract the amount of instant coffe drinkers from the amount of real coffee drinkers. This way we can measure that difference between them, I already went ahead and made equal quantity bins for them with labels low, medium and high 'quality coffee'.

```
food_df['bin_qual_coffee'] = pd.qcut(food_df['Real coffee'] - food_df['Instant coffee
↪'], q=3, labels=['low', 'medium', 'high'])
```

```
food_df[food_df.bin_qual_coffee=='high']
```

```
    Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
1     Italy           82              10   60        2.0      55.0
10   Sweden           97              13   93       31.0       NaN
11  Denmark           96              17   92       35.0      66.0
12   Norway           92              17   83       13.0      62.0
13  Finland           98              12   84       20.0      64.0

    Powder soup  Tin soup  Potatoes  Frozen fish  Frozen veggies  Apples  \
1            41         3         2            4               2      67
10           43        43        39           54              45      56
11           32        17        11           51              42      81
12           51         4        17           30              15      61
13           27        10         8           18              12      50

    Oranges  Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
1        71             9   46      80      66         24         94      5.0
10       78            53   75       9      68         32         48      2.0
11       72            50   64      11      92         91         30     11.0
12       72            34   51      11      63         94         28      2.0
13       57            22   37      15      96         94         17      NaN

    Crisp bread bin_coffee bin_qual_coffee
1            18     medium            high
10           93       high            high
11           34       high            high
12           62     medium            high
13           64       high            high
```

Aha! you can see here which countries prefer the real coffee over the instant version. It seems the scandinavian countries together with obviously Italy are the true Caffeine connoisseur of Europe. Another intersting thing we can do now is take the mean for each product for both group high and low and take the difference for high - low. We can see the result below

```
food_df[food_df.bin_qual_coffee=='high'].mean()-food_df[food_df.bin_qual_coffee=='low
↪'].mean()
```

```
/tmp/ipykernel_16521/3908782487.py:1: FutureWarning: Dropping of nuisance columns in␣
↪DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version␣
↪this will raise TypeError.  Select only valid columns before calling the reduction.
  food_df[food_df.bin_qual_coffee=='high'].mean()-food_df[food_df.bin_qual_coffee==
↪'low'].mean()
```

```
Real coffee        34.500000
Instant coffee    -43.366667
Tea                 2.066667
Sweetener          -0.800000
Biscuits            2.583333
Powder soup       -18.200000
Tin soup           -9.600000
Potatoes            5.066667
Frozen fish        15.400000
Frozen veggies     10.866667
Apples             -4.166667
Oranges             3.666667
Tinned fruit      -14.066667
Jam               -12.233333
Garlic            -13.466667
Butter             10.333333
Margarine           2.500000
Olive oil          -3.433333
Yoghurt           -19.000000
Crisp bread        36.533333
dtype: float64
```

It seems a preference for quality coffee also pairs with crisp bread, who knew? Do you think scaling/normalization might be interesting here? why (not)?

11.2 Ranking

In case normalization fails us and we are for some reason not able to get a normal distribution out of a feature, we can still resort to ranking. Note that non linear machine learning techniques often use a ranking functionality under the hood, therefore this technique is often not required, yet for educational purposes we are going to use it here anyway. Let's see how the distribution for Real coffee consumption looks like.

```
food_df.sort_values('Real coffee')
```

```
        Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
6       England           27              86   99       22.0      91.0
15      Ireland           30              52   99       11.0      80.0
8       Austria           55              31   61       15.0      29.0
14        Spain           70              40   40        NaN      62.0
7      Portugal           72              26   77        2.0      22.0
9   Switzerland           73              72   85       25.0      31.0
1         Italy           82              10   60        2.0      55.0
2        France           88              42   63        4.0      76.0
0       Germany           90              49   88       19.0      57.0
12       Norway           92              17   83       13.0      62.0
4       Belgium           94              38   48       11.0      74.0
3       Holland           96              62   98       32.0      62.0
11      Denmark           96              17   92       35.0      66.0
```

(continues on next page)

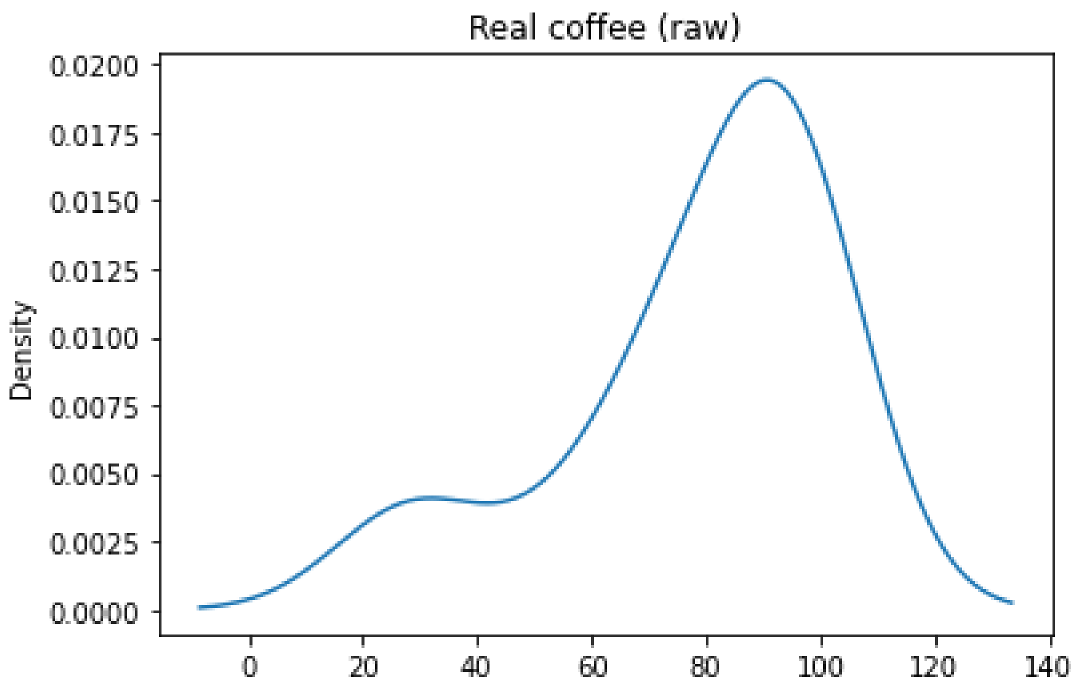
(continued from previous page)

```
5    Luxembourg          97              61   86       28.0      79.0
10       Sweden          97              13   93       31.0       NaN
13      Finland          98              12   84       20.0      64.0

   Powder soup  Tin soup  Potatoes  Frozen fish  Frozen veggies  Apples  \
6           55        76        17           20              24      76
15          75        18         2            5               3      57
8           33         1         5           15              11      49
14          43         2        14           23               7      59
7           34         1         5           20               3      22
9           69        10        17           19              15      79
1           41         3         2            4               2      67
2           53        11        23           11               5      87
0           51        19        21           27              21      81
12          51         4        17           30              15      61
4           37        23         9           13              12      76
3           67        43         7           14              14      83
11          32        17        11           51              42      81
5           73        12         7           26              23      85
10          43        43        39           54              45      56
13          27        10         8           18              12      50

   Oranges  Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
6       68            89   91      11      95         94         57     11.0
15      52            46   89       5      97         25         31      3.0
8       42            14   41      51      51         72         28     13.0
14      77            30   38      86      44         51         91     16.0
7       51             8   16      89      65         78         92      6.0
9       70            46   61      64      82         48         61     48.0
1       71             9   46      80      66         24         94      5.0
2       84            40   45      88      94         47         36     57.0
0       75            44   71      22      91         85         74     30.0
12      72            34   51      11      63         94         28      2.0
4       76            42   57      29      84         80         83     20.0
3       89            61   81      15      31         97         13     53.0
11      72            50   64      11      92         91         30     11.0
5       94            83   20      91      94         94         84     31.0
10      78            53   75       9      68         32         48      2.0
13      57            22   37      15      96         94         17      NaN

   Crisp bread bin_coffee bin_qual_coffee
6           28        low             low
15           9        low             low
8           11        low             low
14          13        low             low
7            9        low          medium
9           30        low             low
1           18     medium            high
2            3     medium          medium
0           26     medium          medium
12          62     medium            high
4            5     medium          medium
3           15       high             low
11          34       high            high
5           24       high          medium
10          93       high            high
13          64       high            high
```

Ah yes, about half of the values are 90 or higher, not really optimal as the range is between 0 and 100! We can also view this in a visual way using a density plot.

```
food_df['Real coffee'].plot(kind='density', title='Real coffee (raw)')
```

```
<AxesSubplot:title={'center':'Real coffee (raw)'}, ylabel='Density'>
```



For larger datasets this would be more useful as we cannot see our whole dataset, it is clear we have to do something about this, now imagine we can not use regular normalization techniques. The rank method now comes in handy!

```
food_df['rank_coffee'] = food_df['Real coffee'].rank()
food_df
```

```
        Country  Real coffee  Instant coffee  Tea  Sweetener  Biscuits  \
0       Germany           90              49   88       19.0      57.0
1         Italy           82              10   60        2.0      55.0
2        France           88              42   63        4.0      76.0
3       Holland           96              62   98       32.0      62.0
4       Belgium           94              38   48       11.0      74.0
5    Luxembourg           97              61   86       28.0      79.0
6       England           27              86   99       22.0      91.0
7      Portugal           72              26   77        2.0      22.0
8       Austria           55              31   61       15.0      29.0
9   Switzerland           73              72   85       25.0      31.0
10       Sweden           97              13   93       31.0       NaN
11      Denmark           96              17   92       35.0      66.0
12       Norway           92              17   83       13.0      62.0
13      Finland           98              12   84       20.0      64.0
14        Spain           70              40   40        NaN      62.0
15      Ireland           30              52   99       11.0      80.0

    Powder soup  Tin soup  Potatoes  Frozen fish  Frozen veggies  Apples  \
```

(continues on next page)

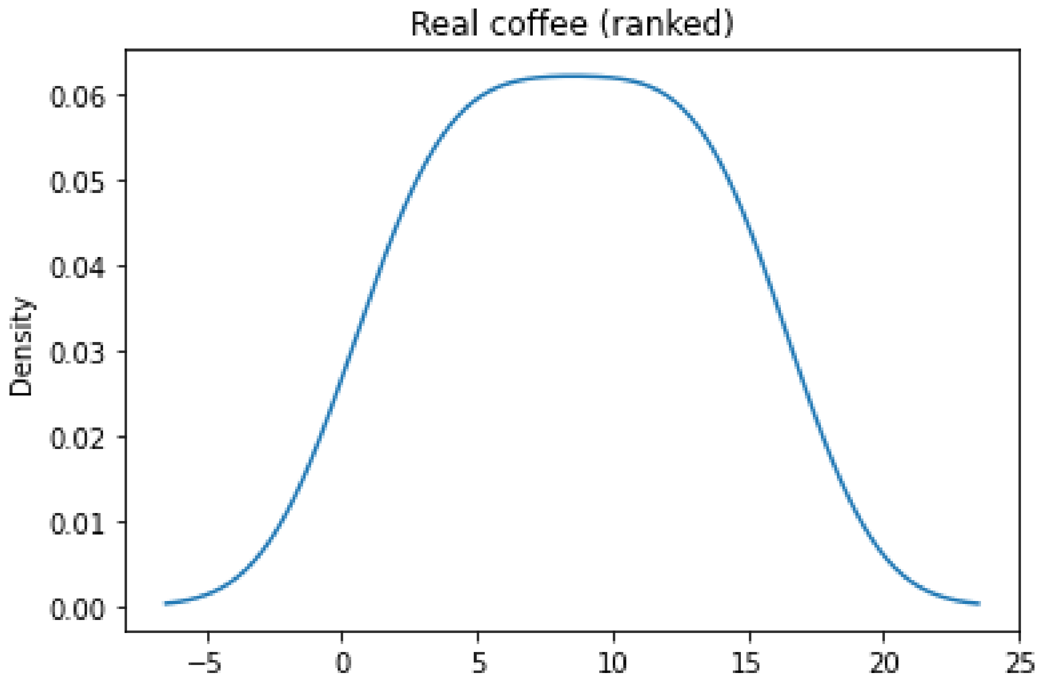
(continued from previous page)

```
0            51        19        21            27              21      81
1            41         3         2             4               2      67
2            53        11        23            11               5      87
3            67        43         7            14              14      83
4            37        23         9            13              12      76
5            73        12         7            26              23      85
6            55        76        17            20              24      76
7            34         1         5            20               3      22
8            33         1         5            15              11      49
9            69        10        17            19              15      79
10           43        43        39            54              45      56
11           32        17        11            51              42      81
12           51         4        17            30              15      61
13           27        10         8            18              12      50
14           43         2        14            23               7      59
15           75        18         2             5               3      57

    Oranges  Tinned fruit  Jam  Garlic  Butter  Margarine  Olive oil  Yoghurt  \
0        75            44   71      22      91         85         74     30.0
1        71             9   46      80      66         24         94      5.0
2        84            40   45      88      94         47         36     57.0
3        89            61   81      15      31         97         13     53.0
4        76            42   57      29      84         80         83     20.0
5        94            83   20      91      94         94         84     31.0
6        68            89   91      11      95         94         57     11.0
7        51             8   16      89      65         78         92      6.0
8        42            14   41      51      51         72         28     13.0
9        70            46   61      64      82         48         61     48.0
10       78            53   75       9      68         32         48      2.0
11       72            50   64      11      92         91         30     11.0
12       72            34   51      11      63         94         28      2.0
13       57            22   37      15      96         94         17      NaN
14       77            30   38      86      44         51         91     16.0
15       52            46   89       5      97         25         31      3.0

    Crisp bread bin_coffee bin_qual_coffee  rank_coffee
0            26     medium          medium          9.0
1            18     medium            high          7.0
2             3     medium          medium          8.0
3            15       high             low         12.5
4             5     medium          medium         11.0
5            24       high          medium         14.5
6            28        low             low          1.0
7             9        low          medium          5.0
8            11        low             low          3.0
9            30        low             low          6.0
10           93       high            high         14.5
11           34       high            high         12.5
12           62     medium            high         10.0
13           64       high            high         16.0
14           13        low             low          4.0
15            9        low             low          2.0
```

At the end of our data a new column was appended, containing the ranking of each country with the lowest being 1 and the highest equal to the amount of countries. When we visualise this distribution we get a uniform distribution, not normal but still better than before!

```
food_df['rank_coffee'].plot(kind='density', title='Real coffee (ranked)')
```

```
<AxesSubplot:title={'center':'Real coffee (ranked)'}, ylabel='Density'>
```



CHAPTER

TWELVE

SOME PRACTICE

Now that you have learned techniques in data preparation, why don't you put them to use in this wonderfully horrifying dataset. Good luck!

```
import os
import json

import pandas as pd
```

```
kaggle_dir = os.path.expanduser("~/.kaggle")
if not os.path.exists(kaggle_dir):
    os.mkdir(kaggle_dir)

with open(f'{kaggle_dir}/kaggle.json', 'w') as f:
    json.dump(
        {
            "username":"lorenzf",
            "key":"7a44a9e99b27e796177d793a3d85b8cf"
        }
        , f)
```

```
import kaggle
kaggle.api.dataset_download_files(dataset='PromptCloudHQ/us-jobs-on-monstercom', path=
↪'./data', unzip=True)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_25600/39646943.py in <module>
----> 1 import kaggle
      2 kaggle.api.dataset_download_files(dataset='PromptCloudHQ/us-jobs-on-monstercom
↪', path='./data', unzip=True)

ModuleNotFoundError: No module named 'kaggle'
```

```
df = pd.read_csv('./data/monster_com-job_sample.csv')
```

```
df.head()
```

```
                    country country_code date_added has_expired  \
0  United States of America           US        NaN          No
1  United States of America           US        NaN          No
2  United States of America           US        NaN          No
```

(continues on next page)

(continued from previous page)

```
3  United States of America           US        NaN          No
4  United States of America           US        NaN          No

          job_board                                    job_description  \
0  jobs.monster.com  TeamSoft is seeing an IT Support Specialist to...
1  jobs.monster.com  The Wisconsin State Journal is seeking a flexi...
2  jobs.monster.com  Report this job About the Job DePuy Synthes Co...
3  jobs.monster.com  Why Join Altec? If you’re considering a career...
4  jobs.monster.com  Position ID#  76162 # Positions  1 State  CT C...

                                          job_title            job_type  \
0              IT Support Technician Job in Madison   Full Time Employee
1           Business Reporter/Editor Job in Madison            Full Time
2  Johnson & Johnson Family of Companies Job Appl...  Full Time, Employee
3                     Engineer - Quality Job in Dixon            Full Time
4      Shift Supervisor - Part-Time Job in Camphill   Full Time Employee

                                           location  \
0                                 Madison, WI 53702
1                                 Madison, WI 53708
2  DePuy Synthes Companies is a member of Johnson...
3                                         Dixon, CA
4                                      Camphill, PA

                     organization  \
0                             NaN
1         Printing and Publishing
2  Personal and Household Services
3                 Altec Industries
4                          Retail

                                           page_url salary  \
0  http://jobview.monster.com/it-support-technici...    NaN
1  http://jobview.monster.com/business-reporter-e...    NaN
2  http://jobview.monster.com/senior-training-lea...    NaN
3  http://jobview.monster.com/engineer-quality-jo...    NaN
4  http://jobview.monster.com/shift-supervisor-pa...    NaN

                     sector                           uniq_id
0    IT/Software Development  11d599f229a80023d2f40e7c52cd941e
1                       NaN  e4cbb126dabf22159aff90223243ff2a
2                       NaN  839106b353877fa3d896ffb9c1fe01c0
3   Experienced (Non-Manager)  58435fcab804439efdcaa7ecca0fd783
4  Project/Program Management  64d0272dc8496abfd9523a8df63c184c
```

Need some inspiration? perhaps this might help!

Part III

3. Data Preprocessing

CHAPTER

THIRTEEN

DATA PREPROCESSING

this is an introduction

CHAPTER

FOURTEEN

INDEXING AND SLICING

In

```
import pandas as pd
```

```
min_temp_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c2_data_preparation/data/temperatures/australia/
↪melbourne/1981.csv')
min_temp_df
```

```
           Date  Temp
0    1981-01-01  20.7
1    1981-01-02  17.9
2    1981-01-03  18.8
3    1981-01-04  14.6
4    1981-01-05  15.8
..          ...   ...
360  1981-12-27  15.5
361  1981-12-28  13.3
362  1981-12-29  15.6
363  1981-12-30  15.2
364  1981-12-31  17.4

[365 rows x 2 columns]
```

```
min_temp_df.Date = pd.to_datetime(min_temp_df.Date)
```

```
min_temp_df = min_temp_df.set_index('Date')
```

```
min_temp_df.loc['1981-06-01':'1981-06-30']
```

```
            Temp
Date
1981-06-01  11.6
1981-06-02  10.6
1981-06-03   9.8
1981-06-04  11.2
1981-06-05   5.7
1981-06-06   7.1
1981-06-07   2.5
1981-06-08   3.5
1981-06-09   4.6
```

(continues on next page)

(continued from previous page)

```
1981-06-10  11.0
1981-06-11   5.7
1981-06-12   7.7
1981-06-13  10.4
1981-06-14  11.4
1981-06-15   9.2
1981-06-16   6.1
1981-06-17   2.7
1981-06-18   4.3
1981-06-19   6.3
1981-06-20   3.8
1981-06-21   4.4
1981-06-22   7.1
1981-06-23   4.8
1981-06-24   5.8
1981-06-25   6.2
1981-06-26   7.3
1981-06-27   9.2
1981-06-28  10.2
1981-06-29   9.5
1981-06-30   9.5
```

```
min_temp_df.loc['1989-06-01':'1989-06-30'].mean()
```

```
Temp   NaN
dtype: float64
```

```
min_temp_df.resample('MS').mean()
```

```
                 Temp
Date
1981-01-01  17.712903
1981-02-01  17.678571
1981-03-01  13.500000
1981-04-01  12.356667
1981-05-01   9.490323
1981-06-01   7.306667
1981-07-01   7.577419
1981-08-01   7.238710
1981-09-01  10.143333
1981-10-01  10.087097
1981-11-01  11.890000
1981-12-01  13.680645
```

```
import seaborn as sns
```

```
tip_df = sns.load_dataset('tips')
tip_df.head()
```

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
```

(continues on next page)

(continued from previous page)

```
4        24.59  3.61  Female      No  Sun  Dinner     4
```

```
tip_index_df = tip_df.set_index('day')
```

```
tip_index_df.loc['Sun']
```

```
     total_bill   tip     sex smoker    time  size
day
Sun       16.99  1.01  Female     No  Dinner     2
Sun       10.34  1.66    Male     No  Dinner     3
Sun       21.01  3.50    Male     No  Dinner     3
Sun       23.68  3.31    Male     No  Dinner     2
Sun       24.59  3.61  Female     No  Dinner     4
..          ...   ...     ...    ...     ...   ...
Sun       20.90  3.50  Female    Yes  Dinner     3
Sun       30.46  2.00    Male    Yes  Dinner     5
Sun       18.15  3.50  Female    Yes  Dinner     3
Sun       23.10  4.00    Male    Yes  Dinner     3
Sun       15.69  1.50    Male    Yes  Dinner     2

[76 rows x 6 columns]
```

```
tip_index_df = tip_df.set_index(['day','time'])
```

```
tip_index_df.loc[('Thur','Lunch')].tip.mean()
```

```
/tmp/ipykernel_25625/2537502835.py:1: PerformanceWarning: indexing past lexsort depth␣
↪may impact performance.
  tip_index_df.loc[('Thur','Lunch')].tip.mean()
```

```
2.767704918032786
```

```
pd.pivot_table(tip_df, values='total_bill', index='day', columns='time', aggfunc=
↪'median')
```

```
time  Lunch  Dinner
day
Thur  16.00  18.780
Fri   13.42  18.665
Sat     NaN  18.240
Sun     NaN  19.630
```

```
tip_df.set_index(['sex', 'time','smoker']).loc[('Male', 'Dinner','Yes')]['tip'].mean()
```

```
/tmp/ipykernel_25625/3467525553.py:1: PerformanceWarning: indexing past lexsort depth␣
↪may impact performance.
  tip_df.set_index(['sex', 'time','smoker']).loc[('Male', 'Dinner','Yes')]['tip'].
↪mean()
```

```
3.123191489361702
```

CHAPTER

FIFTEEN

MERGE

When data becomes multi-dimensional - covering multiple aspects of information - it usually happens that a lot of information is redundant. Take for example the next dataset, we have collected ratings of restaurants from users, when a single user rates 2 restaurants the information of the user relates to both rows, yet it would be wasteful to keep this info twice. The same can happen when we have a restaurant with 2 ratings, the location of the restaurant is kept twice in our data, which is not scalable.

We solve this problem using relational data, the idea is that we have a common key column in 2 of our tables which we can use to join the data for further processing.

In our example we use a dataset with consumers, restaurants and ratings between those, you can find more information here.

```
import pandas as pd
```

```
rating_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c3_data_preprocessing/data/cuisine/rating_final.csv')
rating_df
```

```
     userID  placeID  rating  food_rating  service_rating
0     U1077   135085       2            2               2
1     U1077   135038       2            2               1
2     U1077   132825       2            2               2
3     U1077   135060       1            2               2
4     U1068   135104       1            1               2
...     ...      ...     ...          ...             ...
1156  U1043   132630       1            1               1
1157  U1011   132715       1            1               0
1158  U1068   132733       1            1               0
1159  U1068   132594       1            1               1
1160  U1068   132660       0            0               0

[1161 rows x 5 columns]
```

this first table we read contains the userID from whom the rating came, the placeID is the restaurant he/she rated and the numerical values of the 3 different ratings.

Perhaps you can find out what the min and max values for the ratings are?

to know the type of restaurant, we can not read another table

```
cuisine_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c3_data_preprocessing/data/cuisine/chefmozcuisine.csv')
cuisine_df
```

```
     placeID        Rcuisine
0     135110         Spanish
1     135109         Italian
2     135107  Latin_American
3     135106         Mexican
4     135105       Fast_Food
..       ...             ...
911   132005         Seafood
912   132004         Seafood
913   132003   International
914   132002         Seafood
915   132001   Dutch-Belgian

[916 rows x 2 columns]
```

This table also contains the placeID, so we should be able to merge/join these 2 tables and create a new table with info of both. Notice how we specify the 'on' parameter where we denote placeID as our common key.

```
merged_df = pd.merge(rating_df, cuisine_df, on='placeID', how='inner')
merged_df
```

```
     userID  placeID  rating  food_rating  service_rating   Rcuisine
0     U1077   135085       2            2               2  Fast_Food
1     U1108   135085       1            2               1  Fast_Food
2     U1081   135085       1            2               1  Fast_Food
3     U1056   135085       2            2               2  Fast_Food
4     U1134   135085       2            1               2  Fast_Food
...     ...      ...     ...          ...             ...        ...
1038  U1061   132958       2            2               2   American
1039  U1025   132958       1            0               0   American
1040  U1097   132958       2            1               1   American
1041  U1096   132958       1            2               2   American
1042  U1136   132958       2            2               2   American

[1043 rows x 6 columns]
```

Great! now we have more info about the rating that were given, being the type of cuisine that they rated. We could figure out which cuisines are available in our dataset and do a comparison, let us count the occurences of each cuisine.

```
merged_df.Rcuisine.value_counts()
```

```
Mexican              238
Bar                  140
Cafeteria            102
Fast_Food             91
Seafood               62
Bar_Pub_Brewery       59
Pizzeria              51
Chinese               41
American              39
International         37
Contemporary          32
Burgers               31
Japanese              29
Italian               26
Family                14
```

(continues on next page)

(continued from previous page)

```
Cafe-Coffee_Shop     12
Breakfast-Brunch      9
Game                  7
Vietnamese            6
Bakery                5
Mediterranean         4
Armenian              4
Regional              4
Name: Rcuisine, dtype: int64
```

A lot of mexican, which is not surpising as this dataset comes from Mexico. I wonder if there is a difference between 'Bar' and 'Bar_Pub_Brewery', we can see if the average rating for those 2 differ.

```
for cuisine in ['Bar', 'Bar_Pub_Brewery']:
    print(cuisine)
    print(merged_df[merged_df.Rcuisine==cuisine][['rating', 'food_rating', 'service_
↪rating']].mean())
    print()
```

```
Bar
rating            1.200000
food_rating       1.135714
service_rating    1.085714
dtype: float64

Bar_Pub_Brewery
rating            1.305085
food_rating       1.169492
service_rating    1.203390
dtype: float64
```

just looking at the averages we can deduces that while food ratings do not change a lot, the service seems a lot better at the Brewery.

```
merged_df[merged_df.Rcuisine=='Cafeteria'][['rating', 'food_rating', 'service_rating
↪']].mean()
```

```
rating            1.205882
food_rating       1.127451
service_rating    1.078431
dtype: float64
```

```
merged_df[merged_df.Rcuisine=='Cafe-Coffee_Shop'][['rating', 'food_rating', 'service_
↪rating']].mean()
```

```
rating            1.583333
food_rating       1.333333
service_rating    1.416667
dtype: float64
```

As easy as it looks, we can now merge information of different tables in our dataset and perform some simple comparisons, in later sections we will see how we can improve on those.

As an exercise I already read in the table containing the info about which type of payment the user has opted for. Could you find out if the type of payment could have an influence on the rating?

```
user_payment_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c3_data_preprocessing/data/cuisine/userpayment.csv')
user_payment_df
```

```
    userID          Upayment
0    U1001              cash
1    U1002              cash
2    U1003              cash
3    U1004              cash
4    U1004  bank_debit_cards
..     ...               ...
172  U1134              cash
173  U1135              cash
174  U1136              cash
175  U1137              cash
176  U1138              cash

[177 rows x 2 columns]
```

CHAPTER

SIXTEEN

GROUPBY

In the previous section we saw how to combine information of multiple tables from our dataset. Here we are going to build further on that by using the merged information to group on categorical variables.

```
import pandas as pd
```

```
rating_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c3_data_preprocessing/data/cuisine/rating_final.csv')
rating_df
```

```
     userID  placeID  rating  food_rating  service_rating
0     U1077   135085       2            2               2
1     U1077   135038       2            2               1
2     U1077   132825       2            2               2
3     U1077   135060       1            2               2
4     U1068   135104       1            1               2
...     ...      ...     ...          ...             ...
1156  U1043   132630       1            1               1
1157  U1011   132715       1            1               0
1158  U1068   132733       1            1               0
1159  U1068   132594       1            1               1
1160  U1068   132660       0            0               0

[1161 rows x 5 columns]
```

Again we have our rating data containing the users, places and ratings they gave. As a simple example we could just group by the placeID column and take the mean, this would give us the mean rating for each restaurant

```
grouped_rating_df = rating_df.groupby('placeID').mean().sort_values('rating')
grouped_rating_df
```

```
           rating  food_rating  service_rating
placeID
132654   0.250000         0.25        0.250000
135040   0.250000         0.25        0.250000
132560   0.500000         1.00        0.250000
132663   0.500000         0.50        0.666667
135069   0.500000         0.50        0.750000
...           ...          ...             ...
132755   1.800000         2.00        1.600000
132922   1.833333         1.50        1.833333
134986   2.000000         2.00        2.000000
135034   2.000000         2.00        1.600000
```

(continues on next page)

(continued from previous page)

```
132955   2.000000         1.80        1.800000

[130 rows x 3 columns]
```

Keep in mind that this might be tricky, as we do not always have as much records per group, we could count the amount per records using a groupby operation and count.

```
rating_df.groupby('placeID').rating.count()
```

```
placeID
132560     4
132561     4
132564     4
132572    15
132583     4
          ..
135088     6
135104     7
135106    10
135108    11
135109     4
Name: rating, Length: 130, dtype: int64
```

Taking an average of 4 ratings might not be ideal, so we should keep in mind that our groups have a good sample size.

Let's make things more interesting and insert some location data.

```
geo_df = pd.read_csv('./data/cuisine/geoplaces2.csv').set_index('placeID')
geo_df
```

```
          latitude   longitude  \
placeID
134999   18.915421  -99.184871
132825   22.147392 -100.983092
135106   22.149709 -100.976093
132667   23.752697  -99.163359
132613   23.752903  -99.165076
...            ...         ...
132866   22.141220 -100.931311
135072   22.149192 -101.002936
135109   18.921785  -99.235350
135019   18.875011  -99.159422
132877   22.135364 -100.934948

                                             the_geom_meter  \
placeID
134999   0101000020957F000088568DE356715AC138C0A525FC46...
132825   0101000020957F00001AD016568C4858C1243261274BA5...
135106   0101000020957F0000649D6F21634858C119AE9BF528A3...
132667   0101000020957F00005D67BCDDED8157C1222A2DC8D84D...
132613   0101000020957F00008EBA2D06DC8157C194E03B7B504E...
...                                                     ...
132866   0101000020957F000013871838EC4A58C1B5DF74F8E396...
135072   0101000020957F0000E7B79B1DB94758C1D29BC363D8AA...
135109   0101000020957F0000A6BF695F136F5AC1DADF87B20556...
135019   0101000020957F0000B49B2E5C6E785AC12F9D58435241...
```

(continues on next page)

(continued from previous page)

```
132877   0101000020957F000090735015B84B58C1AF0DC0414698...

                                  name  \
placeID                                   
134999                  Kiku Cuernavaca   
132825                  puesto de tacos   
135106       El Rinc�n de San Francisco   
132667   little pizza Emilio Portes Gil   
132613                    carnitas_mata   
...                                 ...   
132866                          Chaires   
135072                       Sushi Itto   
135109                        Paniroles   
135019      Restaurant Bar Coty y Pablo   
132877                  sirloin stockade   

                                                address             city  \
placeID                                                                  
134999                                       Revolucion       Cuernavaca   
132825            esquina santos degollado y leon guzman           s.l.p.   
135106                                  Universidad 169  San Luis Potosi   
132667                           calle emilio portes gil        victoria   
132613                            lic. Emilio portes gil         victoria   
...                                                 ...              ...   
132866                                 Ricardo B. Anaya  San Luis Potosi   
135072                Venustiano Carranza 1809 C Polanco  San Luis Potosi   
135109                                                ?                ?   
135019   Paseo de Las Fuentes 24 Pedregal de Las Fuentes         Jiutepec   
132877                                                ?                ?   

                   state country fax    zip           alcohol   smoking_area  \
placeID                                                                      
134999           Morelos  Mexico   ?      ?  No_Alcohol_Served           none   
132825            s.l.p.  mexico   ?  78280  No_Alcohol_Served           none   
135106   San Luis Potosi  Mexico   ?  78000          Wine-Beer    only at bar   
132667        tamaulipas       ?   ?      ?  No_Alcohol_Served           none   
132613        Tamaulipas  Mexico   ?      ?  No_Alcohol_Served      permitted   
...                  ...     ...  ..    ...                ...            ...   
132866   San Luis Potosi  Mexico   ?      ?  No_Alcohol_Served  not permitted   
135072               SLP  Mexico   ?  78220  No_Alcohol_Served           none   
135109                 ?       ?   ?      ?          Wine-Beer  not permitted   
135019           Morelos  Mexico   ?      ?  No_Alcohol_Served           none   
132877                 ?       ?   ?      ?  No_Alcohol_Served           none   

        dress_code     accessibility   price                     url Rambience  \
placeID                                                                      
134999    informal  no_accessibility  medium  kikucuernavaca.com.mx  familiar   
132825    informal        completely     low                      ?  familiar   
135106    informal         partially  medium                      ?  familiar   
132667    informal        completely     low                      ?  familiar   
132613    informal        completely  medium                      ?  familiar   
...            ...               ...     ...                    ...       ...   
132866    informal        completely  medium                      ?  familiar   
135072    informal  no_accessibility  medium      sushi-itto.com.mx  familiar   
135109    informal  no_accessibility  medium                      ?     quiet   
135019    informal        completely     low                      ?  familiar   
132877    informal        completely     low                      ?  familiar   
```

(continues on next page)

(continued from previous page)

```
        franchise    area other_services
placeID
134999          f  closed           none
132825          f    open           none
135106          f    open           none
132667          t  closed           none
132613          t  closed           none
...           ...     ...            ...
132866          f  closed           none
135072          f  closed           none
135109          f  closed       Internet
135019          f  closed           none
132877          f  closed           none

[130 rows x 20 columns]
```

Here we have for each restaurant information about its location, I mentioned earlier that grouping per restaurant might be dangerous as some restaurants have nearly no reviews. By adding information such as city, state and country we have other categorical variables to group by. Notice how we use the merge operation from previous section, but this time specify our common key is the index.

```
geo_rating_df = pd.merge(grouped_rating_df, geo_df, left_index=True, right_index=True)
geo_rating_df
```

```
           rating  food_rating  service_rating   latitude   longitude  \
placeID
132654   0.250000         0.25        0.250000  23.735523  -99.129588
135040   0.250000         0.25        0.250000  22.135617 -100.969709
132560   0.500000         1.00        0.250000  23.752304  -99.166913
132663   0.500000         0.50        0.666667  23.752511  -99.166954
135069   0.500000         0.50        0.750000  22.140129 -100.944872
...           ...          ...             ...        ...         ...
132755   1.800000         2.00        1.600000  22.153324 -101.019546
132922   1.833333         1.50        1.833333  22.151135 -100.982311
134986   2.000000         2.00        2.000000  18.928798  -99.239513
135034   2.000000         2.00        1.600000  22.140517 -101.021422
132955   2.000000         1.80        1.800000  22.147622 -101.010275

                                           the_geom_meter  \
placeID
132654   0101000020957F000040E8F628488557C18224E8B94845...
135040   0101000020957F00001B552189B84A58C15A2AAEFD2CA2...
132560   0101000020957F0000FC60BDA8E88157C1B2C357D6DA4E...
132663   0101000020957F0000FDF8D26EE08157C1FEDB6A1FDB4E...
135069   0101000020957F000038E5D546B74A58C18FD29AD0D29A...
...                                                   ...
132755   0101000020957F000026CADE45A14658C1F011EBCA55AF...
132922   0101000020957F000060A98A38FF4758C146718E41D9A4...
134986   0101000020957F00002A0D05E2D96D5AC1AB058CB1EC56...
135034   0101000020957F000026D92BB4894858C161A7552DA2B0...
132955   0101000020957F000068BE7C87C24758C1920A360A08AD...

                                         name  \
placeID
132654   Carnitas Mata  Calle 16 de Septiembre
```

(continues on next page)

(continued from previous page)

```
135040                Restaurant los Compadres
132560                      puesto de gorditas
132663                               tacos abi
135069               Abondance Restaurante Bar
...                                        ...
132755                    La Estrella de Dimas
132922                    cafe punta del cielo
134986                Restaurant Las Mananitas
135034              Michiko Restaurant Japones
132955                               emilianos

                                             address            city  \
placeID
132654                            16 de Septiembre        victoria
135040                Camino a Simon Diaz 155 Centro  San Luis Potosi
132560                         frente al tecnologico         victoria
132663                                             ?         victoria
135069                   Industrias 908 Valle Dorado  San Luis Potosi
...                                              ...             ...
132755                         Av. de los Pintores  San Luis Potosi
132922                                             ?               ?
134986                         Ricardo Linares 107       Cuernavaca
135034   Cordillera de Los Alpes 160 Lomas 2 Seccion  San Luis Potosi
132955                         venustiano carranza   san luis potos

              state  ...          alcohol smoking_area dress_code  \
placeID              ...
132654   tamaulipas  ...  No_Alcohol_Served         none   informal
135040          SLP  ...          Wine-Beer         none   informal
132560   tamaulipas  ...  No_Alcohol_Served    permitted   informal
132663   tamaulipas  ...  No_Alcohol_Served         none   informal
135069          SLP  ...          Wine-Beer         none   informal
...             ...  ...                ...          ...        ...
132755       S.L.P.  ...  No_Alcohol_Served         none   informal
132922            ?  ...  No_Alcohol_Served    permitted     formal
134986      Morelos  ...          Wine-Beer         none     formal
135034          SLP  ...  No_Alcohol_Served         none   informal
132955       mexico  ...          Wine-Beer         none   informal

           accessibility   price                 url Rambience franchise  \
placeID
132654        completely     low                   ?  familiar         f
135040   no_accessibility    high                   ?  familiar         f
132560   no_accessibility     low                   ?  familiar         f
132663        completely     low                   ?  familiar         f
135069   no_accessibility     low                   ?  familiar         f
...                   ...     ...                 ...       ...       ...
132755         partially  medium                   ?  familiar         f
132922        completely  medium                   ?  familiar         f
134986   no_accessibility    high  lasmananitas.com.mx  familiar         f
135034   no_accessibility  medium                   ?  familiar         f
132955        completely     low                   ?  familiar         t

           area other_services
placeID
132654   closed           none
135040   closed           none
```

(continues on next page)

(continued from previous page)

```
132560     open          none
132663   closed          none
135069   closed          none
...         ...           ...
132755   closed       variety
132922   closed          none
134986   closed          none
135034   closed          none
132955   closed       variety

[130 rows x 23 columns]
```

By adding this amount of data, things are getting a bit cluttered, thankfully we can use pandas to get a list of all our columns.

```
geo_rating_df.columns
```

```
Index(['rating', 'food_rating', 'service_rating', 'latitude', 'longitude',
       'the_geom_meter', 'name', 'address', 'city', 'state', 'country', 'fax',
       'zip', 'alcohol', 'smoking_area', 'dress_code', 'accessibility',
       'price', 'url', 'Rambience', 'franchise', 'area', 'other_services'],
      dtype='object')
```

How about we try and see if we can find a difference between countries for the ratings?

```
geo_rating_df.groupby('country')[['rating', 'food_rating', 'service_rating']].mean()
```

```
           rating  food_rating  service_rating
country
?        1.166045     1.232946        1.069169
Mexico   1.200977     1.229093        1.118162
mexico   1.062660     1.069006        0.900064
```

Ah, it seems we forgot to do some data cleaning here, perhaps you could jump in and fix this string problem, might as well tackle the missing value while we are at it. Aside from that, we can see that lower-case Mexico is not doing very well, perhaps the food was so bad they forgot how to write Mexico?

Jokes aside, do you see the ressemblance between this and our rudimentary approach of comparing different categories? We are slowly getting more and more efficient using these operations, how about the difference between alcohol consumption?

```
geo_rating_df.groupby('alcohol')[['rating', 'food_rating', 'service_rating']].mean()
```

```
                     rating  food_rating  service_rating
alcohol
Full_Bar           1.287124     1.218315        1.170311
No_Alcohol_Served  1.148075     1.194730        1.042417
Wine-Beer          1.231887     1.261840        1.174437
```

Something we can remark here is that the food rating for no alcohol locations seems to be holding up, whilst the general rating and service rating fall behind. This would suggest that the food rating indeed is for the food, where the type of drinks served have no influence.

As a last we look at the difference between accessibility, does that influences our ratings?

```
geo_rating_df.groupby('accessibility')[['rating', 'food_rating', 'service_rating']].
↪mean()
```

```
                     rating  food_rating  service_rating
accessibility
completely         1.132494     1.203597        1.049709
no_accessibility   1.196189     1.206242        1.091278
partially          1.275356     1.330294        1.219991
```

It seems having partial accessibility is the way to go here, performing better than complete accessibility. We can however find that is due to a low sample size of 9 restaurants, making it prone to variation.

```
geo_rating_df.accessibility.value_counts()
```

```
no_accessibility    76
completely          45
partially            9
Name: accessibility, dtype: int64
```

You should get the hang of it by now, perhaps you can play some more with the other categories.

There is one thing I still would like to address, you perhaps have notices that in the beginning I first took the average rating per restaurant and later again took the average per category. This is a bad practice as a bad restaurant with one review has equal influence as a good restaurant with 100 reviews, perhaps you can think of a way to group all reviews from a category instead of the average for each restaurant?

In the previous section we added the cuisine type, perhaps you could do some groupby operations on that too here?

CHAPTER

SEVENTEEN

PIVOT

When using the groupby operation we used 1 categorical variable to seperate/group our data into those categories. Here we go a step further and use 2 categories to aggregate our data, resulting in a comparison matrix.

Aside from that, the pivot operation can in general be used to go from a long data format, to a wide data format. To keep things uniform we stick with the same cuisine dataset.

```
import pandas as pd
```

```
rating_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c3_data_preprocessing/data/cuisine/rating_final.csv')
rating_df
```

```
     userID  placeID  rating  food_rating  service_rating
0     U1077   135085       2            2               2
1     U1077   135038       2            2               1
2     U1077   132825       2            2               2
3     U1077   135060       1            2               2
4     U1068   135104       1            1               2
...     ...      ...     ...          ...             ...
1156  U1043   132630       1            1               1
1157  U1011   132715       1            1               0
1158  U1068   132733       1            1               0
1159  U1068   132594       1            1               1
1160  U1068   132660       0            0               0

[1161 rows x 5 columns]
```

And again we merge with the geolocations data, I feel that it becomse obvious here how these operations are very related to eachother.

```
geo_df = pd.read_csv('https://raw.githubusercontent.com/LorenzF/data-science-
↪practical-approach/main/src/c3_data_preprocessing/data/cuisine/geoplaces2.csv')
```

A subtle difference between last time is that I did not first group per restaurant, however this leads to a dataframe that has a lot of redundant information! Try to look in the merged dataframe and spot the copies of data.

```
geo_rating_df = pd.merge(rating_df, geo_df, on='placeID')
geo_rating_df
```

```
     userID  placeID  rating  food_rating  service_rating   latitude  \
0     U1077   135085       2            2               2  22.150802
1     U1108   135085       1            2               1  22.150802
2     U1081   135085       1            2               1  22.150802
```

(continues on next page)

(continued from previous page)

```
3     U1056   135085       2           2              2  22.150802
4     U1134   135085       2           1              2  22.150802
...     ...      ...     ...         ...            ...        ...
1156  U1061   132958       2           2              2  22.144979
1157  U1025   132958       1           0              0  22.144979
1158  U1097   132958       2           1              1  22.144979
1159  U1096   132958       1           2              2  22.144979
1160  U1136   132958       2           2              2  22.144979

       longitude                                   the_geom_meter  \
0    -100.982680  0101000020957F00009F823DA6094858C18A2D4D37F9A4...
1    -100.982680  0101000020957F00009F823DA6094858C18A2D4D37F9A4...
2    -100.982680  0101000020957F00009F823DA6094858C18A2D4D37F9A4...
3    -100.982680  0101000020957F00009F823DA6094858C18A2D4D37F9A4...
4    -100.982680  0101000020957F00009F823DA6094858C18A2D4D37F9A4...
...          ...                                              ...
1156 -101.005683  0101000020957F000049095EB34A4858C15CB4BD1EE1AB...
1157 -101.005683  0101000020957F000049095EB34A4858C15CB4BD1EE1AB...
1158 -101.005683  0101000020957F000049095EB34A4858C15CB4BD1EE1AB...
1159 -101.005683  0101000020957F000049095EB34A4858C15CB4BD1EE1AB...
1160 -101.005683  0101000020957F000049095EB34A4858C15CB4BD1EE1AB...

                        name                       address  ...  \
0     Tortas Locas Hipocampo  Venustiano Carranza 719 Centro  ...
1     Tortas Locas Hipocampo  Venustiano Carranza 719 Centro  ...
2     Tortas Locas Hipocampo  Venustiano Carranza 719 Centro  ...
3     Tortas Locas Hipocampo  Venustiano Carranza 719 Centro  ...
4     Tortas Locas Hipocampo  Venustiano Carranza 719 Centro  ...
...                      ...                             ...  ...
1156      tacos los volcanes          avenida hivno nacional  ...
1157      tacos los volcanes          avenida hivno nacional  ...
1158      tacos los volcanes          avenida hivno nacional  ...
1159      tacos los volcanes          avenida hivno nacional  ...
1160      tacos los volcanes          avenida hivno nacional  ...

                alcohol   smoking_area dress_code     accessibility   price  \
0     No_Alcohol_Served  not permitted   informal  no_accessibility  medium
1     No_Alcohol_Served  not permitted   informal  no_accessibility  medium
2     No_Alcohol_Served  not permitted   informal  no_accessibility  medium
3     No_Alcohol_Served  not permitted   informal  no_accessibility  medium
4     No_Alcohol_Served  not permitted   informal  no_accessibility  medium
...                 ...            ...        ...               ...     ...
1156  No_Alcohol_Served           none   informal        completely     low
1157  No_Alcohol_Served           none   informal        completely     low
1158  No_Alcohol_Served           none   informal        completely     low
1159  No_Alcohol_Served           none   informal        completely     low
1160  No_Alcohol_Served           none   informal        completely     low

     url Rambience franchise    area other_services
0      ?  familiar         f  closed           none
1      ?  familiar         f  closed           none
2      ?  familiar         f  closed           none
3      ?  familiar         f  closed           none
4      ?  familiar         f  closed           none
...   ..       ...       ...     ...            ...
1156   ?     quiet         t  closed           none
1157   ?     quiet         t  closed           none
```

(continues on next page)

(continued from previous page)

```
1158   ?     quiet         t  closed          none
1159   ?     quiet         t  closed          none
1160   ?     quiet         t  closed          none

[1161 rows x 25 columns]
```

Now that we have our workable data, we can choose 2 categories and create a comparison matrix using the pivot operation. Yet there might be a problem that we still have to resolve, can you figure out the problem reading the error at the end of the stack trace below?

```
geo_rating_df.pivot(index='alcohol', columns='smoking_area', values='rating')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_20513/1351770208.py in <module>
----> 1 geo_rating_df.pivot(index='alcohol', columns='smoking_area', values='rating')

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪frame.py in pivot(self, index, columns, values)
   7791         from pandas.core.reshape.pivot import pivot
   7792
-> 7793         return pivot(self, index=index, columns=columns, values=values)
   7794
   7795     _shared_docs[

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪reshape/pivot.py in pivot(data, index, columns, values)
    515         else:
    516             indexed = data._constructor_sliced(data[values]._values,␣
↪index=multiindex)
--> 517     return indexed.unstack(columns_listlike)
    518
    519

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪series.py in unstack(self, level, fill_value)
   4079         from pandas.core.reshape.reshape import unstack
   4080
-> 4081         return unstack(self, level, fill_value)
   4082
   4083     # ----------------------------------------------------------------------

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪reshape/reshape.py in unstack(obj, level, fill_value)
    458         if is_1d_only_ea_dtype(obj.dtype):
    459             return _unstack_extension_series(obj, level, fill_value)
--> 460         unstacker = _Unstacker(
    461             obj.index, level=level, constructor=obj._constructor_expanddim
    462         )

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪reshape/reshape.py in __init__(self, index, level, constructor)
    131             raise ValueError("Unstacked DataFrame is too big, causing int32␣
↪overflow")
    132
--> 133         self._make_selectors()
```

(continues on next page)

(continued from previous page)

```
    134
    135     @cache_readonly

~/git/data-science-practical-approach/venv/lib/python3.8/site-packages/pandas/core/
↪reshape/reshape.py in _make_selectors(self)
    183
    184         if mask.sum() < len(self.index):
--> 185             raise ValueError("Index contains duplicate entries, cannot reshape
↪")
    186
    187         self.group_index = comp_index

ValueError: Index contains duplicate entries, cannot reshape
```

It says: 'Index contains duplicate entries, cannot reshape' meaning that some combinations of our 2 categories, alcohol and smoking area have duplicates, which is understandable. I opted to solve this by grouping over the 2 categories and taking the mean for each combination, then i take this grouped data and pivot by setting the alcohol consumption as index and the smoking are as columns.

```
grouped_geo_rating_df = geo_rating_df.groupby(['alcohol','smoking_area'])[['rating',
↪'food_rating', 'service_rating']].mean().reset_index()
grouped_geo_rating_df.pivot(index='alcohol', columns='smoking_area', values='rating')
```

```
smoking_area           none  not permitted  only at bar  permitted   section
alcohol
Full_Bar           1.305556       0.857143          NaN   1.500000  1.272727
No_Alcohol_Served  1.186788       1.124402          NaN   1.114286  1.265823
Wine-Beer          1.217391       1.000000     1.368421   1.300000  1.275000
```

Wonderful! Now we have for each combination an average rating, notice however that not every combination has the same sample size, so comparing might be tricky if you only have a few ratings.

To figure that out I counted the ratings per combination.

```
geo_rating_df.groupby(['alcohol','smoking_area']).count().reset_index().pivot(index=
↪'alcohol', columns='smoking_area', values='rating')
```

```
smoking_area        none  not permitted  only at bar  permitted  section
alcohol
Full_Bar            36.0            7.0          NaN        4.0     33.0
No_Alcohol_Served  439.0          209.0          NaN       35.0     79.0
Wine-Beer          161.0            9.0         19.0       10.0    120.0
```

It seems that there might e a correlation between the 2 categories, as a lot of place where smoking is not permitted/none, there is no alcohol served, which makes sense. Comparing the ratings with alcohol allowance for places where smoking is not permitted is not a good idea, the counts are 7, 209 and 9, very unbalanced.

```
geo_df.columns
```

```
Index(['placeID', 'latitude', 'longitude', 'the_geom_meter', 'name', 'address',
       'city', 'state', 'country', 'fax', 'zip', 'alcohol', 'smoking_area',
       'dress_code', 'accessibility', 'price', 'url', 'Rambience', 'franchise',
       'area', 'other_services'],
      dtype='object')
```

I printed the columns above, perhaps you could figure out a relation between the price category and the (R)ambience of the restaurant? Perhaps there are other combinations of which I did not think of, try some out!

Part IV

4. Data Visualisation

CHAPTER EIGHTEEN

INTRODUCTION

this is an introduction

CHAPTER

NINETEEN

LINE PLOT

The most straight-forward yet very useful plotting graph is the line plot. With the line plot we achieve the visualisation of a single feature organized in a usually time based reference.

The line plot is ideal if you want to achieve a time critical pattern residing within your data. In this example we use the prepared taxi dataframe that comes with our plotting library seaborn.

From all possible plotting libraries in Python we opted for the seaborn as it has an optimal combination of simplicity and beaty, yet other libraries are equally powerful.

We begin by importing our neccesary libraries

```
import pandas as pd
import seaborn as sns
sns.set_theme()
```

For aestetic reasons we change the figure size to something a bit larger

```
sns.set(rc={'figure.figsize':(16,12)})
```

We load our dataset, this dataset contains the trip of taxi's in regions of New York City with timestamps of pickup and dropoff.

```
taxi_df = sns.load_dataset('taxis')
taxi_df.head()
```

```
                pickup             dropoff  passengers  distance  fare   tip  \
0  2019-03-23 20:21:09  2019-03-23 20:27:24           1      1.60   7.0  2.15
1  2019-03-04 16:11:55  2019-03-04 16:19:00           1      0.79   5.0  0.00
2  2019-03-27 17:53:01  2019-03-27 18:00:25           1      1.37   7.5  2.36
3  2019-03-10 01:23:59  2019-03-10 01:49:51           1      7.70  27.0  6.15
4  2019-03-30 13:27:42  2019-03-30 13:37:14           3      2.16   9.0  1.10

   tolls  total   color      payment            pickup_zone  \
0    0.0  12.95  yellow  credit card        Lenox Hill West
1    0.0   9.30  yellow         cash  Upper West Side South
2    0.0  14.16  yellow  credit card          Alphabet City
3    0.0  36.95  yellow  credit card              Hudson Sq
4    0.0  13.40  yellow  credit card           Midtown East

            dropoff_zone pickup_borough dropoff_borough
0    UN/Turtle Bay South      Manhattan       Manhattan
1  Upper West Side South      Manhattan       Manhattan
2           West Village      Manhattan       Manhattan
3         Yorkville West      Manhattan       Manhattan
4         Yorkville West      Manhattan       Manhattan
```

As we saw earlier, it is important to prepare the data, due to storage specification they did not parse the dates into a datetime format, which we do here.

```
taxi_df.pickup = pd.to_datetime(taxi_df.pickup)
taxi_df.dropoff = pd.to_datetime(taxi_df.dropoff)
```

Before we can do anything with this dataset, we need to format it into a proper format, for our first graph I would like to view the total amount of passengers per day. This means we have to take our data and resample on the pickup date, taking the sum.

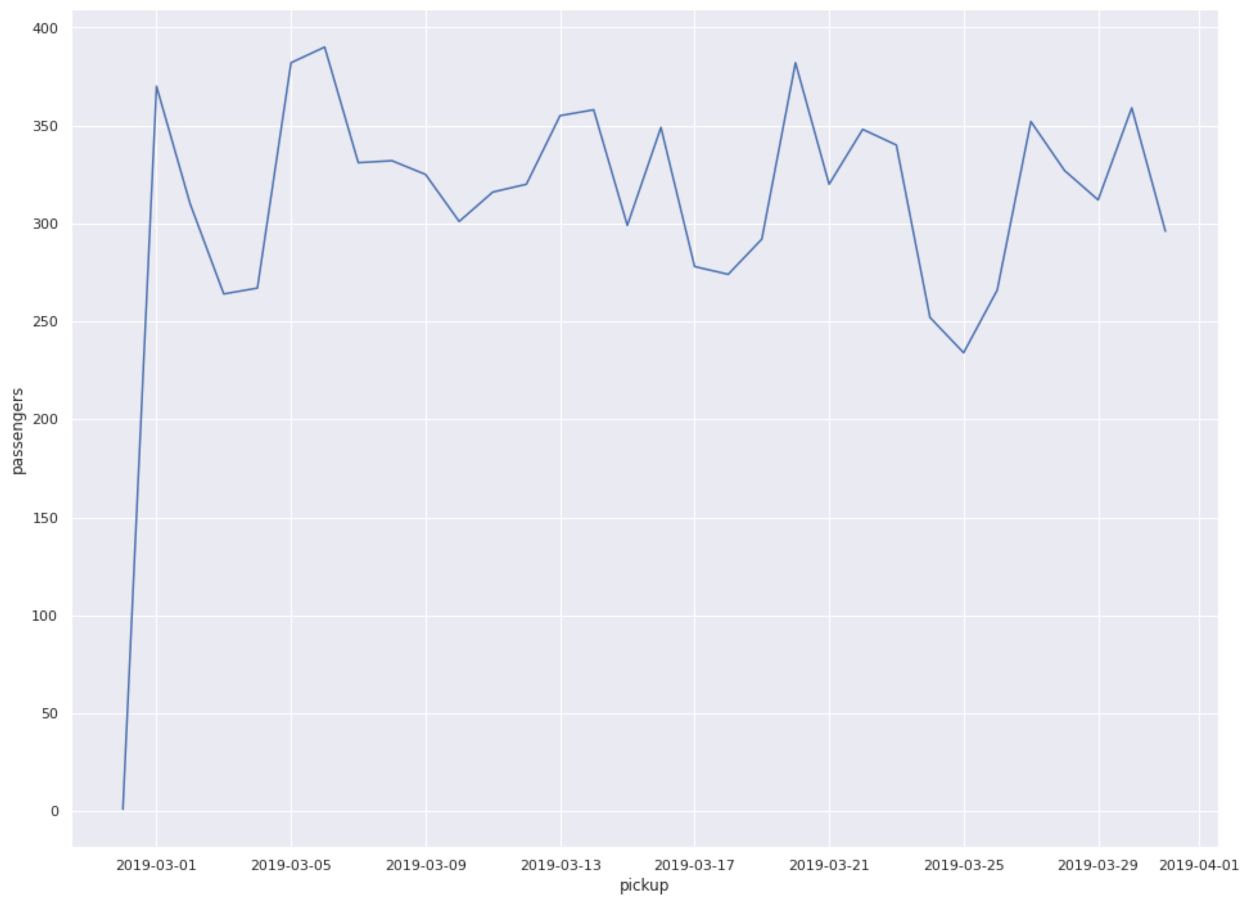
```
pass_df = taxi_df.set_index('pickup').resample('D').sum()
pass_df.head()
```

```
            passengers  distance     fare     tip  tolls    total
pickup
2019-02-28           1      0.90     5.00    0.00   0.00     6.30
2019-03-01         370    640.29  2946.97  442.47  60.34  4213.83
2019-03-02         310    548.70  2358.00  333.97  28.80  3319.02
2019-03-03         264    554.04  2187.89  307.47  34.56  3027.32
2019-03-04         267    583.81  2335.74  334.98  63.36  3269.08
```

You can almost see the plot here, we have an index of dates and a feature 'passengers', these two will make the backbone of our visualisation.

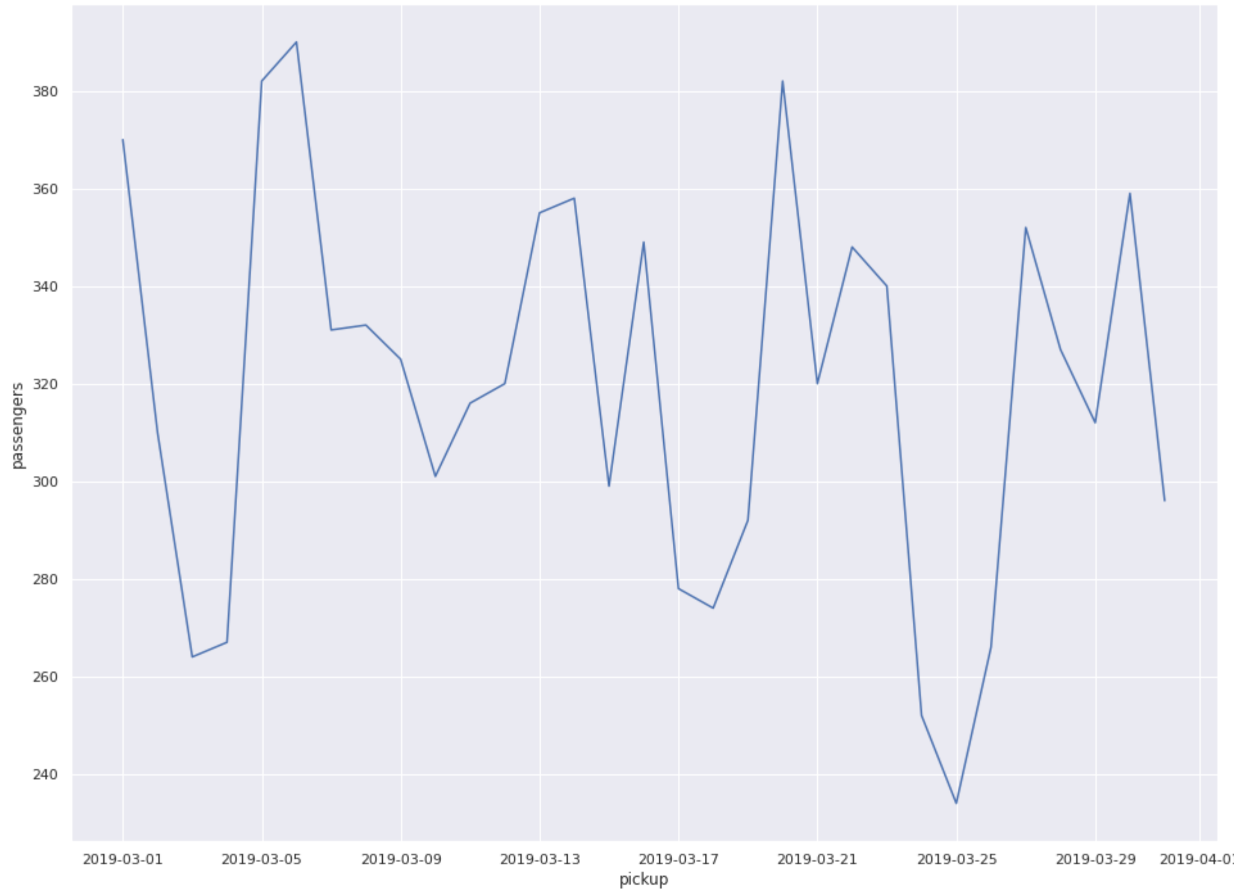
```
sns.lineplot(x=pass_df.index, y=pass_df.passengers)
```

```
<AxesSubplot:xlabel='pickup', ylabel='passengers'>
```



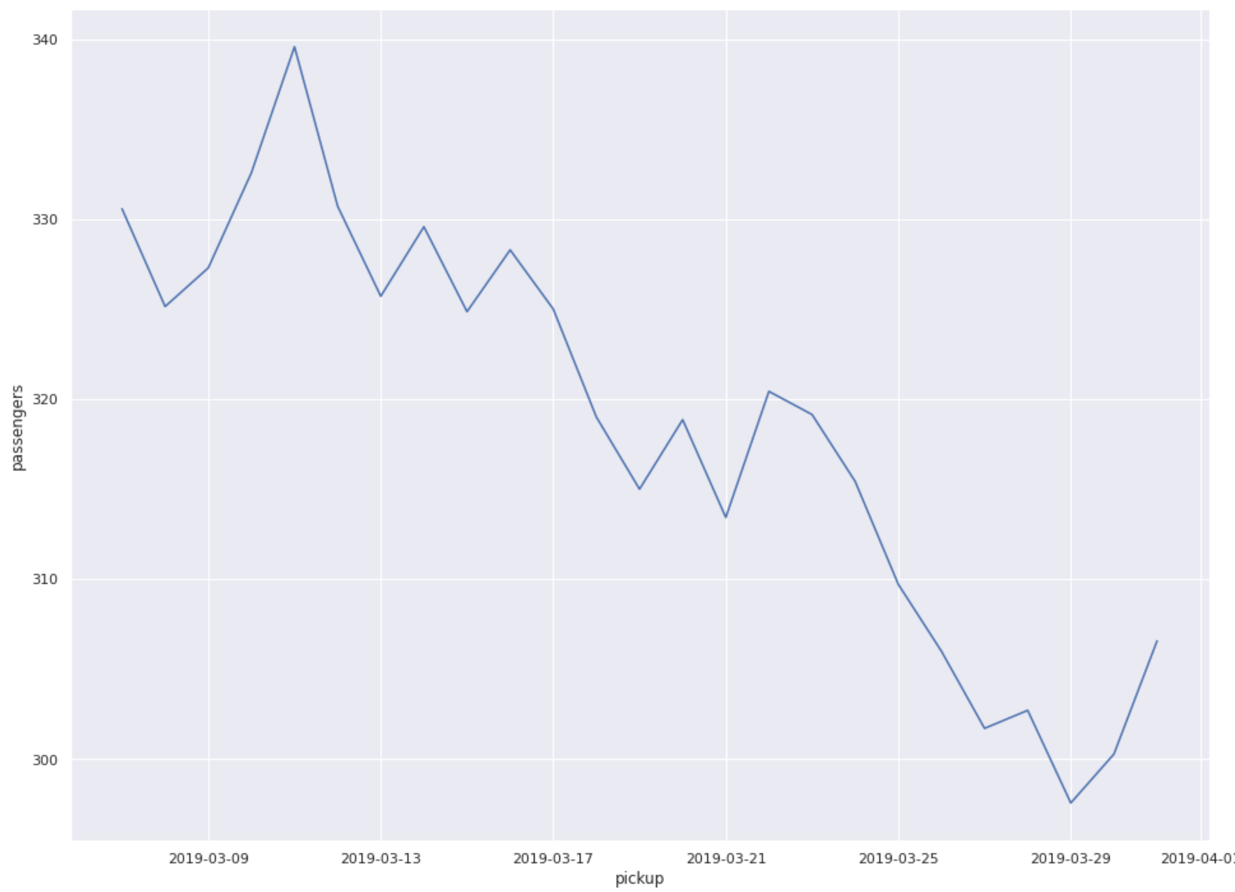
Looks about right, however I don't like the start of it, the data started late on that first day, resampling shows we only have 1 passenger for that day. This is not representable, so we remove that record.

```
pass_df = pass_df.loc['2019-03-01':]
ax = sns.lineplot(x=pass_df.index, y=pass_df.passengers)
```



Much better, however the plot feels like there is a lot of fluctuations, so it would be practical to apply a rolling sum or mean. This rolling operation takes the last x values and applies an operation (sum, mean,…) to it, creating a smoother graph and is visually more sensitive to trends.

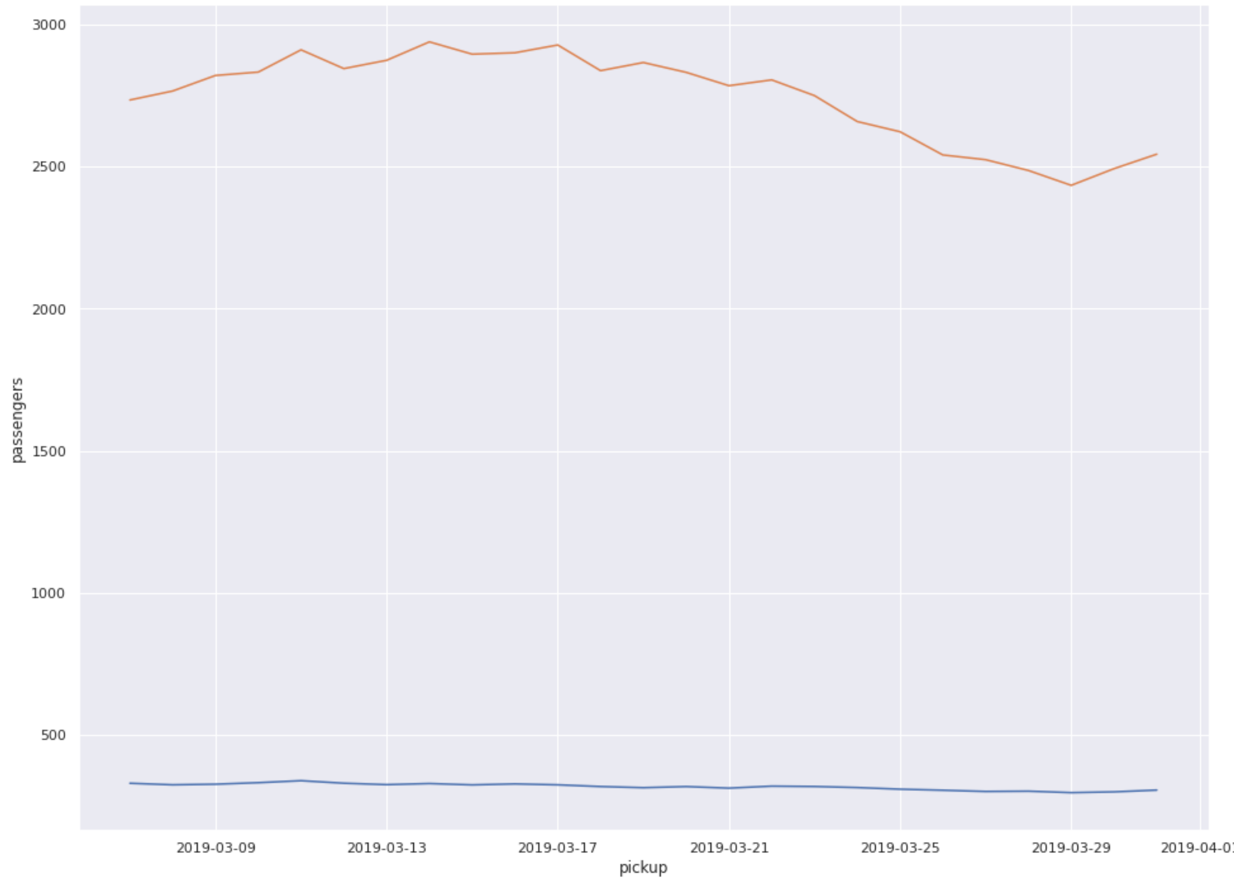
```
rolling_pass_df = pass_df.rolling(7).mean()
ax = sns.lineplot(x=rolling_pass_df.index, y=rolling_pass_df.passengers)
```



By applying a rolling mean, we can see that the average amount of passengers per day is decreasing. I feel there is no need to panick, as this is only 1 month of data and seasonal fluxtuations do happen.

Something else that triggers my curiosity is the amount these passengers paid, can we perhaps see a trend there? It would be ideal to plot these together so the comparison is simple.

```
ax = sns.lineplot(x=rolling_pass_df.index, y=rolling_pass_df.passengers)
ax = sns.lineplot(x=rolling_pass_df.index, y=rolling_pass_df.fare, ax=ax)
```



As we only have a few passengers per trip, yet trips can be costly the ranges of these 2 features are completely different. Before we think about scaling, we actually do want to know the scale here, we just cant fit them in the same graph.

A first approach would be to use a secondary axis, where the right side of the y-axis is used to show the fare scale. You can see that the graph is already getting more complicated code-wise, this is where using the right library is key as they usually have built in features for that.

```
ax = sns.lineplot(x=rolling_pass_df.index, y=rolling_pass_df.passengers, label=
↪'passengers', legend=False)
ax2 = ax.twinx()
ax = sns.lineplot(x=rolling_pass_df.index, y=rolling_pass_df.fare, ax=ax2, color='r',␣
↪label='fare', legend=False)
ax.figure.legend()
```

```
<matplotlib.legend.Legend at 0x7fc9f08c1fd0>
```

Interesting! It shows that there was a period where they did not follow eachother perfect, yet the trend is almost exact for these features.

Another method where you can compare them would require feature engineering, where we calculate the fare per passenger per day, apply the rolling window and plot. Perhaps you could figure that out? create a new feature that divides the fare by the passengers, recreate the rolling dataframe and use seaborn to plot the results.

At the start we used the sum of passengers per day, however we could also visualise the average amount of passengers per ride. The reason why I would like to do this is because earlier I saw a difference in trend for the fare and the amount of passengers, an explanation for this could be that the average amount of passengers dropped, resulting in lower passengers, yet the total expenditure of fares would remain constant.

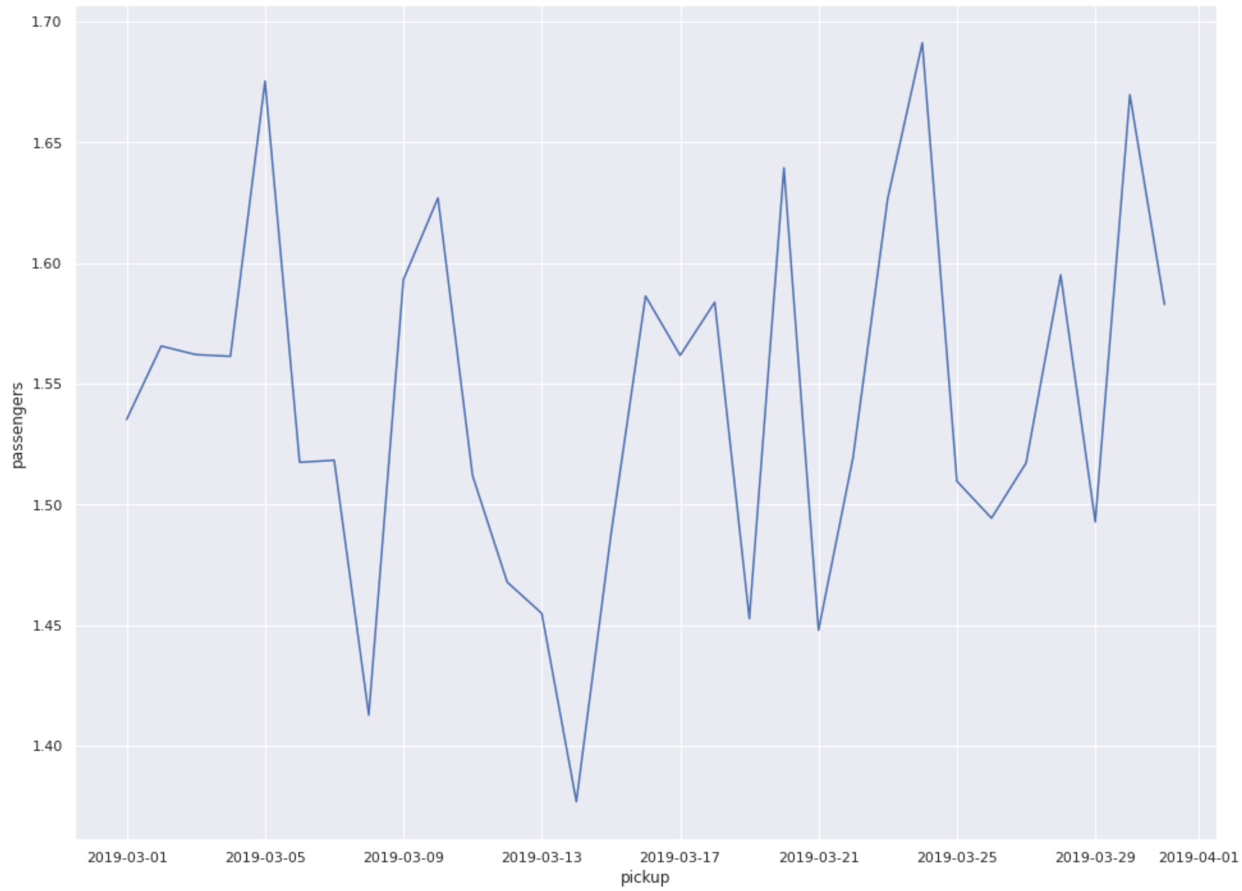
Let us figure this out, we here calculate the average (mean) of the passengers per day.

```
avg_pass_df = taxi_df.set_index('pickup').resample('D').mean()
avg_pass_df.head()
```

```
            passengers  distance       fare       tip     tolls      total
pickup
2019-02-28    1.000000  0.900000   5.000000  0.000000  0.000000   6.300000
2019-03-01    1.535270  2.656805  12.228091  1.835975  0.250373  17.484772
2019-03-02    1.565657  2.771212  11.909091  1.686717  0.145455  16.762727
2019-03-03    1.562130  3.278343  12.946095  1.819349  0.204497  17.913136
2019-03-04    1.561404  3.414094  13.659298  1.958947  0.370526  19.117427
```

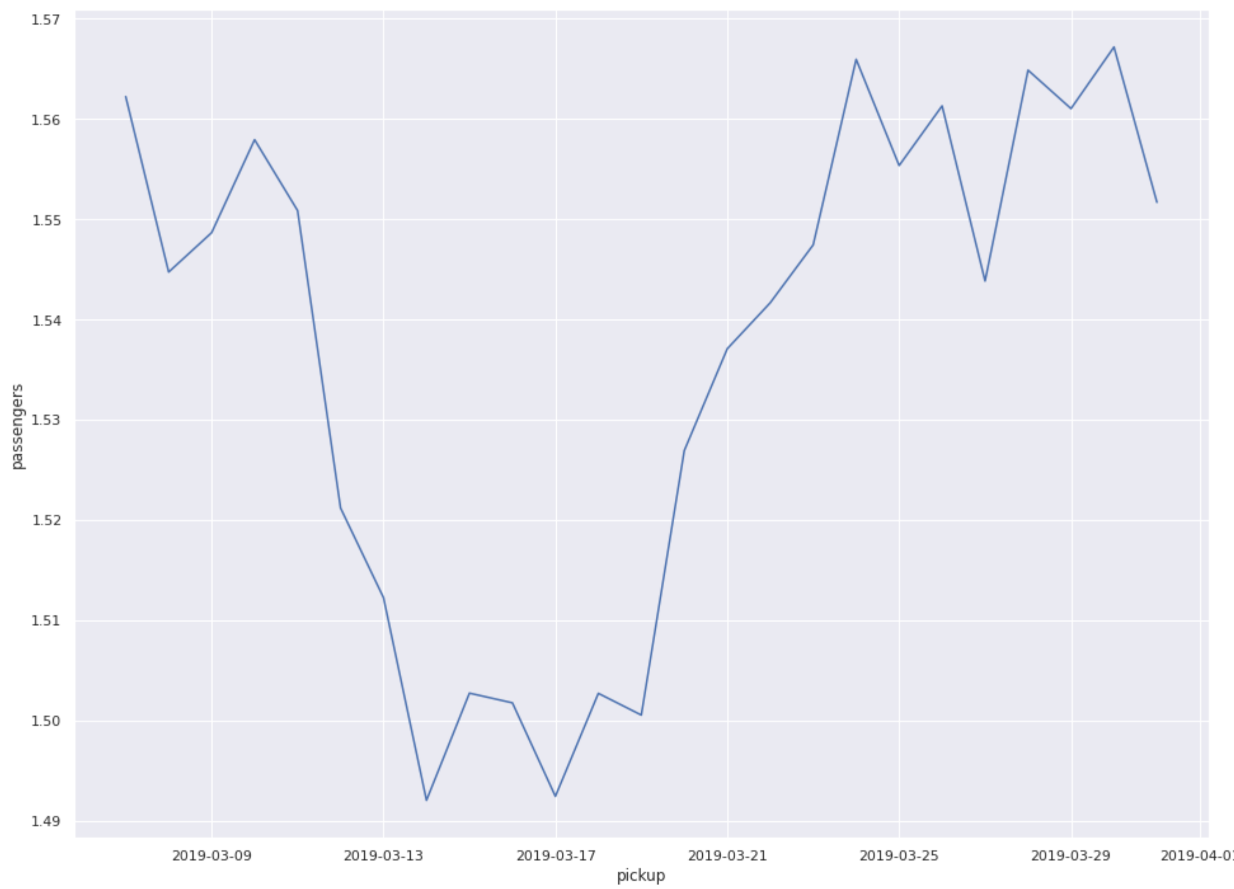
Doing more or less exactly the same we can create a simple plot with the average amount of passengers in a taxi.

```
avg_pass_df = avg_pass_df[1:]
ax = sns.lineplot(x=avg_pass_df.index, y=avg_pass_df.passengers)
```



For the same reasons, this plot is not suitable as it has too much variance. We apply a rolling mean of 7 days and re-evaluate.

```
rolling_avg_pass_df = avg_pass_df.rolling(7).mean()
ax = sns.lineplot(x=rolling_avg_pass_df.index, y=rolling_avg_pass_df.passengers)
```



We find a dip in passengers per ride that looks to be in the same time interval, therefore we could conclude here that fares did not get more expensive, rather the sharing of cabs was less. You could try and find a method to add the data of these two graphs together, yet this is already advanced visualisation.

Another question that I have for you, do you think that the dip is relevant? Not specifically from a business point of view, rather from a statistical view, Perhaps if you look at the range of the y-axis you might feel that our plot is a bit magnified. This is a good example of how you can use ranges of your axi to make data more dramatic. Be weary of these malpractices!

We are not done yet, as our dataset contains much more information. Harnessing the powers of the preprocessing we learned, we could include other (mostly categorical) feature into our line plot.

Here we take the payment option (either cash or card) and use it to create 2 time series in long format (2 datasets below each other).

```
pass_payment_df = taxi_df.groupby('payment').apply(
    lambda x: x.set_index('pickup').resample('D').sum()
)
pass_payment_df
```

```
                        passengers  distance    fare     tip  tolls    total
payment     pickup
cash        2019-02-28           1      0.90    5.00    0.00   0.00     6.30
            2019-03-01         104    112.31  571.50    0.00   5.76   748.76
            2019-03-02          86    159.46  690.50    0.00   5.76   863.96
            2019-03-03          67    172.34  641.50    0.00  17.28   782.18
            2019-03-04          71    130.60  571.50    0.00   0.00   710.95
```

(continues on next page)

(continued from previous page)

```
...                                ...      ...      ...      ...     ...      ...
credit card 2019-03-27             263   532.61  2260.64   485.63   69.12  3342.29
            2019-03-28             227   403.41  1886.07   404.45   40.32  2802.94
            2019-03-29             211   404.61  1831.98   410.13   23.04  2747.85
            2019-03-30             268   540.71  2211.10   487.97   78.62  3249.49
            2019-03-31             202   376.78  1632.93   345.83   29.16  2408.42

[63 rows x 6 columns]
```

Seaborn does not like this long format type, therefore we unstack the first index and create a wide format. For those wo are punctilious, you can notice we created a missing value, with wat should we fill it? (Our luck that seaborn can handle missing values!)

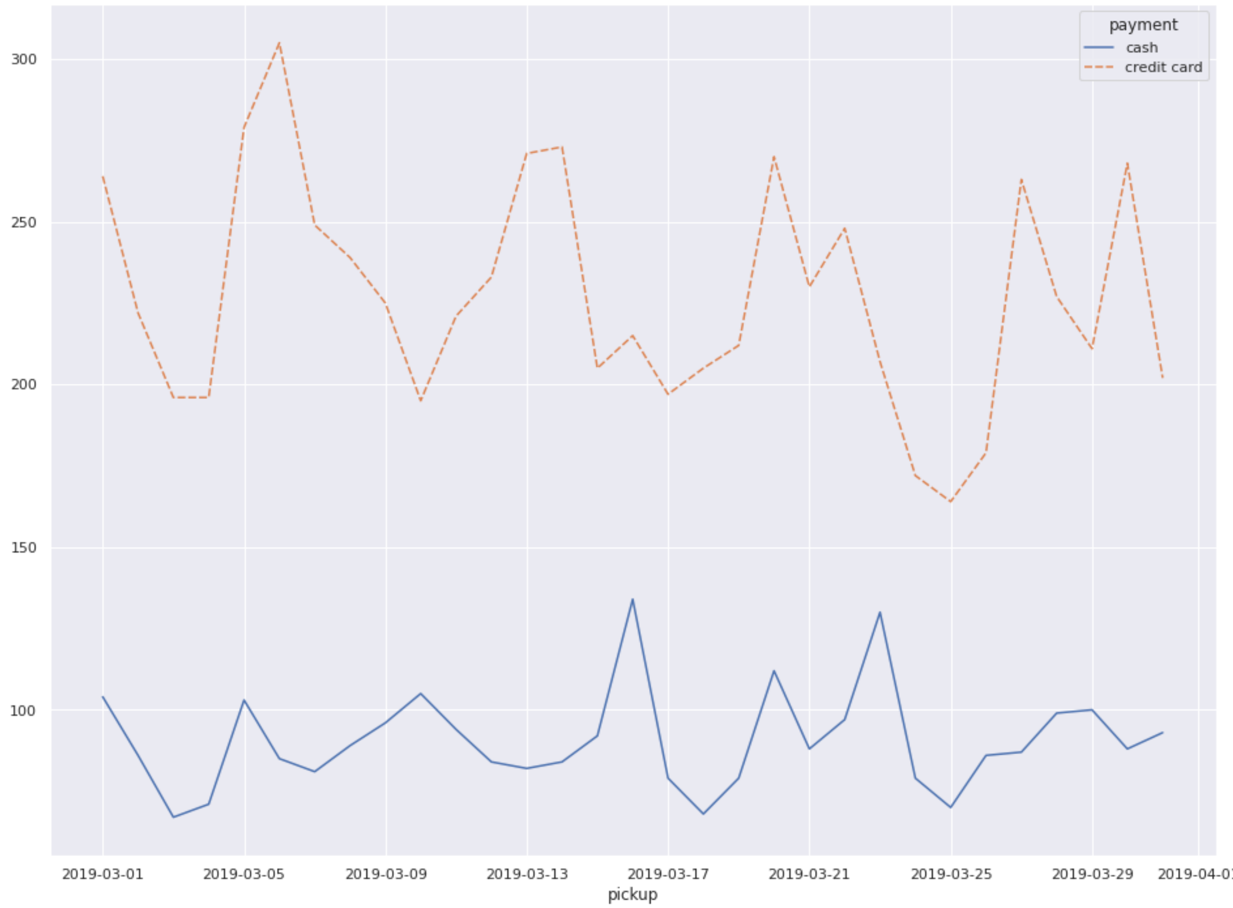
```
pass_payment_df.unstack(0).head()
```

```
           passengers             distance              fare              \
payment          cash credit card     cash credit card   cash credit card
pickup
2019-02-28        1.0         NaN     0.90         NaN    5.0         NaN
2019-03-01      104.0       264.0   112.31      527.08  571.5     2363.97
2019-03-02       86.0       222.0   159.46      377.74  690.5     1651.50
2019-03-03       67.0       196.0   172.34      381.60  641.5     1526.39
2019-03-04       71.0       196.0   130.60      453.21  571.5     1764.24

            tip              tolls              total
payment    cash credit card   cash credit card   cash credit card
pickup
2019-02-28  0.0         NaN   0.00         NaN   6.30         NaN
2019-03-01  0.0      442.47   5.76       54.58 748.76     3446.47
2019-03-02  0.0      333.97   5.76       23.04 863.96     2430.36
2019-03-03  0.0      307.47  17.28       17.28 782.18     2224.34
2019-03-04  0.0      334.98   0.00       63.36 710.95     2558.13
```

Same data, different structure, now seaborn understands the format and we can go back to visualisation.

For simplicity we start with a simple passengers line plot

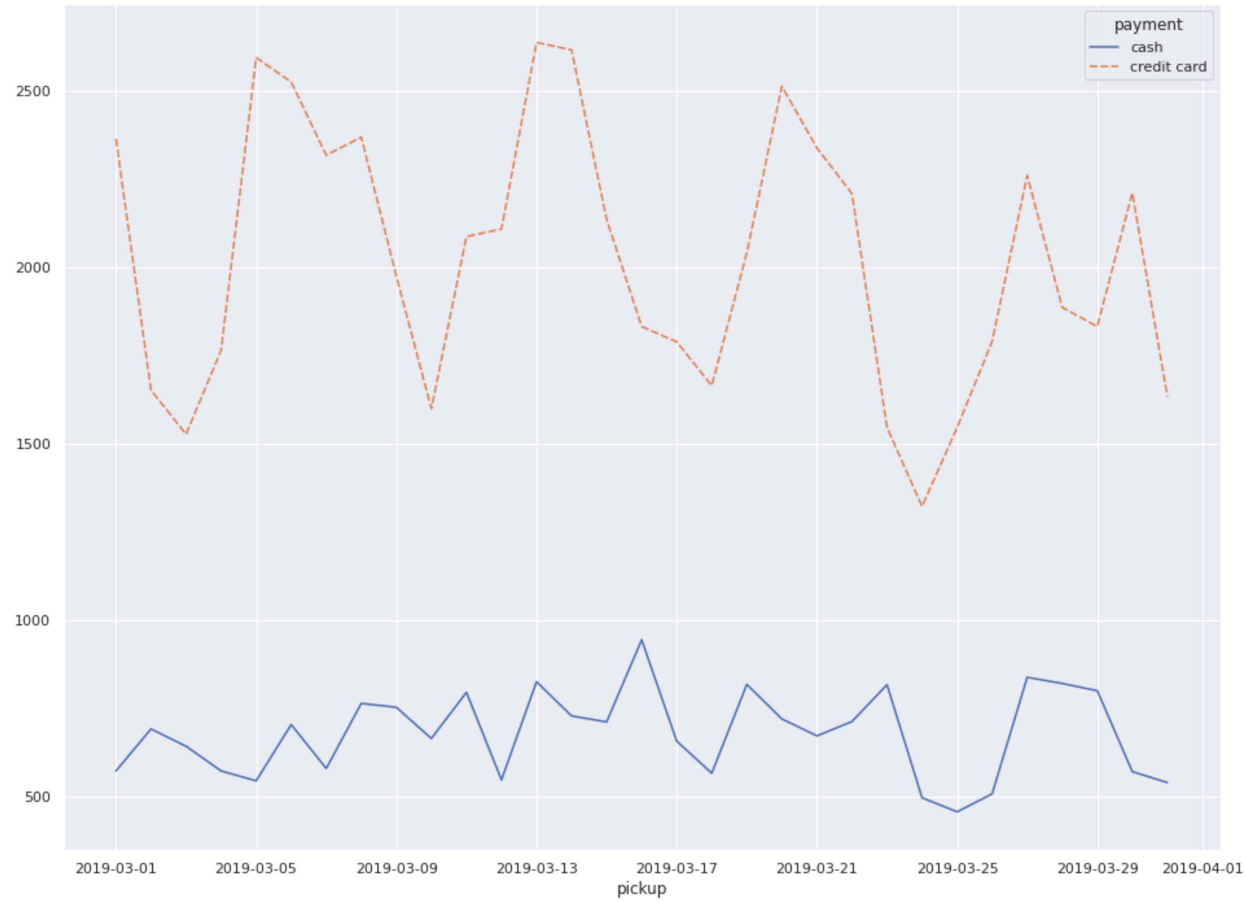
```
ax = sns.lineplot(data=pass_payment_df.passengers.unstack(0)[1:])
```



You can see that there are generally more people paying by card, which is more convenient in such an occasion. Note that here we should not use a seperate y-axis as we are comparing 2 sets of data that are similar by origin.

We do the same for fares.

```
ax = sns.lineplot(data=pass_payment_df.fare.unstack(0)[1:])
```



This is more or less a no-brainer, as more people pay by card, the fares by card are also more. So we can't really compare fares with this plot, we have to be creative.

I opted to go for an average fare per passenger, as this is in my opinion more relevant than the amount of rides

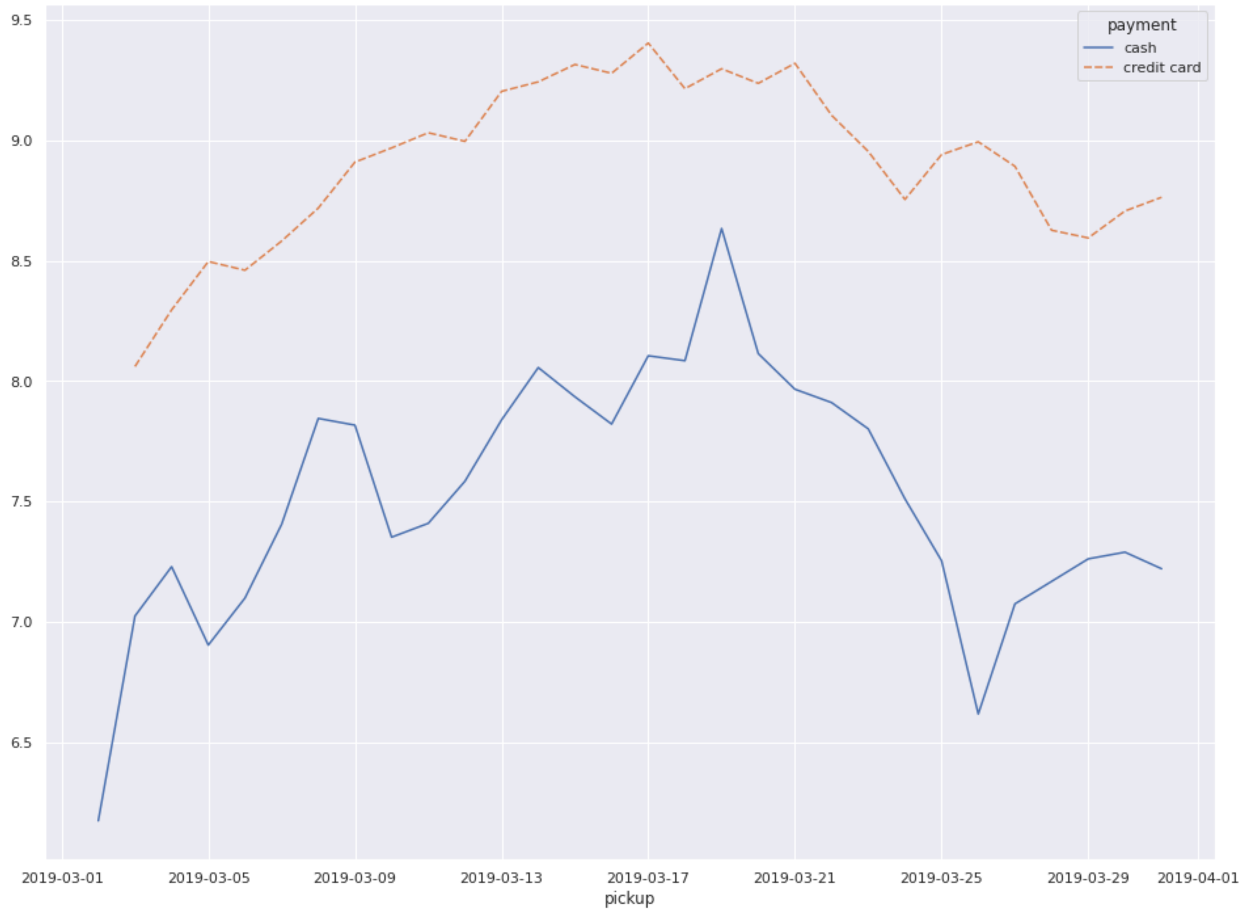
```
pass_payment_df['fare_pass'] = pass_payment_df.fare/pass_payment_df.passengers
pass_payment_df.head()
```

```
                    passengers  distance   fare  tip  tolls   total  fare_pass
payment pickup
cash    2019-02-28           1      0.90    5.0  0.0   0.00    6.30   5.000000
        2019-03-01         104    112.31  571.5  0.0   5.76  748.76   5.495192
        2019-03-02          86    159.46  690.5  0.0   5.76  863.96   8.029070
        2019-03-03          67    172.34  641.5  0.0  17.28  782.18   9.574627
        2019-03-04          71    130.60  571.5  0.0   0.00  710.95   8.049296
```

We created a new feature both containing info of fares and passengers, using this we create a new visualisations.

In this visualisation we show for both payment options the average fare amount per passenger in the cab.

```
ax = sns.lineplot(data=pass_payment_df.fare_pass.unstack(0).rolling(7, min_periods=3).
↪mean())
```

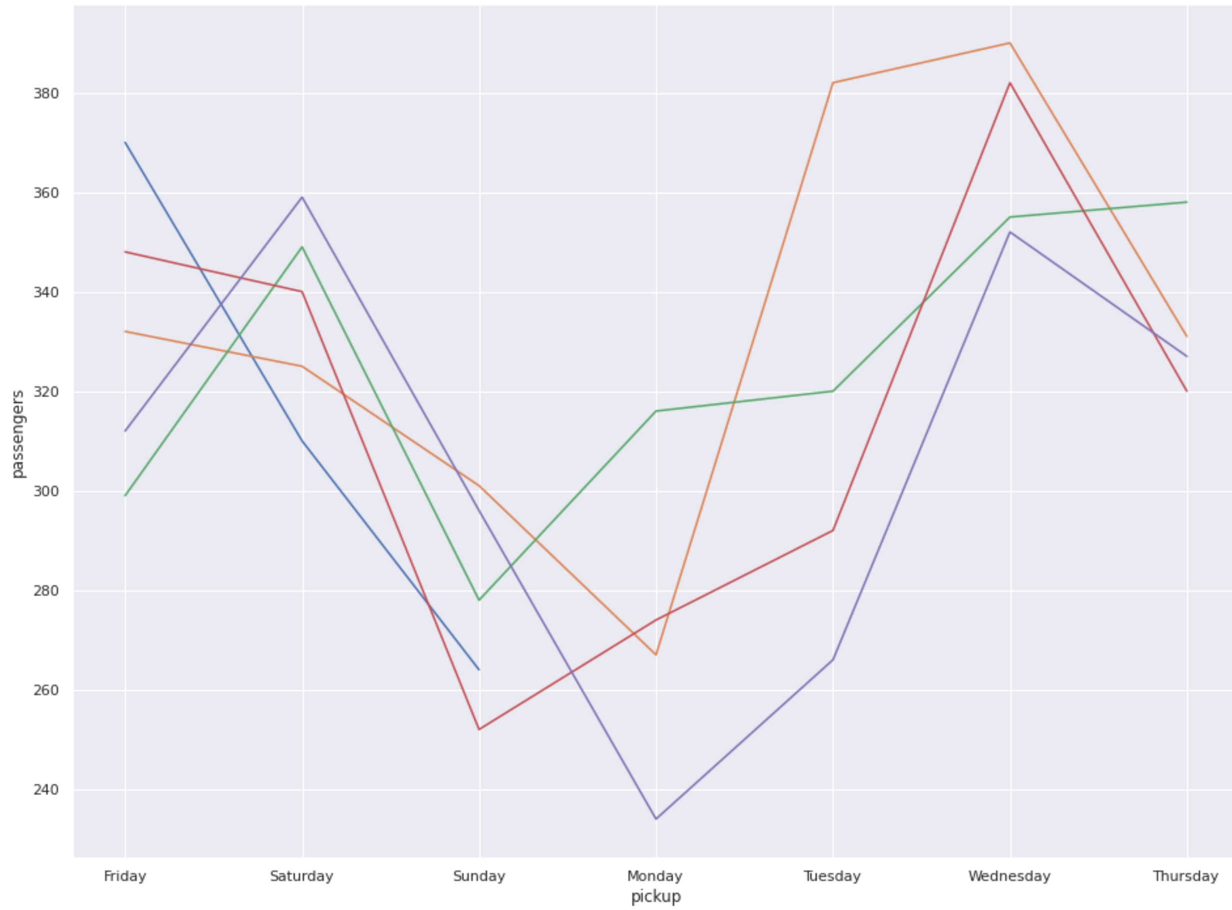


We can conclude that the average amount that has to be paid per person is lower for cash, indicating that people jump to their debit card as soon as the amount gets too high.

As a last I would like to emphasise that the x-axis, being time does not have to be linear. To illustrate this we create a weekly passenger rate and impose each week over the others.

```
pass_df.groupby(pd.Grouper(freq='W')).apply(
    lambda x: sns.lineplot(x=x.index.day_name(), y=x.passengers)
)
```

```
pickup
2019-03-03    AxesSubplot(0.125,0.125;0.775x0.755)
2019-03-10    AxesSubplot(0.125,0.125;0.775x0.755)
2019-03-17    AxesSubplot(0.125,0.125;0.775x0.755)
2019-03-24    AxesSubplot(0.125,0.125;0.775x0.755)
2019-03-31    AxesSubplot(0.125,0.125;0.775x0.755)
Freq: W-SUN, dtype: object
```



Here we can see there is a weekly trend occuring, where Sundays and Mondays are usually less busy days. The origin of this is hard to argue, as it might be less traffic, less taxi drivers working,…

Perhaps you could complete this visualisation by investigating the distance and/or tips?

CHAPTER

TWENTY

HISTOGRAM PLOT

When visualising one dimensional data without relating it to other information an option would be histograms. Histograms are used when describing distributions in your data, it is not the values itself you are visualising, rather the counts/frequencies of each value.

We again start with importing our libraries

```
import pandas as pd
import seaborn as sns
sns.set_theme()
sns.set(rc={'figure.figsize':(16,8)})
```

For this example we will be using the prepared dataset from seaborn containing mileages of several cars. Information about the cars is also given.

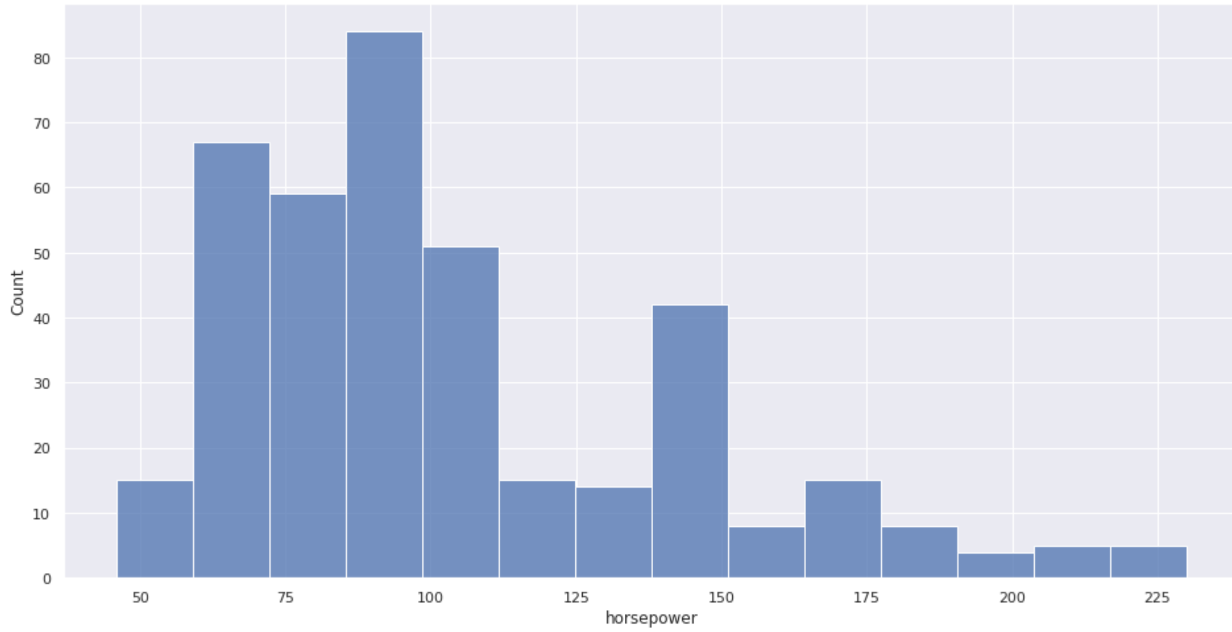
```
mpg_df = sns.load_dataset('mpg')
mpg_df.head()
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year origin                       name
0          70    usa  chevrolet chevelle malibu
1          70    usa          buick skylark 320
2          70    usa         plymouth satellite
3          70    usa              amc rebel sst
4          70    usa                ford torino
```

We start of simple by plotting the distribution of horsepower in our dataset.

```
sns.histplot(data=mpg_df, x='horsepower')
```

```
<AxesSubplot:xlabel='horsepower', ylabel='Count'>
```

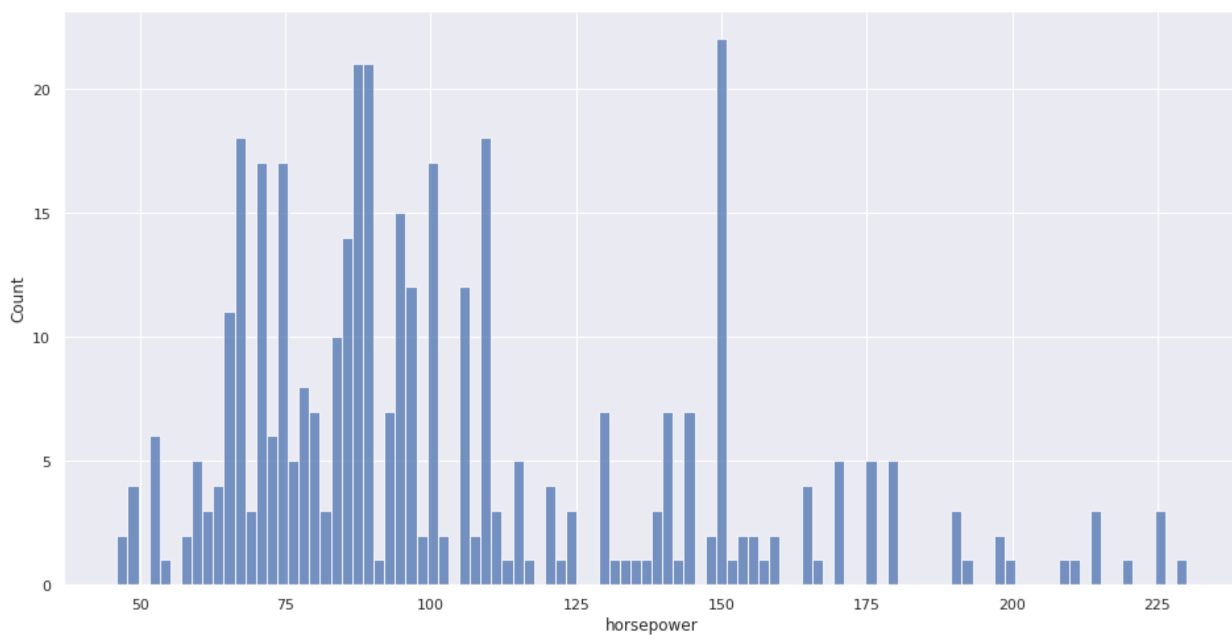


A first thing that is visible is that our feature is not normally distributed, we have a long tail to the higer end.

For histograms we can specify the amount of bins in which we seperate the counts, seaborn selects a suitable number yet we can change this.

```
sns.histplot(data=mpg_df, x='horsepower', bins=100)
```

```
<AxesSubplot:xlabel='horsepower', ylabel='Count'>
```

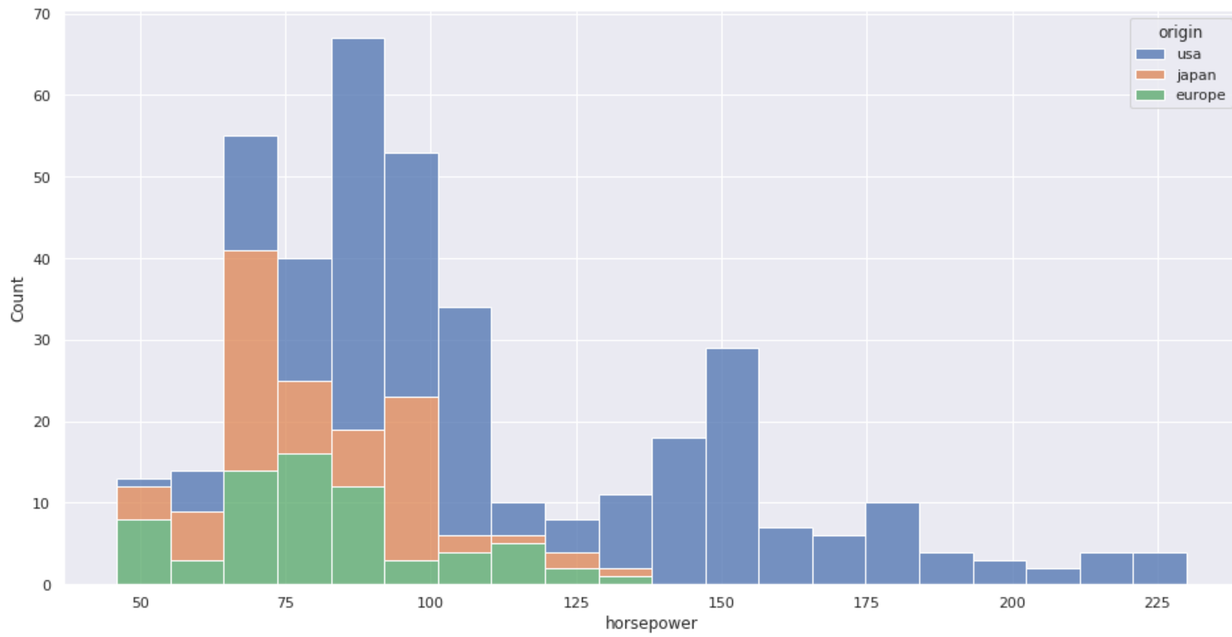


As you can see, the previous option looks a lot better. Taking the right amount of bins is important.

In order to add more information to our plot, we can use categorical data to split our data into multiple histograms. Here we used the origin of the cars to split into 3 categories, notice how each of them has their own area, japan and europe are on the lower end whilst usa is centered in higher horsepower.

```
sns.histplot(data=mpg_df, x='horsepower', hue='origin', bins=20, multiple='stack')
```

```
<AxesSubplot:xlabel='horsepower', ylabel='Count'>
```

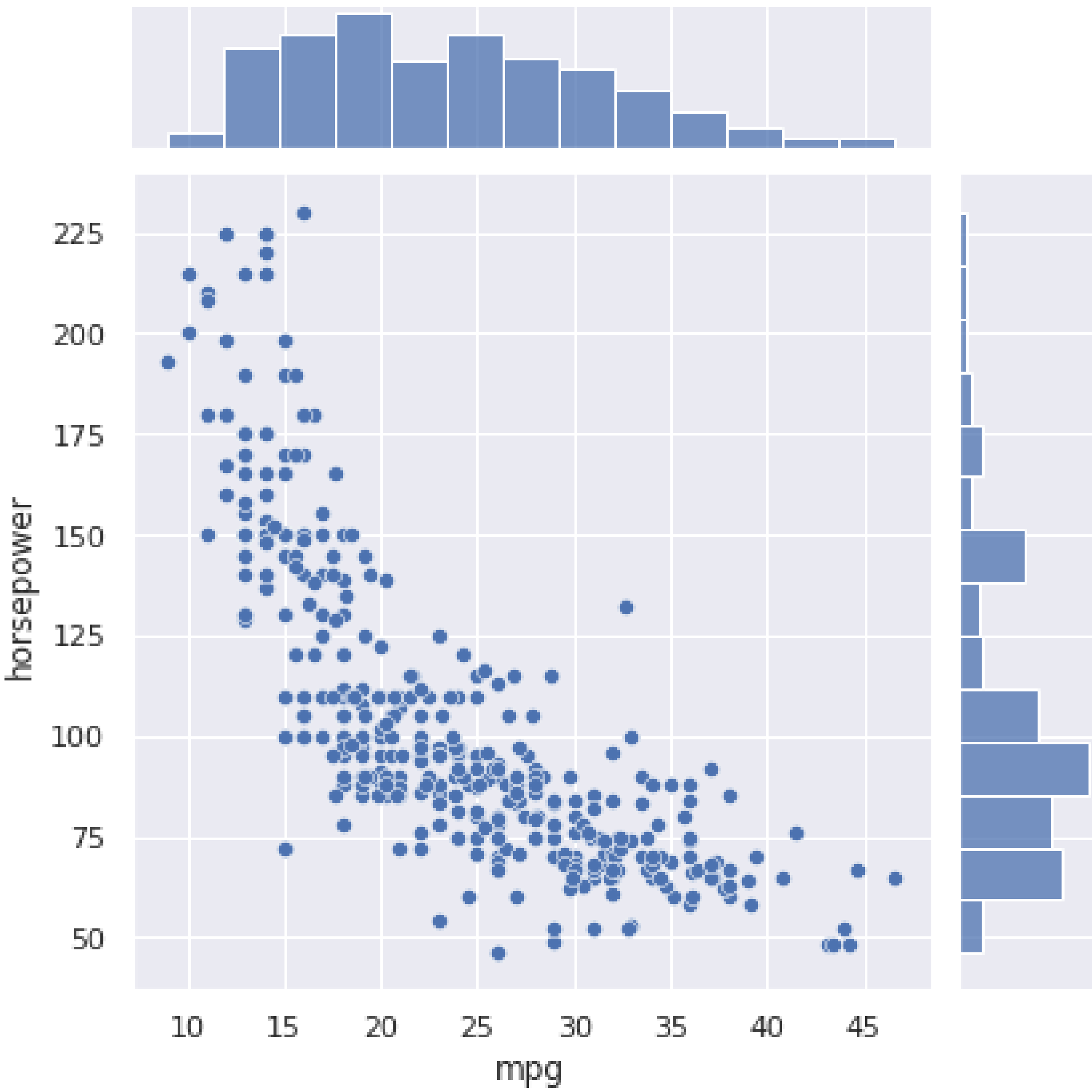


A neat feature of seaborn is that it can join histograms and scatter plots (in the next section) together.

Here we see how the visualisations of 2 one dimensional histograms perfectly combine together into a scatter plot, where 2 dimensional data is shown (both mileage and horsepower).

```
sns.jointplot(data=mpg_df, x='mpg', y='horsepower')
```

```
<seaborn.axisgrid.JointGrid at 0x7ff9e42c8fa0>
```



Histograms are a really powerfull tool when it comes to validating your data, we can easily the distribution of each feature, see if they are normally distributed and visualise distributions of subgroups.

Yet for final visualisations they are often not interesting enough.

CHAPTER

TWENTYONE

BOX PLOT

In the previous section we looked into visualising the distributions of 1 dimensional data. We used histograms for this, but there is a second more statistical option for this, the Boxplot.

To be brief, the boxplot shows a box containing the InterQuartile Data that we already talked about and also has 2 whiskers, showing the threshold for outliers. Actual outliers are then printed seperately, making this plot ideal for outlier detection aswel as distributions.

I personally think this option is more suited for multiple categories compared to histograms, yet your mileage may vary.

```
import pandas as pd
import seaborn as sns
sns.set_theme()
sns.set(rc={'figure.figsize':(16,12)})
```

For this section we will look into the discovery of extrasolar planets, or planets that are ourside our own solar system. For each planet they listed the method of discovery, orbital period, mass, distance and year of discovery.

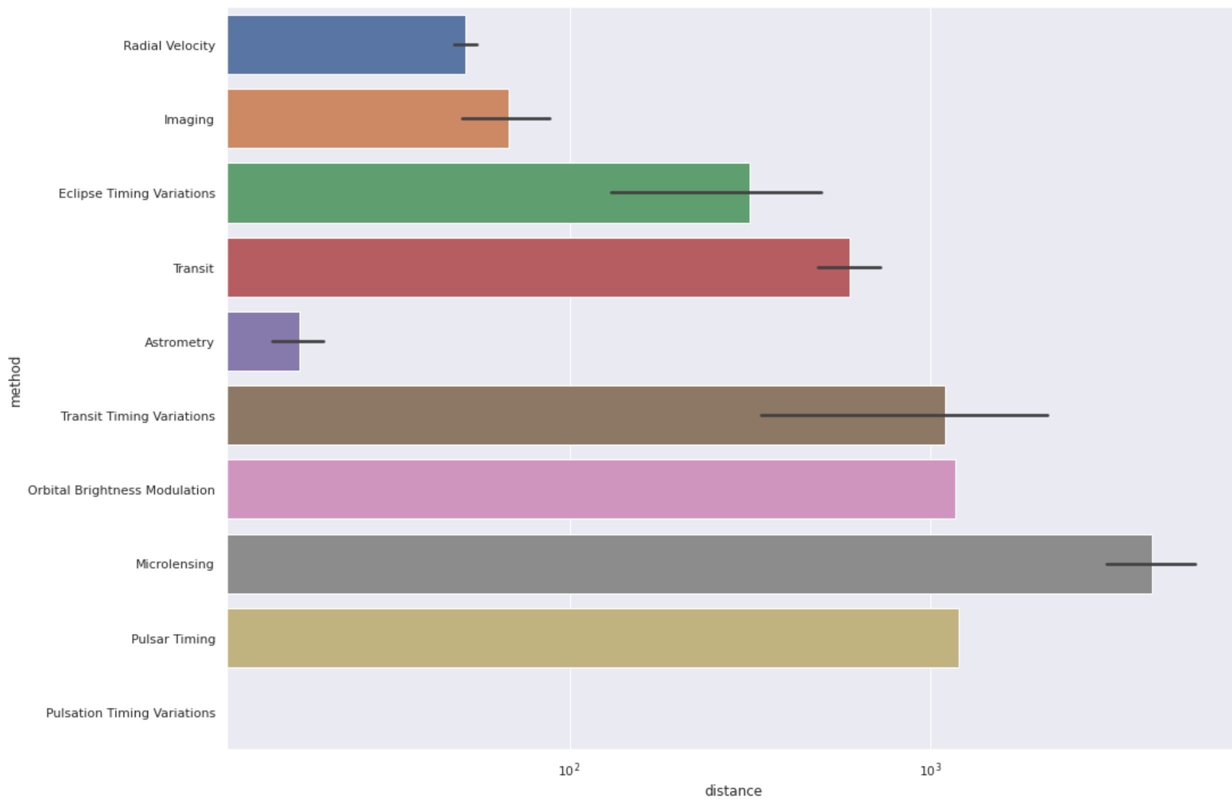
```
planet_df = sns.load_dataset('planets')
planet_df.head()
```

```
            method  number  orbital_period   mass  distance  year
0  Radial Velocity       1         269.300   7.10     77.40  2006
1  Radial Velocity       1         874.774   2.21     56.95  2008
2  Radial Velocity       1         763.000   2.60     19.84  2011
3  Radial Velocity       1         326.030  19.40    110.62  2007
4  Radial Velocity       1         516.220  10.50    119.47  2009
```

Let's say we would like to show the distances of each discovery method, if we would use a bar plot, the results might be hard to interpret.

```
ax = sns.barplot(data=planet_df, x='distance', y='method')
ax.set(xscale="log")
```

```
[None]
```



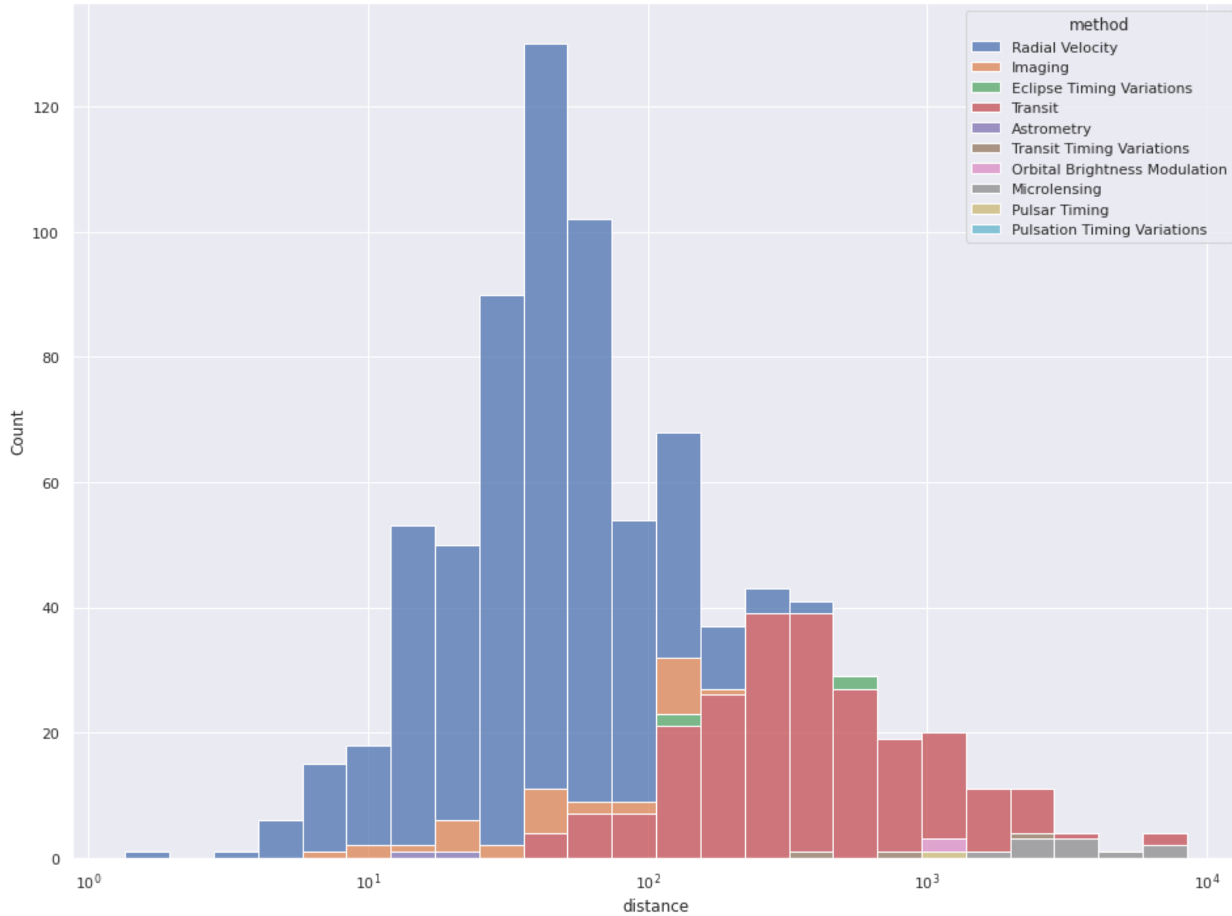
Whilst bar plots can be a good idea, here they are not.

Only use bar plots when visualising singular data points who are related to zero, not aggregations of multiple data points. Bar plots do not work if:

- your datapoints have no relation to zero
- your categories are related with different intervals
- you are dealing with groups of datapoints, not single datapoints (this case)

anyway, we could use a histogram similar to previous section, let's see how that turns out.

```
ax = sns.histplot(data=planet_df, x='distance', hue='method', multiple='stack', log_
↪scale=True)
```

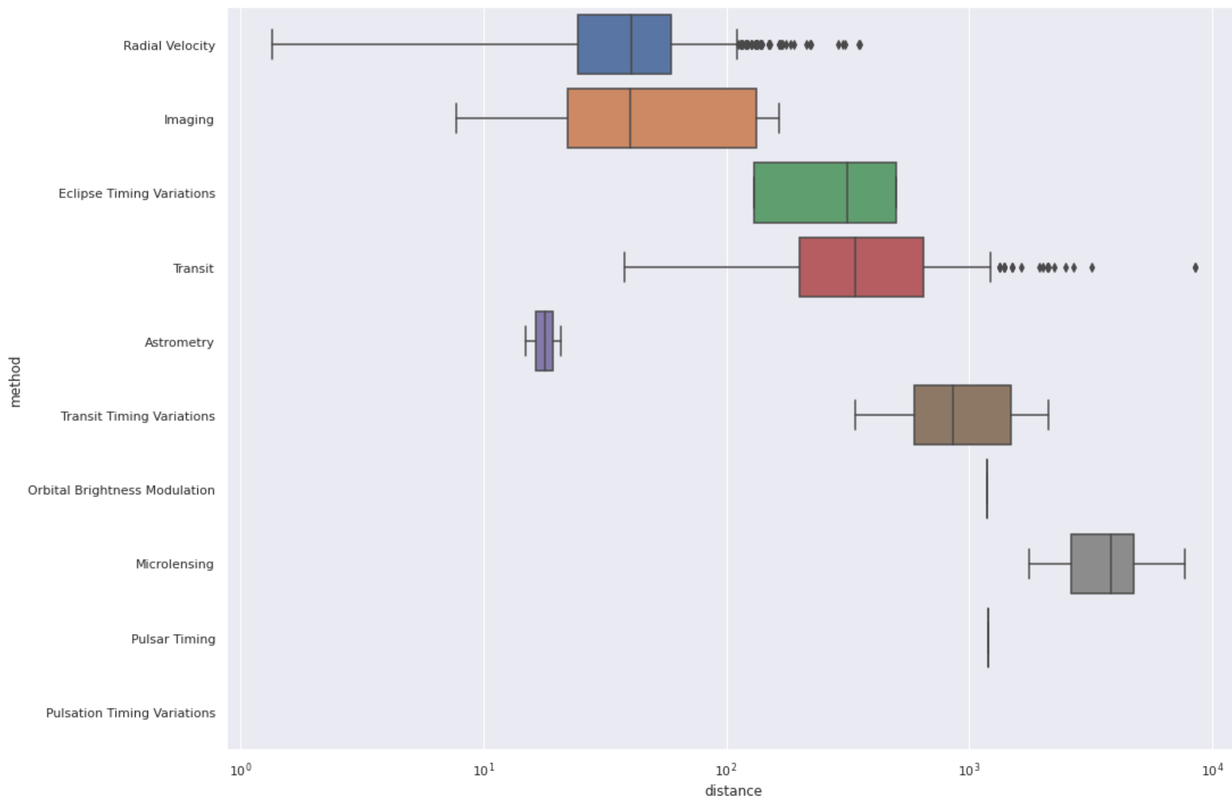


The histogram seems to be working, yet the methods with lower count are suppressed. A boxplot can overcome this and we can also compare medians of each method with eachother.

Take a few minutes to understand the next plot, at first it is very confusing, yet when adapted this is the most powerful visualisation of data exploration.

```
ax = sns.boxplot(data=planet_df, x='distance', y='method')
ax.set(xscale="log")
```

```
[None]
```



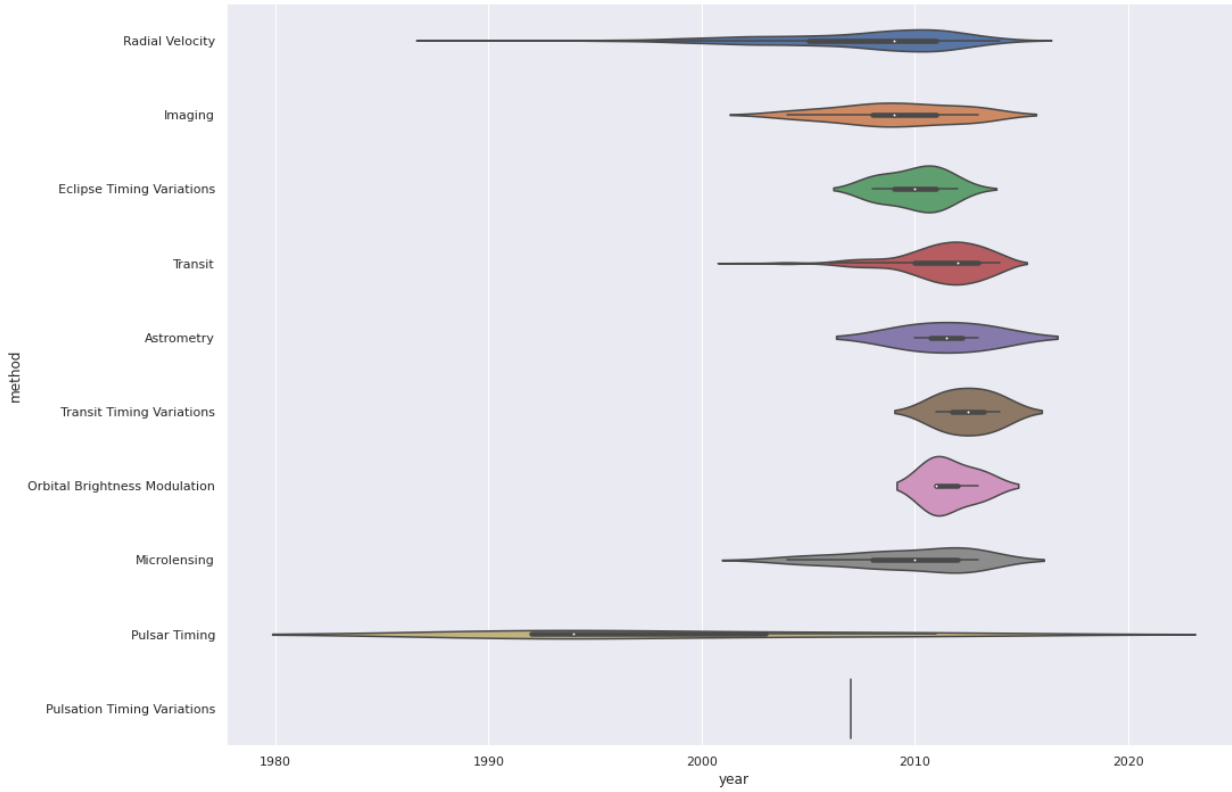
Can you see now why the bar plot here is a bad idea? Some methods have a broader distribution and relating our data to zero makes no real sense. With financial data this is different as budgets always start with 0.

Here we can conclude that some methods of detecting a planet requires a further or closer distance. You could say that if you want to discover a far extrasolar plant pick one of the last methods

An addition to the boxplot, where we focus more on distribution instead of statistics, would be the violin plot. Can you see why they would call it like that?

```
sns.violinplot(data=planet_df, x='year', y='method')
```

```
<AxesSubplot:xlabel='year', ylabel='method'>
```



As an exercise calculate the median, Q1 and Q3 of the distance per method and see if you come to the same conclusion as the boxplot

CHAPTER

TWENTYTWO

SCATTER PLOT

Thus far we dealt with one dimensional data in our visualisations, sometimes adding a category to divide our data. Here we take it a step further, scatter plots are to visualise the relation between 2 numerical features.

One remark that I would like to make here is that discrete numerical features (age, n_persons,...) are possible to use, yet when dealing with a small range (e.g. 0-10) the results are skewed.

```
import pandas as pd
import seaborn as sns
sns.set_theme()
sns.set(rc={'figure.figsize':(16,8)})
```

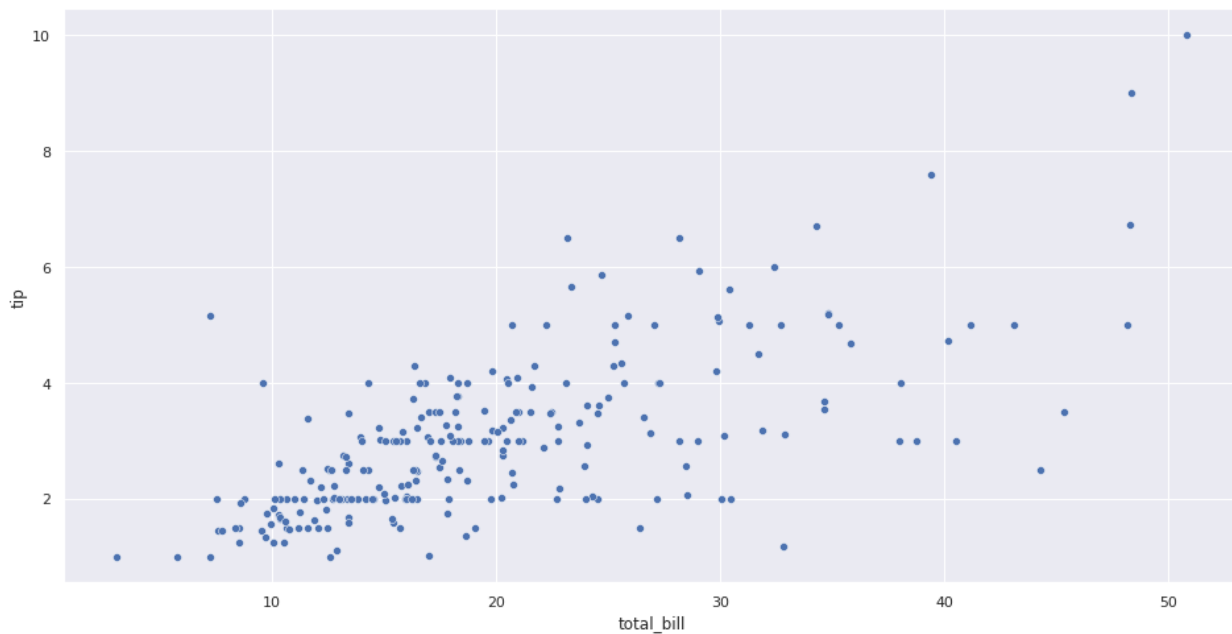
For scatter plots I opted to use a dataset containing tips from a restaurant, the tips are divided in gender, smoker, time of day and day of week.

```
tips_df = sns.load_dataset('tips')
tips_df.head()
```

```
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

The most simple scatter plot we can make would be showing the relation between the total bill and the tip, we would assume the tip is proportional to the size of the bill.

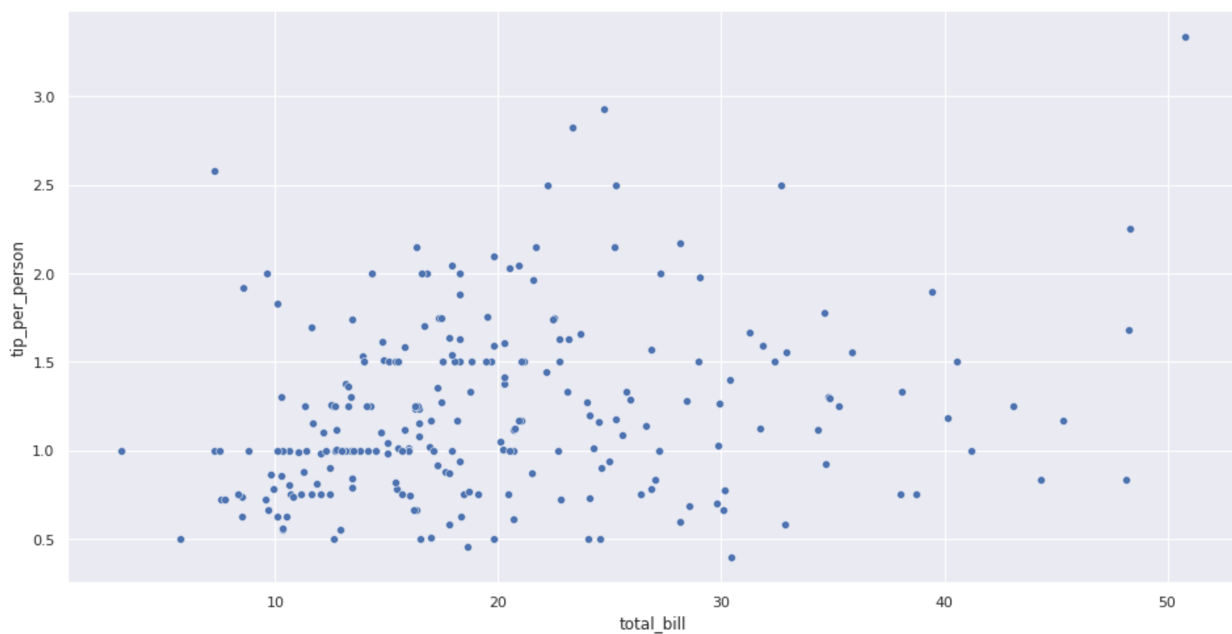
```
ax = sns.scatterplot(data=tips_df, x="total_bill", y="tip")
```



Just as expected, when the total bill rises, the tip grows too, we have some generous persons, and some less generous, but nothing out of the ordinary.

To get a better idea of the tipping habits we could calculate the tip per person in the bill, which is noted by size. We divide the tip by the amount of people and plot again.

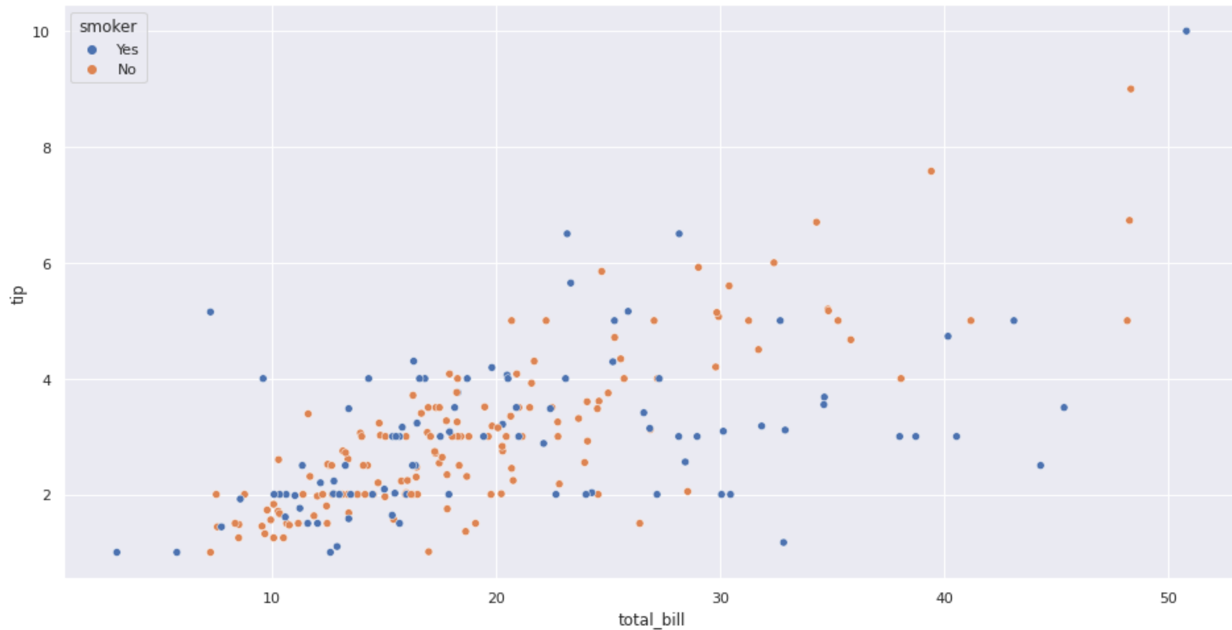
```
tips_df['tip_per_person'] = tips_df.tip/tips_df['size']
ax = sns.scatterplot(data=tips_df, x="total_bill", y="tip_per_person")
```



It is much harder to see a relation now, so we could argue that depending of the service everyone gives a specific amount. So it is not the size of the bill that is defining the tip, rather the amount of persons (although this is very similar) in the bill.

Aside from feature engineering, we can also add categorical features, using different colors for each feature. Here we added if they smoked or not.

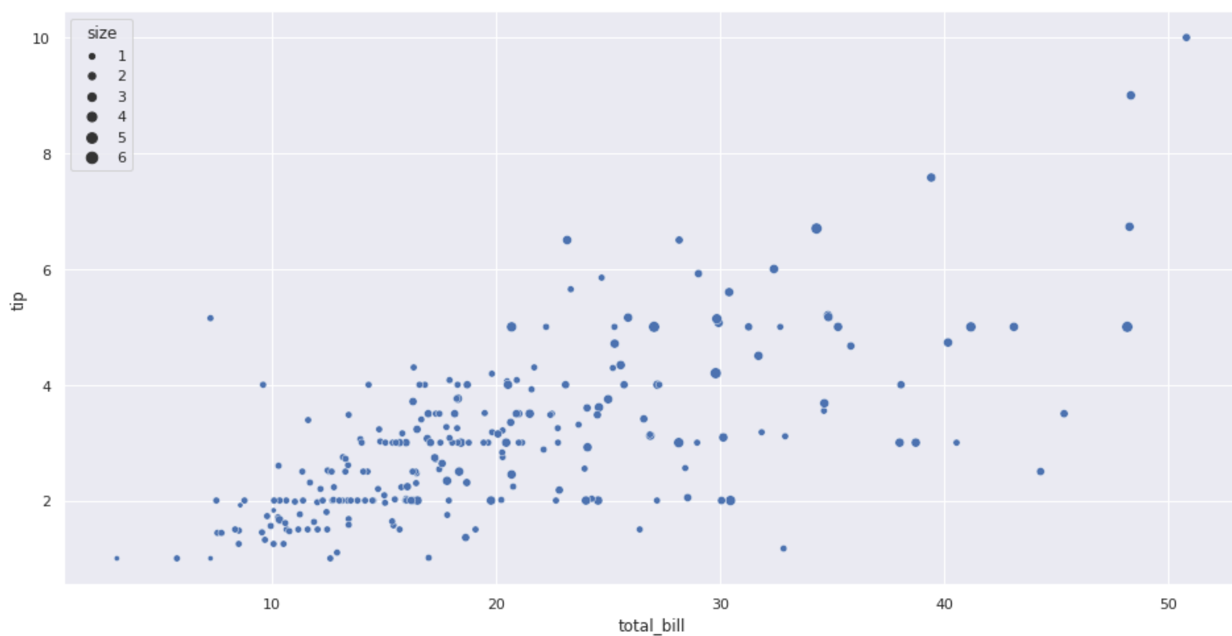
```
ax = sns.scatterplot(data=tips_df, x="total_bill", y="tip", hue='smoker')
```



It is hard to see if smoking had an effect on either the bill or the tip, which indicates that your plot is not that useful. This is not true if you wanted to prove that there is no effect of smoking obviously!

We can also add a numerical feature into the scatter plot, by using sizes of our dots in the scatter plot. The size of the group now influences the size of our dots.

```
ax = sns.scatterplot(data=tips_df, x="total_bill", y="tip", size="size")
```

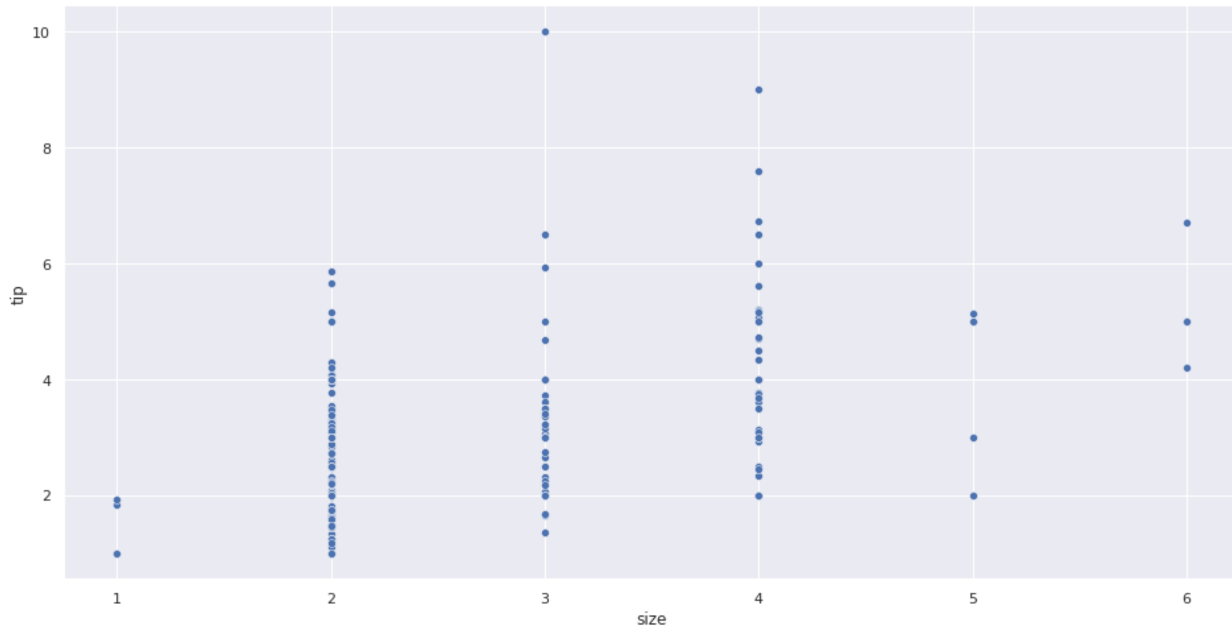


Whilst it might not be really visible because of the linear nature of the size - it is only going from 1 to 6 - the relation is not obvious. Perhaps you could do some feature engineering where you artificially increase the size by taking the square?

You could argue if that is still representable, but for the sake of the exercise let's say it is.

In the beginning I talked about numerical features with a low range, the size of our group is one of them. See what happens when i would use it in a scatter plot.

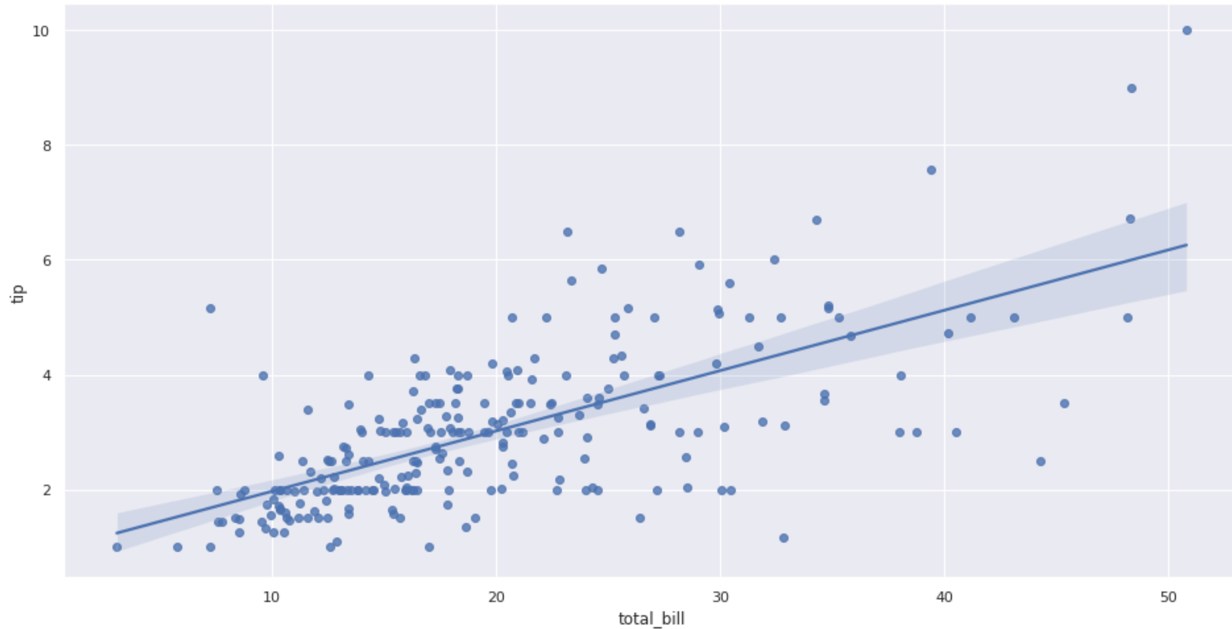
```
ax = sns.scatterplot(data=tips_df, x="size", y="tip")
```



It clearly shows that a higher size means statistical higher tips, up to a cut-off of 5 appearantly. Yet do you feel this is an aesthetically satisfying plot?

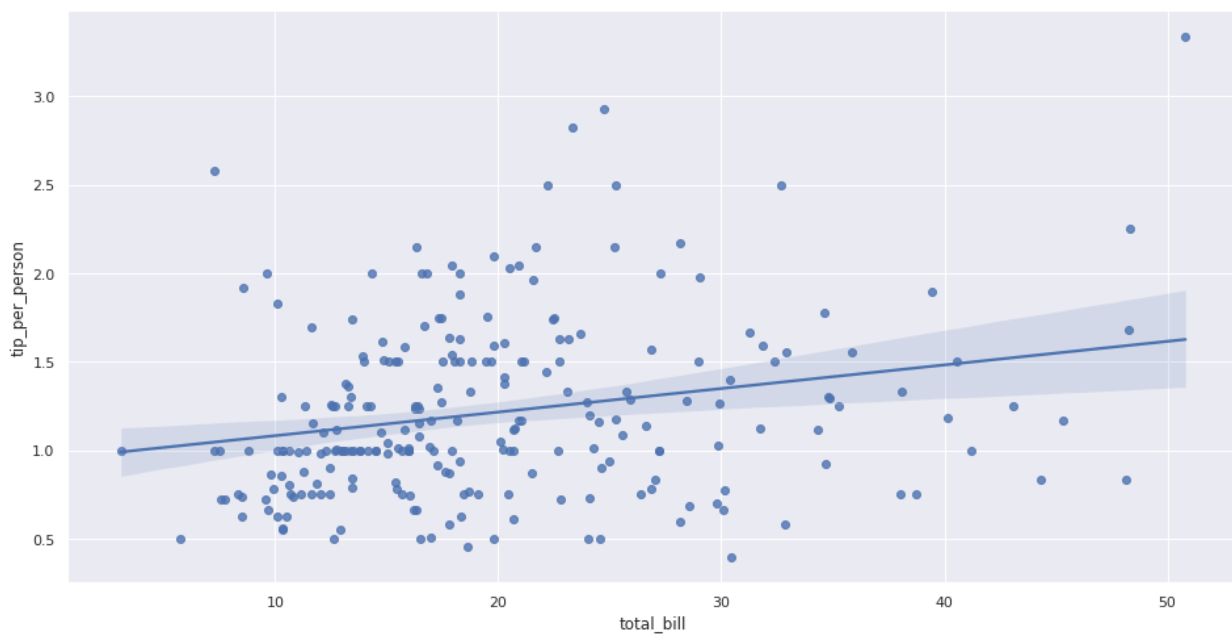
Not going to much in the field of machine learning, seaborn has an interesting feature built-in. They offer a regression plot, where a linear regression is draw with a confidence interval (the light blue area). Not wanting to give mathematical number it shows what it thinks is the relation between the 2 variables.

```
ax = sns.regplot(data=tips_df, x="total_bill", y="tip")
```



It seems to be very confident about the relation, how about where we corrected for group size?

```
ax = sns.regplot(data=tips_df, x="total_bill", y="tip_per_person")
```



Less confident, less appearent. Keep in mind that it will always see a relations, the question is how confident!

CHAPTER

TWENTYTHREE

HEATMAP PLOT

A heatmap also deals with 2 dimensional data and cares about the relation. Here instead of numerical data with dots, we are using categorical data where every combination of the 2 categories has a singular value.

This results into a matrix that we visualize where each index of the matrix has its own color based on a color gradient. This plot got its name as it is used to find 'hot spots' between combinations of 2 categorical features.

```
import pandas as pd
import seaborn as sns
sns.set_theme()
sns.set(rc={'figure.figsize':(16,12)})
```

To make optimal use of this plot, we are going to take on a rather complex dataset, where we have measurements of brain networks. The idea is that we have several networks with several nodes in 2 hemispheres, the content of the data is not as important here, what matters is that we want to find correlations between different nodes in the brain.

```
brain_df = sns.load_dataset("brain_networks", header=[0, 1, 2], index_col=0)
brain_df.head()
```

```
network          1                     2                     3             \
node             1                     1                     1
hemi            lh         rh         lh         rh         lh         rh
0        56.055744  92.031036   3.391576  38.659683  26.203819 -49.715569
1        55.547253  43.690075 -65.495987 -13.974523 -28.274963 -39.050129
2        60.997768  63.438793 -51.108582 -13.561346 -18.842947  -1.214659
3        18.514868  12.657158 -34.576603 -32.665958  -7.420454  17.119448
4        -2.527392 -63.104668 -13.814151 -15.837989 -45.216927   3.483550

network          4                     5             ...         16   \
node             1                     1             ...          3
hemi            lh         rh         lh         rh  ...         rh
0        47.461037  26.746613 -35.898861  -1.889181  ...   0.607904
1        -1.210660 -19.012897  19.568010  15.902983  ...  57.495071
2       -65.575806 -85.777428  19.247454  37.209419  ...  28.317369
3       -41.800869 -58.610184  32.896915  11.199619  ...  71.439629
4       -62.613335 -49.076508  18.396759   3.219077  ...  95.597565

network                                17                               \
node             4                      1                     2
hemi            lh          rh         lh         rh         lh         rh
0       -70.270546   77.365776 -21.734550   1.028253   7.791784  68.903725
1       -76.393219  127.261360 -13.035799  46.381824 -15.752450  31.000332
2         9.063977   45.493263  26.033442  34.212200   1.326110 -22.580757
3        65.842979  -10.697547  55.297466   4.255006  -2.420144  12.098393
```

(continues on next page)

(continued from previous page)

```
4        50.960453  -23.197300  43.067562  52.219875  28.232882 -11.719750

network
node             3                       4
hemi            lh          rh          lh
0       -10.520872  120.490463 -39.686432
1       -39.607521   24.764011 -36.771008
2        12.985169  -75.027451   6.434262
3       -15.819172  -37.361431  -4.650954
4         5.453649    5.169828  87.809135

[5 rows x 62 columns]
```

luckily for us, the pandas library has an easy method of finding out what the correlation is between different columns of numerical data. These correlations are denoted between -1 (completely opposite) to 1 (completely related). Take a minute to understand how the columns and index changed using the operation, you can see that a node in a network and hemisphere has a correlation of 1.00 with itself.

```
brain_df.corr()
```

```
network                    1                   2                   3           \
node                       1                   1                   1
hemi                      lh        rh        lh        rh        lh        rh
network node hemi
1       1    lh    1.000000  0.881516 -0.042699 -0.074437 -0.342849 -0.169498
             rh    0.881516  1.000000  0.013073  0.033733 -0.351509 -0.162006
2       1    lh   -0.042699  0.013073  1.000000  0.813394 -0.006940 -0.039375
             rh   -0.074437  0.033733  0.813394  1.000000 -0.027324 -0.023608
3       1    lh   -0.342849 -0.351509 -0.006940 -0.027324  1.000000  0.553183
...                     ...       ...       ...       ...       ...       ...
17      2    lh   -0.206379 -0.273370 -0.151724 -0.224447  0.026579 -0.056687
             rh   -0.212601 -0.266456 -0.124508 -0.172704 -0.089109 -0.144020
        3    lh   -0.142770 -0.174222 -0.179912 -0.250455 -0.012675 -0.047434
             rh   -0.204326 -0.223572 -0.044706 -0.090798 -0.024644 -0.103875
        4    lh   -0.219283 -0.273626 -0.209557 -0.216674  0.013747 -0.058838

network                    4                   5            ...        16  \
node                       1                   1            ...         3
hemi                      lh        rh        lh        rh  ...        rh
network node hemi                                           ...
1       1    lh   -0.373050 -0.361726  0.431619  0.418708  ... -0.106642
             rh   -0.333244 -0.337476  0.431953  0.519916  ... -0.173530
2       1    lh   -0.019773  0.007099 -0.147374 -0.104164  ... -0.215429
             rh   -0.017577 -0.014632 -0.173501 -0.094717  ... -0.184458
3       1    lh    0.528787  0.503403 -0.157154 -0.185008  ... -0.146451
...                     ...       ...       ...       ...  ...       ...
17      2    lh    0.020064  0.084837 -0.359879 -0.394522  ...  0.173117
             rh    0.007278  0.029909 -0.299152 -0.295150  ...  0.299440
        3    lh    0.070114  0.100063 -0.245179 -0.303354  ... -0.055529
             rh    0.101791  0.128318 -0.302654 -0.277378  ...  0.079460
        4    lh   -0.069100 -0.031653 -0.282767 -0.279381  ...  0.418857

network                                      17                           \
node                       4                   1                   2
hemi                      lh        rh        lh        rh        lh        rh
network node hemi
```

(continues on next page)

(continued from previous page)

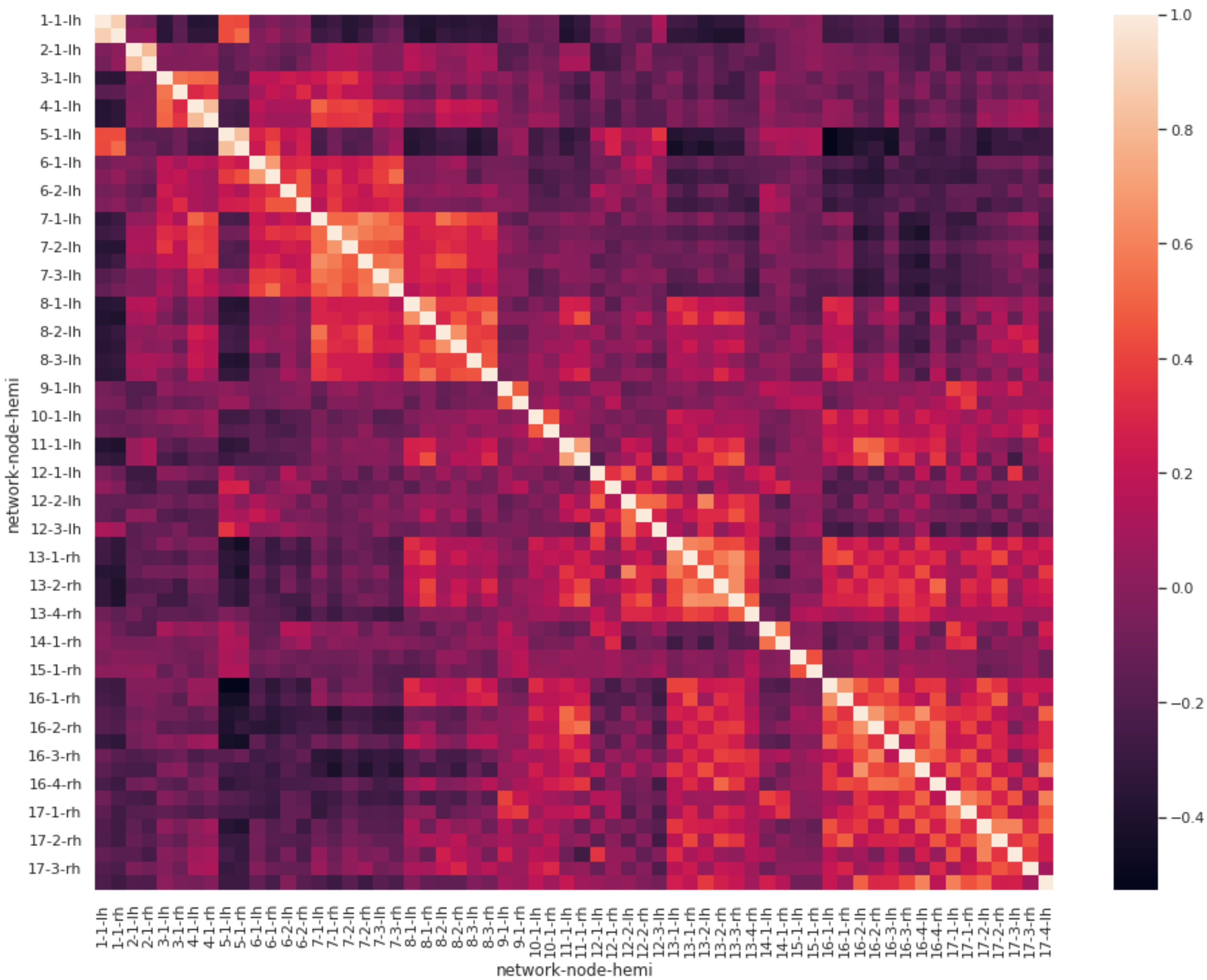
```
1       1    lh   -0.162254 -0.232501 -0.099781 -0.161649 -0.206379 -0.212601
             rh   -0.224436 -0.277954 -0.212964 -0.262915 -0.273370 -0.266456
2       1    lh   -0.239876 -0.093679 -0.240455 -0.190721 -0.151724 -0.124508
             rh   -0.244956 -0.061151 -0.255101 -0.169402 -0.224447 -0.172704
3       1    lh   -0.033931 -0.156972 -0.015964 -0.149944  0.026579 -0.089109
...                     ...       ...       ...       ...       ...       ...
17      2    lh    0.478606  0.258958  0.499351  0.319184  1.000000  0.597620
             rh    0.204444  0.453497  0.272868  0.440901  0.597620  1.000000
        3    lh    0.259191  0.046663  0.454838  0.188905  0.601382  0.345253
             rh    0.005291  0.296318  0.087061  0.224760  0.319382  0.456019
        4    lh    0.603491  0.172167  0.589364  0.451264  0.517481  0.256544

network
node                    3                   4
hemi                   lh        rh        lh
network node hemi
1       1    lh   -0.142770 -0.204326 -0.219283
             rh   -0.174222 -0.223572 -0.273626
2       1    lh   -0.179912 -0.044706 -0.209557
             rh   -0.250455 -0.090798 -0.216674
3       1    lh   -0.012675 -0.024644  0.013747
...                     ...       ...       ...
17      2    lh    0.601382  0.319382  0.517481
             rh    0.345253  0.456019  0.256544
        3    lh    1.000000  0.379705  0.264381
             rh    0.379705  1.000000  0.090302
        4    lh    0.264381  0.090302  1.000000

[62 rows x 62 columns]
```

This result is way to much to see a pattern, yet if we add a color scale and give each a gradation, we can see some correlations.

Can you see how nodes from the same network are related with a more whitish color? The heatmap might be fairly intimidating at first but is a powerful tool when handling bigger datasets.

```
sns.heatmap(data=brain_df.corr())
```

```
<AxesSubplot:xlabel='network-node-hemi', ylabel='network-node-hemi'>
```



Without going into the medical details we can also apply some machine learning to it and create a clustermap. This map is a way to group nodes from similar networks into clusters, an advances technique!

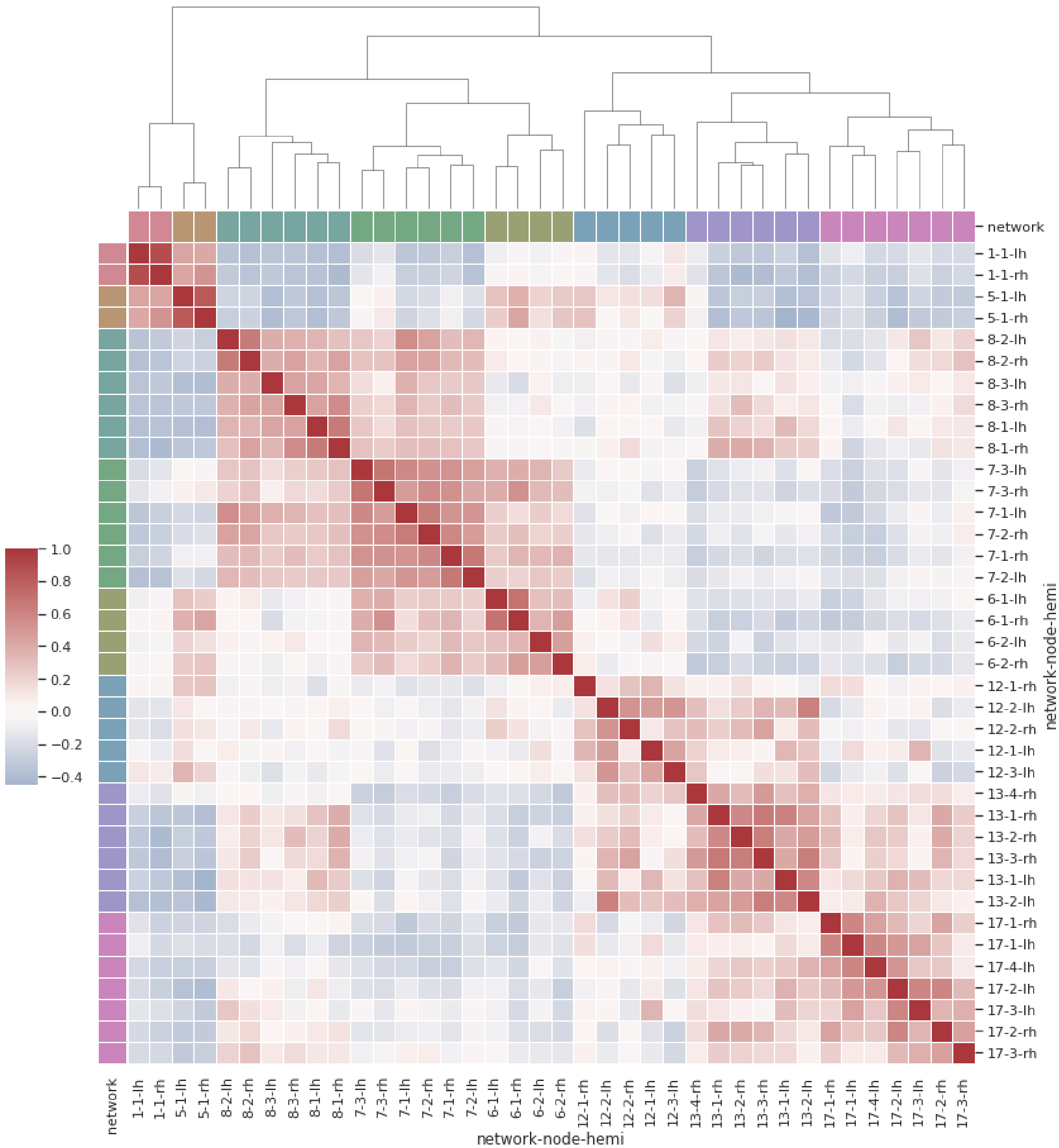
Gaze over the colors and look at the axi, notice how the computer figured out how to group the most similar nodes from networks. Also, I did not create this by myself, so don't give me credit for this!

```
# Select a subset of the networks
used_networks = [1, 5, 6, 7, 8, 12, 13, 17]
used_columns = (brain_df.columns.get_level_values("network")
                          .astype(int)
                          .isin(used_networks))
brain_df = brain_df.loc[:, used_columns]

# Create a categorical palette to identify the networks
network_pal = sns.husl_palette(8, s=.45)
network_lut = dict(zip(map(str, used_networks), network_pal))

# Convert the palette to vectors that will be drawn on the side of the matrix
networks = brain_df.columns.get_level_values("network")
network_colors = pd.Series(networks, index=brain_df.columns).map(network_lut)

# Draw the full plot
g = sns.clustermap(brain_df.corr(), center=0, cmap="vlag",
```

(continues on next page)

(continued from previous page)

```
                    row_colors=network_colors, col_colors=network_colors,
                    dendrogram_ratio=(.1, .2),
                    cbar_pos=(.02, .32, .03, .2),
                    linewidths=.75, figsize=(12, 13))

g.ax_row_dendrogram.remove()
```



Part V

5. Data Exploration

CHAPTER TWENTYFOUR

INTRODUCTION

this is an introduction

CHAPTER

TWENTYFIVE

VARIABLE IDENTIFICATION

in this notebook we are going to look into a few simple but interesting techniques about getting to know more about what is inside the dataset you are given. Whenever you start out on a new project these steps are usually the first that are performed in order to know how to proceed.

We start out by loading the titanic dataset from seaborn

```
import seaborn as sns
titanic_df = sns.load_dataset('titanic')
sns.set_theme()
sns.set(rc={'figure.figsize':(16,12)})
```

25.1 description

Let us start out simple and retrieve information about each column, using the .info method we can get non-null counts (giving us an idea if there are nans) and the type of each column (to see if we need to change types).

```
titanic_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

it looks like all types are already correctlyaddressed, but we can see a lot of nans are present for age and deck, this might be a problem!

For numerical columns we can get a bunch of information using the .describe method. this can also be used for categories but has less info

```
titanic_df.describe()
```

```
         survived      pclass         age       sibsp       parch        fare
count  891.000000  891.000000  714.000000  891.000000  891.000000  891.000000
mean     0.383838    2.308642   29.699118    0.523008    0.381594   32.204208
std      0.486592    0.836071   14.526497    1.102743    0.806057   49.693429
min      0.000000    1.000000    0.420000    0.000000    0.000000    0.000000
25%      0.000000    2.000000   20.125000    0.000000    0.000000    7.910400
50%      0.000000    3.000000   28.000000    0.000000    0.000000   14.454200
75%      1.000000    3.000000   38.000000    1.000000    0.000000   31.000000
max      1.000000    3.000000   80.000000    8.000000    6.000000  512.329200
```

```
titanic_df.describe(include=['category', 'object'])
```

```
         sex embarked  class  who deck  embark_town alive
count    891      889    891  891  203          889   891
unique     2        3      3    3    7            3     2
top     male        S  Third  man    C  Southampton    no
freq     577      644    491  537   59          644   549
```

25.2 Uniques, frequencies and ranges

the describe method is a bit lacklusting for categorical features, so we use some good old data wrangling to get more info, asking for unique values gives us all the possible values for a column. Aside from the uniques, we can also get the value counts or frequencies and the range of a column.

```
titanic_df['embark_town'].unique()
```

```
array(['Southampton', 'Cherbourg', 'Queenstown', nan], dtype=object)
```

```
titanic_df['embark_town'].value_counts()
```

```
Southampton    644
Cherbourg      168
Queenstown      77
Name: embark_town, dtype: int64
```

```
titanic_df['age'].min(), titanic_df['age'].max()
```

```
(0.42, 80.0)
```

25.3 mean and deviation

to get more information about a numerical range, we calculate the mean and deviation. Note that these statistics imply that our column is normally distributed!

You can also see that I applied the dropna method, this because the calculations cannot handle nan values, but this means our outcome might be distorted from the truth, thread carefuly.

```
import statistics
```

```
titanic_df['age'].dropna().mean()
```

```
29.69911764705882
```

```
titanic_df['age'].dropna().median()
```

```
28.0
```

25.4 median and interquantile range

When our distribution is not normal, using the median and IQR is advised. First we apply the shapiro wilk test and it has a very low p-value (the second value) which means we can reject the null-hypothesis that there is a normal distribution. more info about shapiro-wilk can be found on wikipedia

```
from scipy.stats import shapiro
shapiro(titanic_df['age'].dropna())
```

```
ShapiroResult(statistic=0.9814548492431641, pvalue=7.322165629375377e-08)
```

```
titanic_df['age'].dropna().median()
```

```
28.0
```

```
from scipy.stats import iqr
iqr(titanic_df['age'].dropna())
```

```
17.875
```

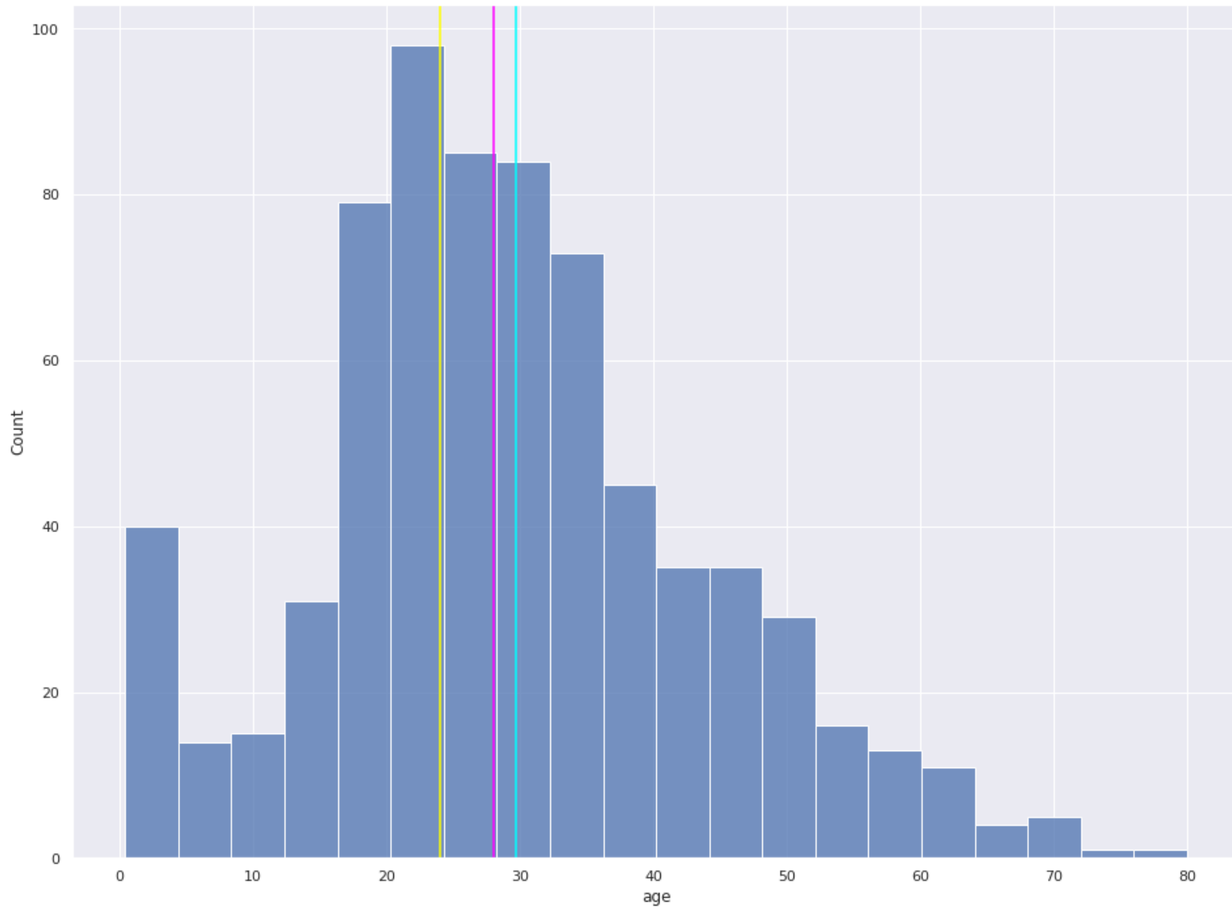
```
from scipy.stats.mstats import mquantiles
mquantiles(titanic_df['age'].dropna())
```

```
array([20., 28., 38.])
```

Appearently the average of 29.70 is fairly higher than the median at 28, meaning that there is a shift towards older people. You can also see this on the following plot, where we note the mean, median and mode.

```
ax = sns.histplot(data=titanic_df, x='age')

ax.axvline(titanic_df.age.mean(), color='cyan')
ax.axvline(titanic_df.age.median(), color='magenta')
ax.axvline(titanic_df.age.mode()[0], color='yellow')
```

```
<matplotlib.lines.Line2D at 0x7fa1b3657e50>
```



25.5 modes and frequencies

When we don't have numerical data we can still find some interesting results, here we use the mode (most frequent value) and the proporties of each value to deduce the proporties of people that embarked in the 3 different towns. Nearly 3/4 people embarked in one harbour.

```
titanic_df['embark_town'].mode()
```

```
0    Southampton
dtype: object
```

```
titanic_df['embark_town'].value_counts()/len(titanic_df)
```

```
Southampton    0.722783
Cherbourg      0.188552
Queenstown     0.086420
Name: embark_town, dtype: float64
```

CHAPTER

TWENTYSIX

UNI-VARIATE ANALYSIS

In this notebook we will go a bit deeper into the analysis of a single column or variable of our dataset. This means we will be looking into how visualisations might be useful to attain more information. We start out again by loading the titanic dataset and obtaining the same info as before.

```
import seaborn as sns
titanic_df = sns.load_dataset('titanic')
sns.set_style()
sns.set(rc={'figure.figsize':(16,12)})
```

```
titanic_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype   
---  ------       --------------  -----   
 0   survived     891 non-null    int64   
 1   pclass       891 non-null    int64   
 2   sex          891 non-null    object  
 3   age          714 non-null    float64 
 4   sibsp        891 non-null    int64   
 5   parch        891 non-null    int64   
 6   fare         891 non-null    float64 
 7   embarked     889 non-null    object  
 8   class        891 non-null    category
 9   who          891 non-null    object  
 10  adult_male   891 non-null    bool    
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object  
 13  alive        891 non-null    object  
 14  alone        891 non-null    bool    
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

26.1 Nominal data

Lets take a look into a norminal column, the embark town has 3 different options and we already saw how to count the values and calculate proportions.

```
titanic_df['embark_town'].value_counts()
```

```
Southampton    644
Cherbourg      168
Queenstown      77
Name: embark_town, dtype: int64
```

```
titanic_df['embark_town'].value_counts()/len(titanic_df)
```

```
Southampton    0.722783
Cherbourg      0.188552
Queenstown     0.086420
Name: embark_town, dtype: float64
```

```
import statistics
statistics.mode(titanic_df['embark_town'])
```

```
'Southampton'
```

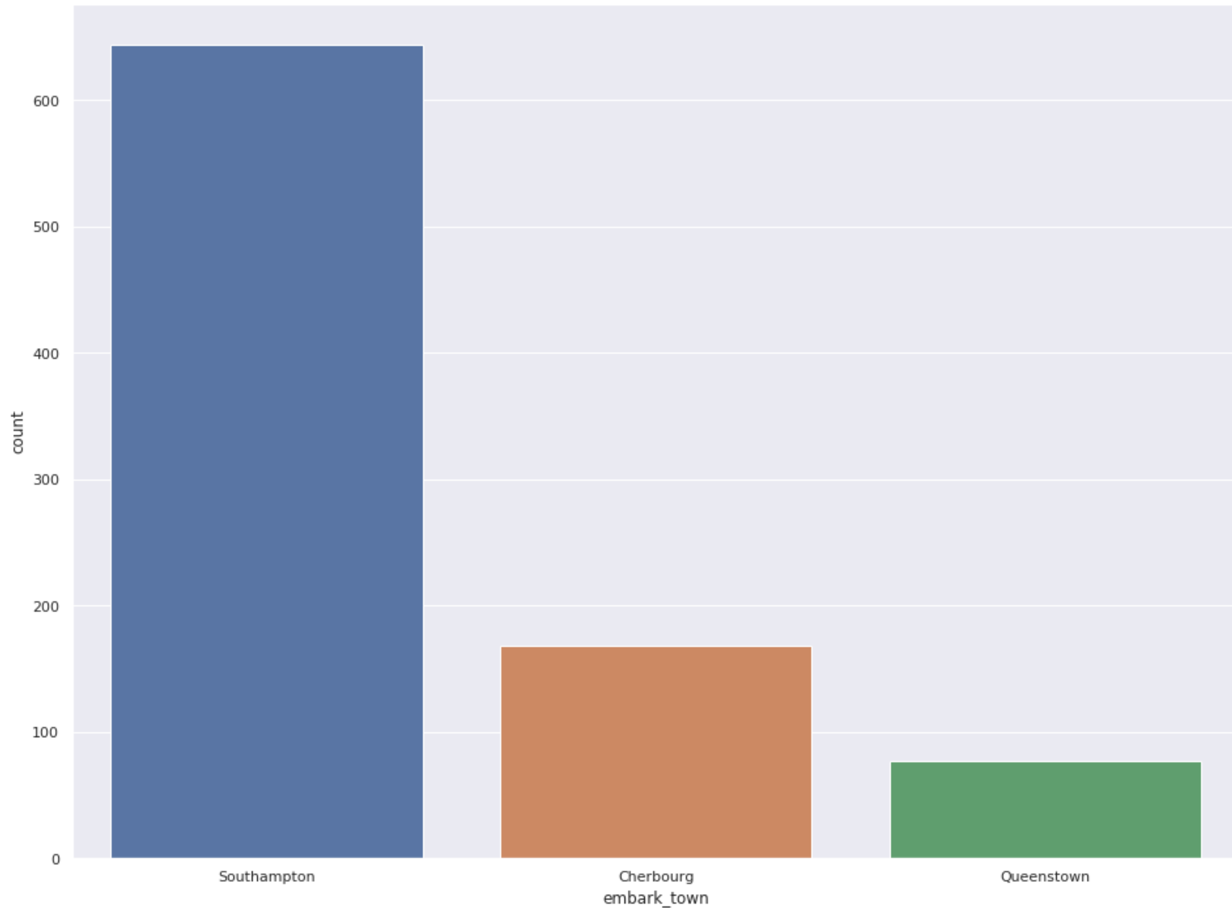
```
statistics.median(titanic_df['embark_town'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_11541/1735617235.py in <module>
----> 1 statistics.median(titanic_df['embark_town'])

/usr/lib/python3.8/statistics.py in median(data)
    425     
    426     """
--> 427     data = sorted(data)
    428     n = len(data)
    429     if n == 0:

TypeError: '<' not supported between instances of 'float' and 'str'
```

Hmmm, it seems we can not take the median because python does not know the order of the categories. Let's kick it up a notch and use some plots to make these proportions more clear, we'll use a bar chart to do this.

```
sns.countplot(data=titanic_df, x='embark_town')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7f61d100>
```

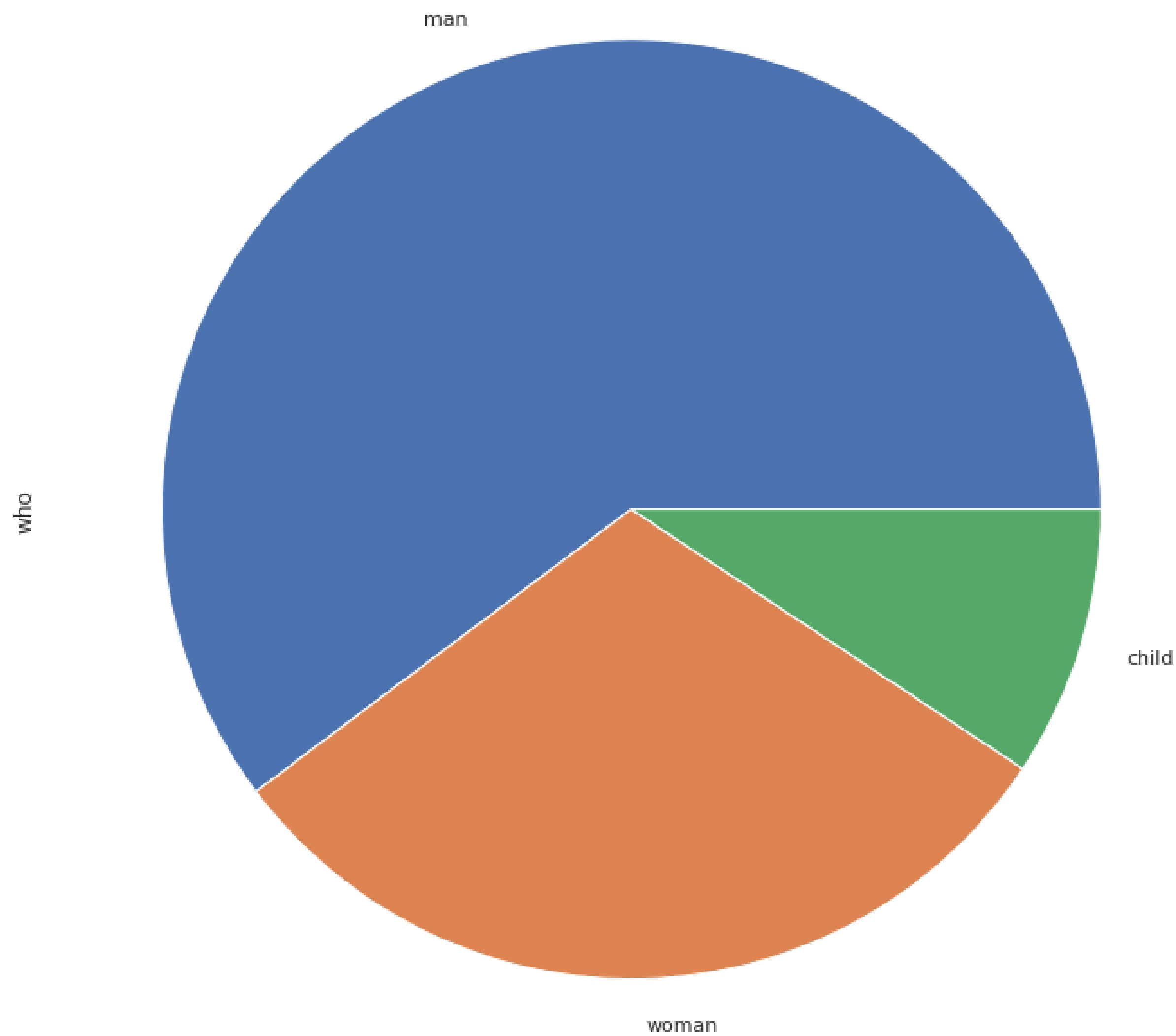


Something important that I would like to mention here is that this serves as a method to validate sample size, if e.g. only a handful persons would embark on a location, the statistics in this group will have a high variance which will not always shot in your visualisations. Be mindful to check sample sizes of categories when applying statistics.

The bar chart is ideal to compare the values to eachother, yet if we would like to visualise the proportions to eachother, we need a pie plot. Here we use the 'who' feature containing information about the person itself, we have 3 categories: man, woman, child.

```
titanic_df.who.value_counts().plot.pie()
```

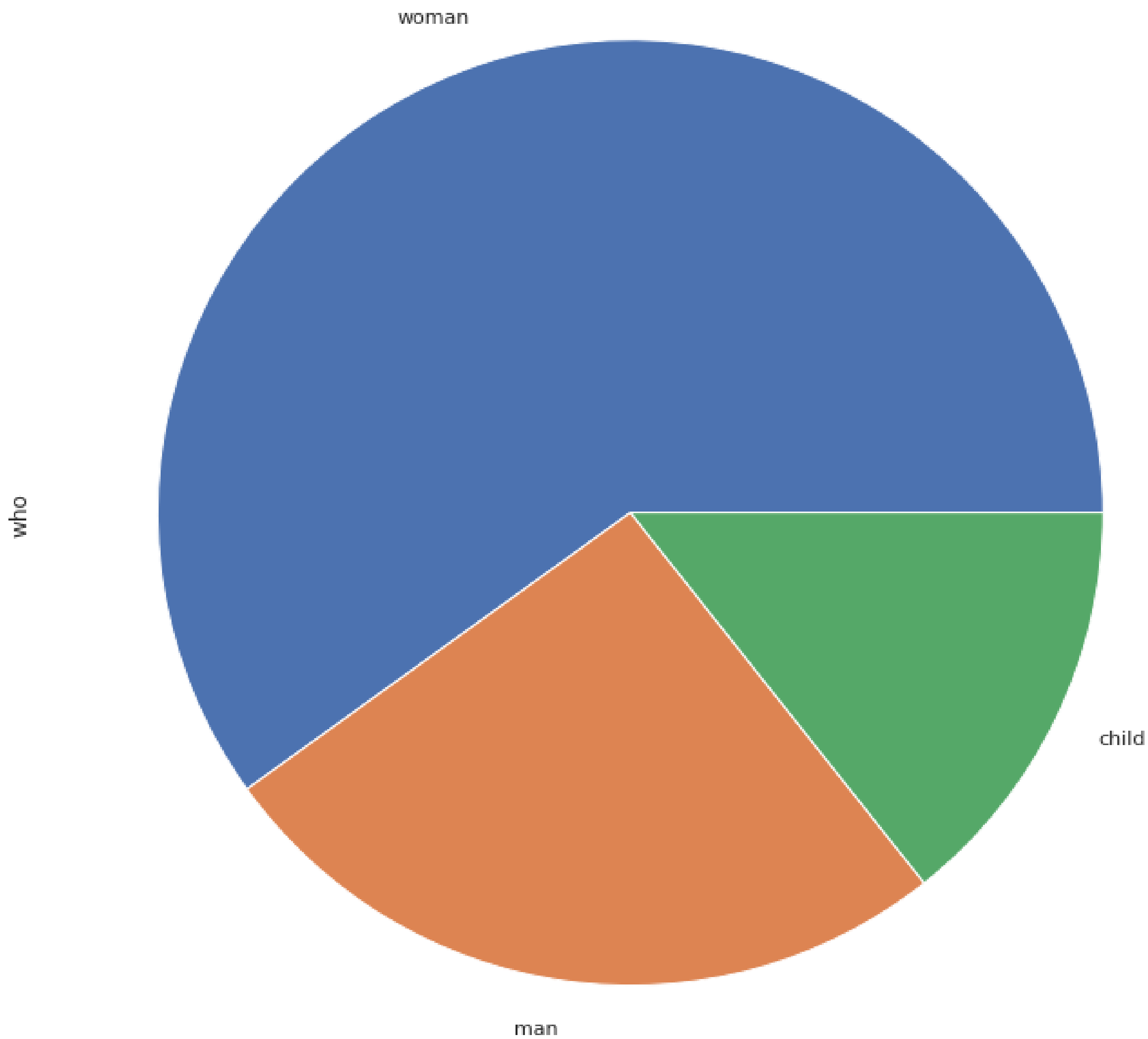
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7d2ca430>
```



The saying goes 'Woman and children first' which would mean the survivors are mainly those 2 groups, let us confirm that by subselecting only the survivors and recreate the pie plot.

```
titanic_df[titanic_df.survived==1].who.value_counts().plot.pie()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7d285dc0>
```



You can see that the groups are now reversed, where men are proportionally less represented. By using a pie plot we circumvent the problem where we have a bias towards size of our dataset, the pie plot applies scaling by itself.

26.2 Ordinal data

Whilst there was no order in the town where passengers embarked, there is in the class of the ticket they bought. So we need to keep this in mind when exploring. We can not just say they belonged to any class as there is a difference in these classes! However the same statistics apply, but with a different twist.

```
titanic_df['class'].value_counts()
```

```
Third     491
First     216
```

(continues on next page)

(continued from previous page)

```
Second    184
Name: class, dtype: int64
```

```
titanic_df['class'].value_counts()/len(titanic_df)
```

```
Third     0.551066
First     0.242424
Second    0.206510
Name: class, dtype: float64
```

it seems more people travelled on the titanic in first class than second class! nothing you would see nowadays.

```
statistics.mode(titanic_df['class'])
```

```
'Third'
```

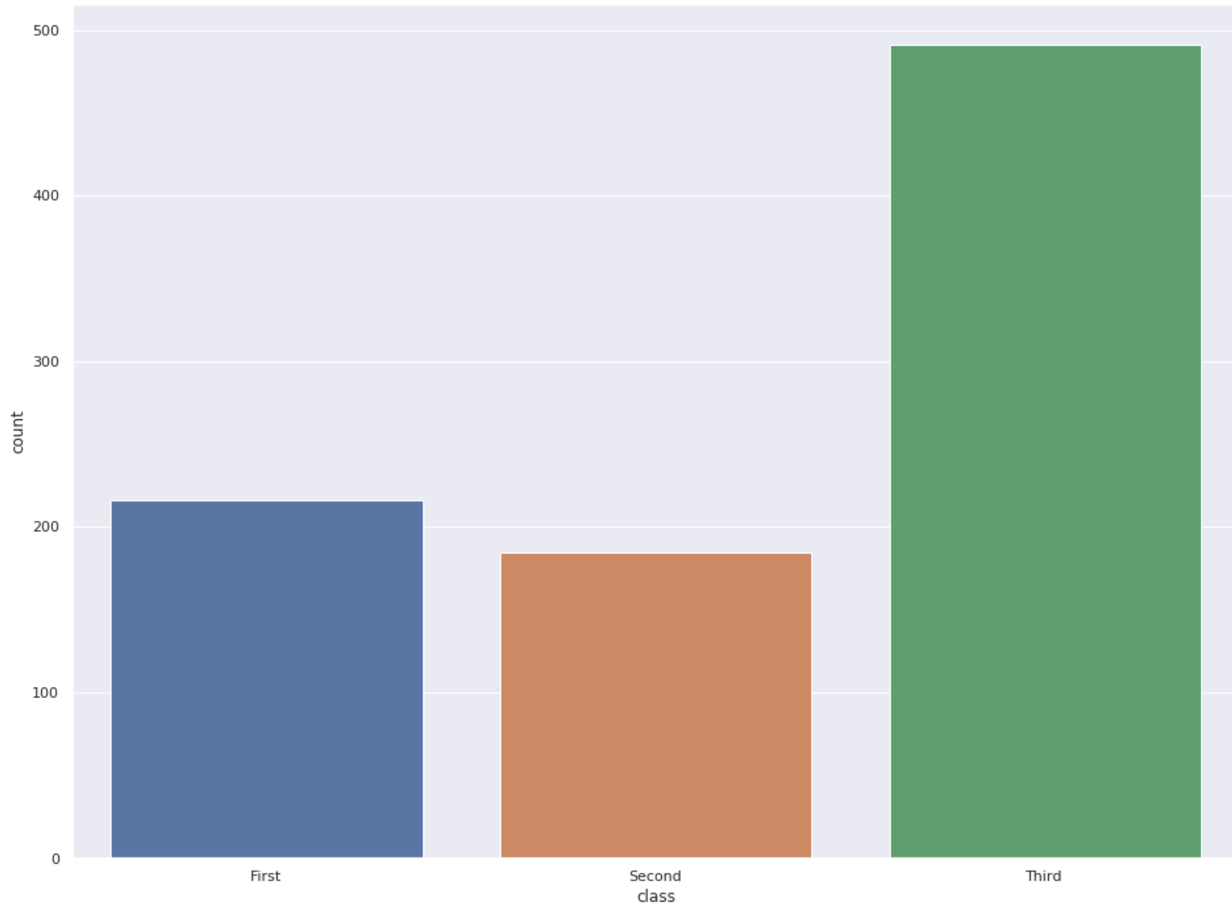
```
statistics.median(titanic_df['class'])
```

```
'Third'
```

Here we can use the median, as there is an order in the classes! By using a bar plot we can visualise the distribution, because the graphing library knows the order of the categories, they will also be properly displayed, how convenient.

```
sns.countplot(data=titanic_df, x='class')
```

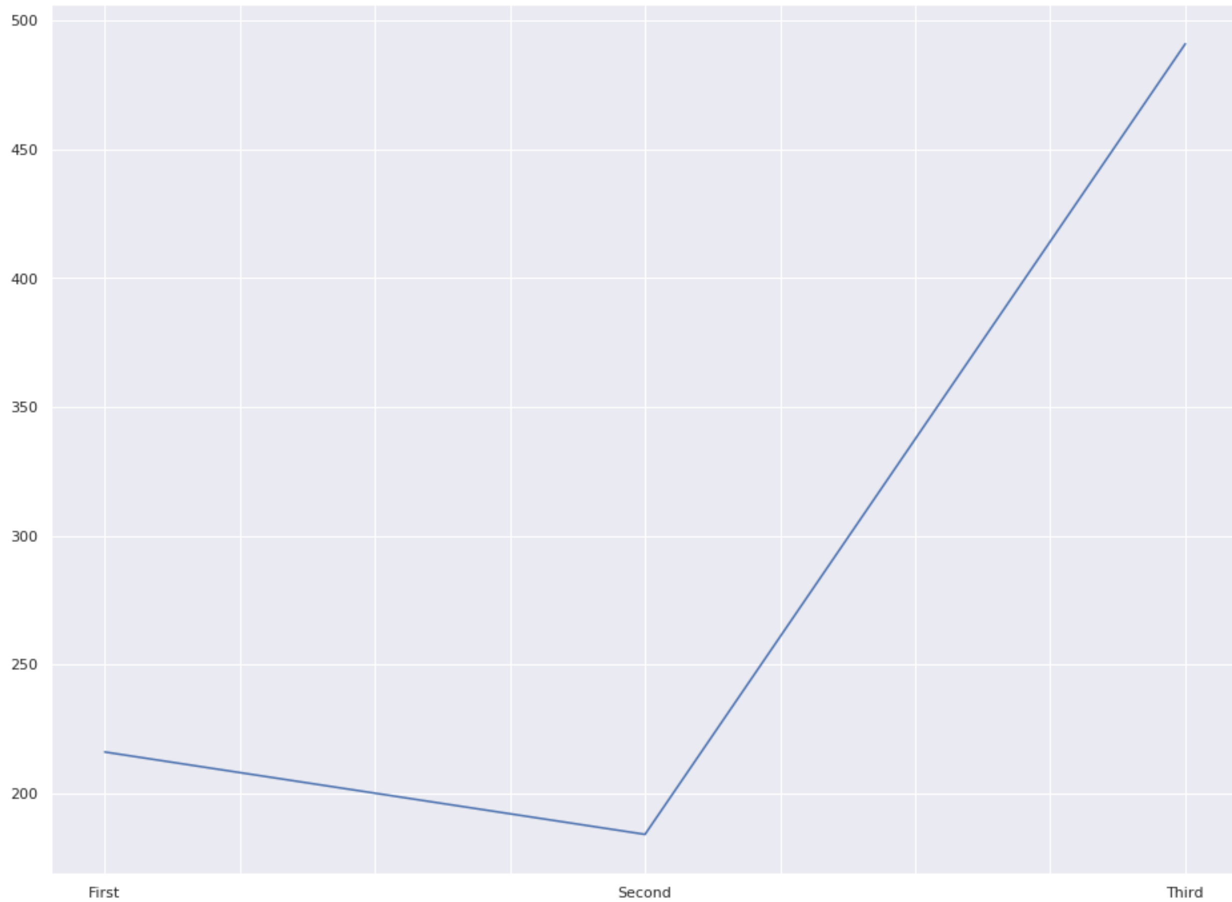
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7d250580>
```



You could however also create a line plot with this, as there is a relation between the classes, as shown below.

```
titanic_df['class'].value_counts()[['First', 'Second', 'Third']].plot()
```

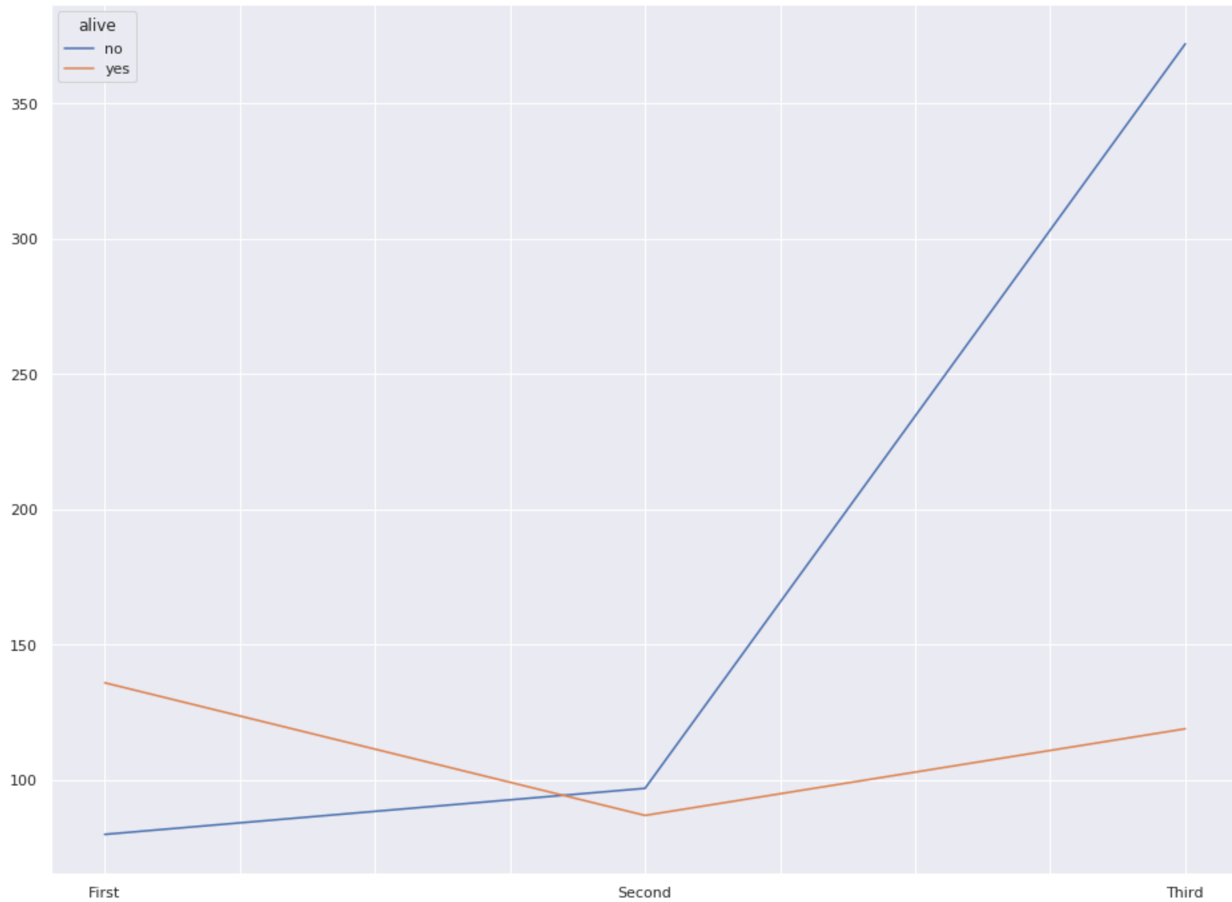
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7d224370>
```



This plot feels underwhelming with only 3 points, but we could make it more interesting, we divide our data on who survived and count the amount of persons per class that survived or not. It is clear to say the a higher class meant higher chances of survival.

```
titanic_df['class'].groupby(titanic_df.alive).apply(lambda x: x.value_counts()).
↪unstack(0).reindex(['First', 'Second', 'Third']).plot()
```

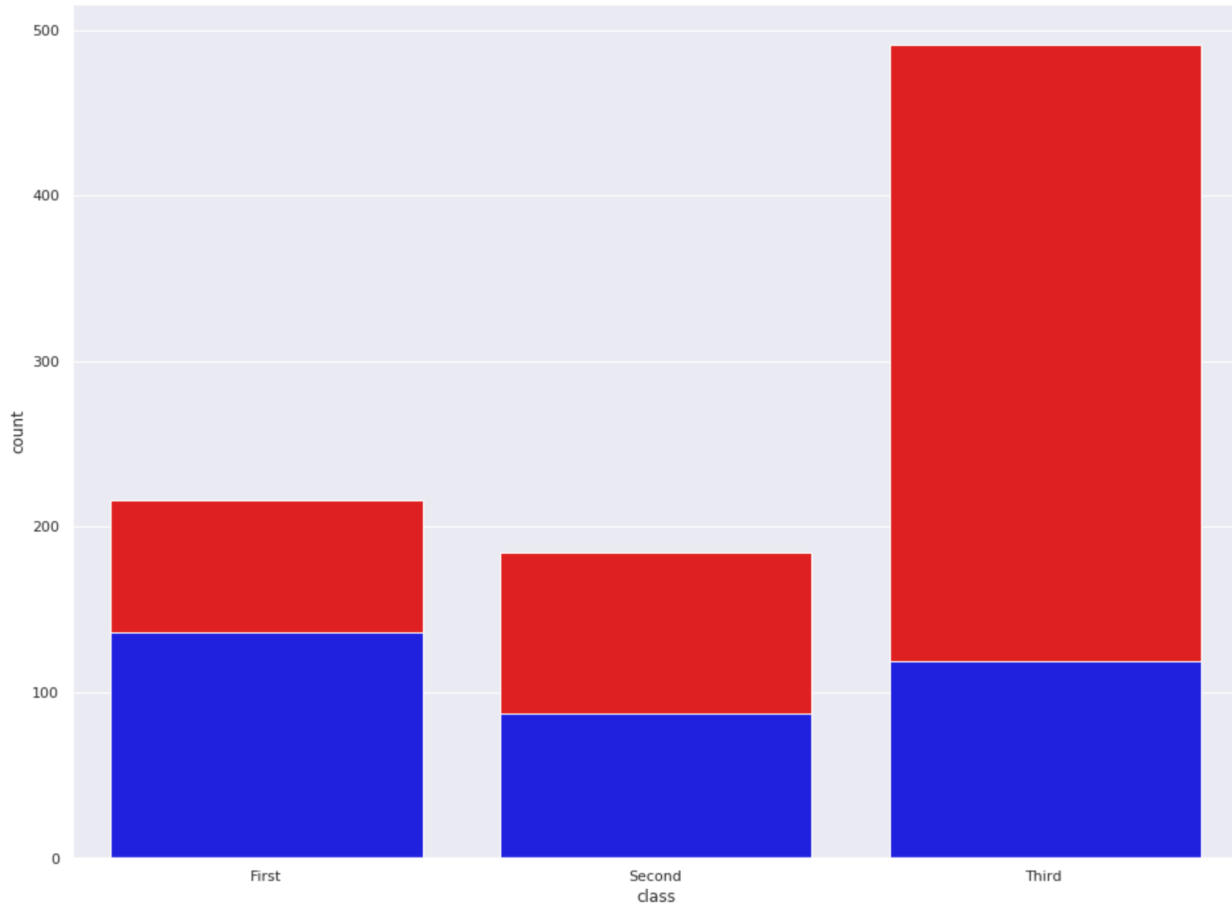
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7d1f9550>
```



Personally as there is no time related factor in our x-axis, the line or parallel plot here is not as convenient. Since we have a situation where there is a confinement that the amount of survived can not be more that the total, I would opt for a bar plot, which is show below.

```
sns.countplot(x = 'class', data = titanic_df, color = 'red')
sns.countplot(x = 'class', data = titanic_df[titanic_df.survived==1], color = 'blue')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7d278b50>
```



26.3 Continuous data

After categories which are discrete we also have continuous data, which is by nature always ordered. Here we can perform all the other statistical methods along with the mean, but again keep in mind that using the mean does come with a lot of responsibility.

```
statistics.mode(titanic_df['age'])
```

```
24.0
```

```
statistics.median(titanic_df['age'].dropna())
```

```
28.0
```

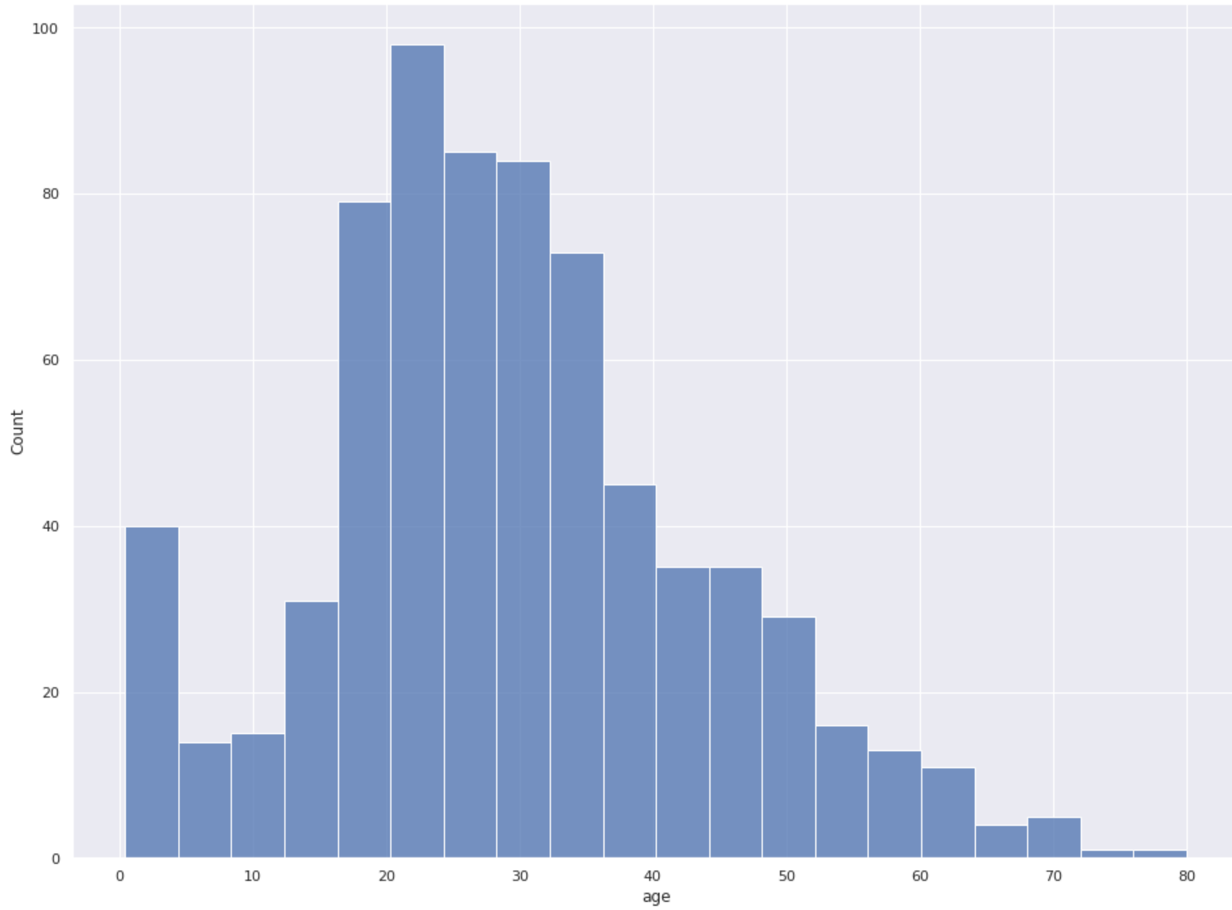
```
statistics.mean(titanic_df['age'].dropna())
```

```
29.699117647058824
```

A very potent method of showing the distribution is a histogram or distribution plot as shown below, here we can see the long tail on the right which we correctly predicted earlier when we saw that the mean was slightly higher than the median.

```
sns.histplot(titanic_df['age'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7ce39f10>
```



going into more mathematical calculations, we can calculate the interquartile ranges, the upper and lower bounds and therefore find any outliers

```
q1, q3 = titanic_df['age'].quantile([0.25, 0.75])
q3-q1
```

```
17.875
```

```
lower_bound = q1 - (1.5 * q1)
upper_bound = q3 + (1.5 * q3)
lower_bound, upper_bound
```

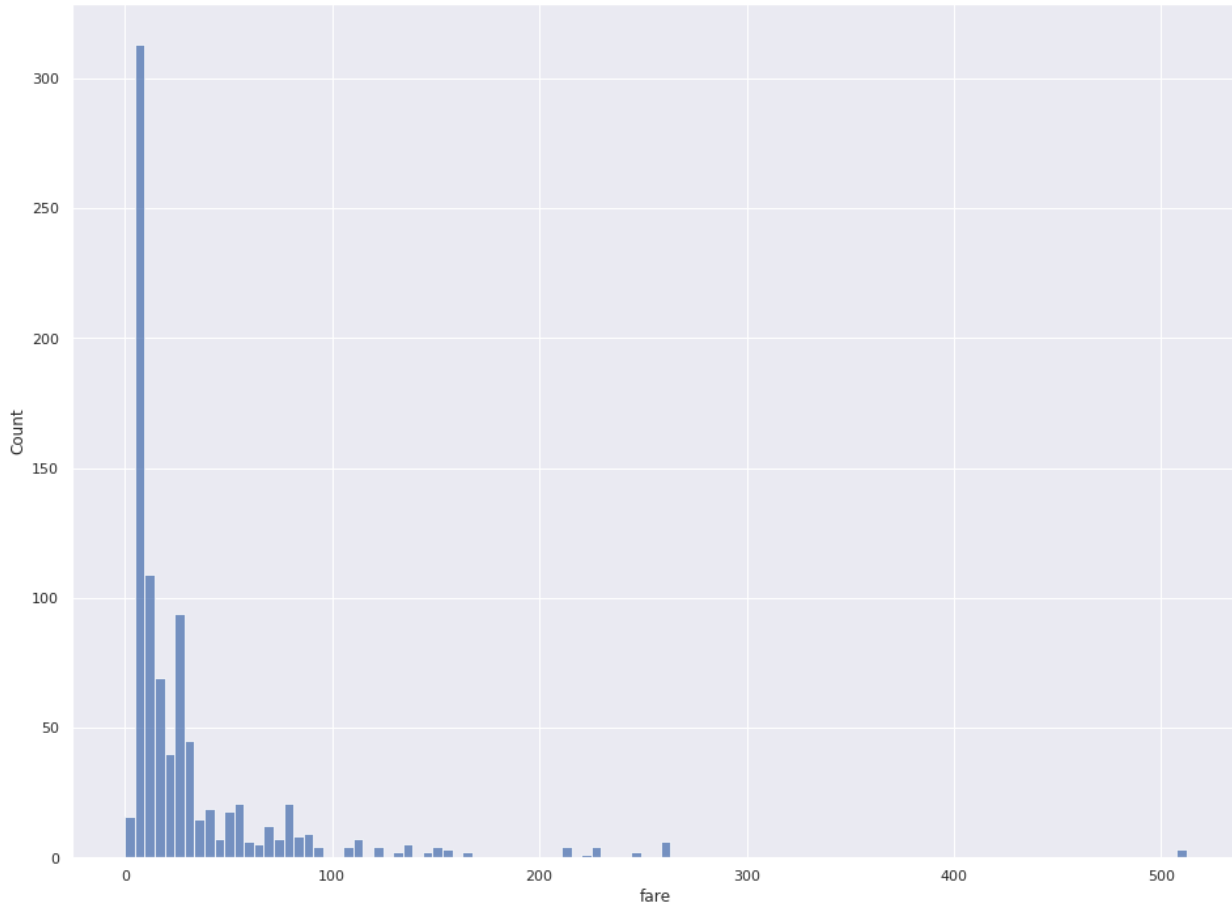
```
(-10.0625, 95.0)
```

It seems that for age, no outliers have been found, which is not really suprising as you don't have any control over your age, unfortunately…

Another numerical feature they had control over was the fare, we give a visualisation of the distrubition here.

```
sns.histplot(titanic_df['fare'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7cddbf10>
```



This distribution looks horrific, we could also look at the mean and median differences to see this tremendous shift towards higher fares.

```
print('median')
print(titanic_df.fare.median())
print('mean')
print(titanic_df.fare.mean())
```

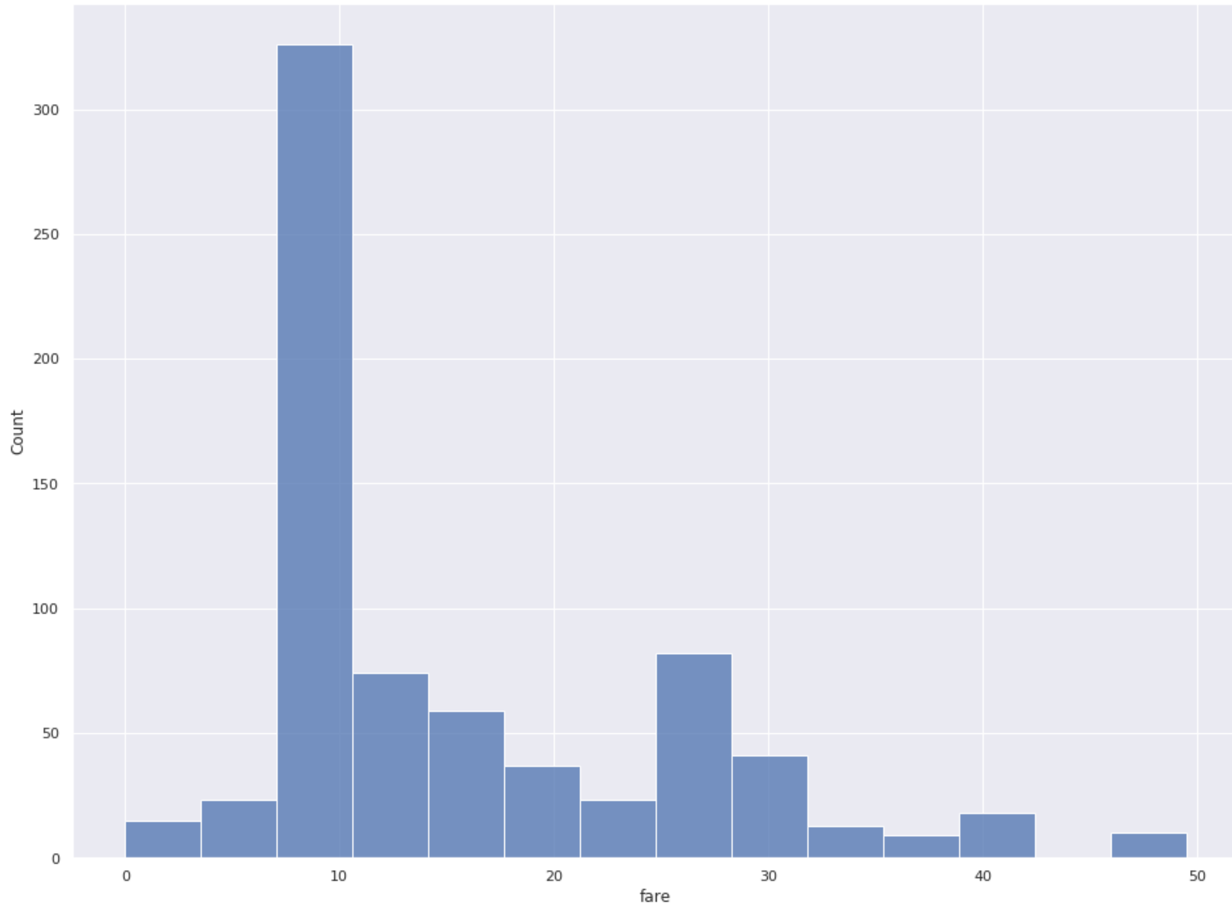
```
median
14.4542
mean
32.204207968574636
```

Perhaps you can use the outlier detection of above to find the upper outlier treshold?

Let us assume that the upper bound for fares is about 50, which is lower than some tickets. By removing these values we can correct our distribution and get a more evened out result. This is especially useful in cases of machine learning where we would not not our algorithm to be biased due to a few extraordinary values, we would have to seperate these specific cases to ensure higher accuracy.

```
sns.histplot(titanic_df[titanic_df.fare<50].fare)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f4e7cbf1220>
```



Much better, here we can clearly see our values, keep the records with outliers seperate for other purposes. Again looking at the new mean and median we see a lot less difference, indicating a better distribution.

```
print('median')
print(titanic_df[titanic_df.fare<50].fare.median())
print('mean')
print(titanic_df[titanic_df.fare<50].fare.mean())
```

```
median
11.1333
mean
15.500598493150687
```

CHAPTER

TWENTYSEVEN

BI-VARIATE ANALYSIS

In this notebook we are going to look at correlations between two columns in our dataset, this is were it becomes interesting as it opens more opportunities to explore our dataset. We start out by importing necessary libraries and loading the titanic dataset.

```
import seaborn as sns
import pandas as pd
from scipy import stats
titanic_df = sns.load_dataset('titanic')
sns.set_style()
sns.set(rc={'figure.figsize':(16,12)})
```

```
titanic_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    category
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    category
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), category(2), float64(2), int64(4), object(5)
memory usage: 80.7+ KB
```

27.1 Categorical vs categorical

The first comparison we can do is between 2 categorical variables, in this dataset we can use the class of the passenger and the town they embarked the titanic, let's make a contingency table first.

```
contingency_table = pd.crosstab(titanic_df['embark_town'], titanic_df['class'])
contingency_table
```

```
class        First  Second  Third
embark_town
Cherbourg       85      17     66
Queenstown       2       3     72
Southampton    127     164    353
```

With all these numbers it is fairly hard to find if there is a correlation between these 2 variables. Let statistics do the work and get the chi squared test involved, we do not apply a continuity correction as the embarkment is a nominal variable.

The results of the Cramer V test (simplified chi squared test).

```
chi, p, dof, exp = stats.chi2_contingency(contingency_table, correction=False)
chi, p, dof, exp
```

```
(123.75190952951289,
 8.435267819894384e-26,
 4,
 array([[ 40.44094488,  34.77165354,  92.78740157],
        [ 18.53543307,  15.93700787,  42.52755906],
        [155.02362205, 133.29133858, 355.68503937]]))
```

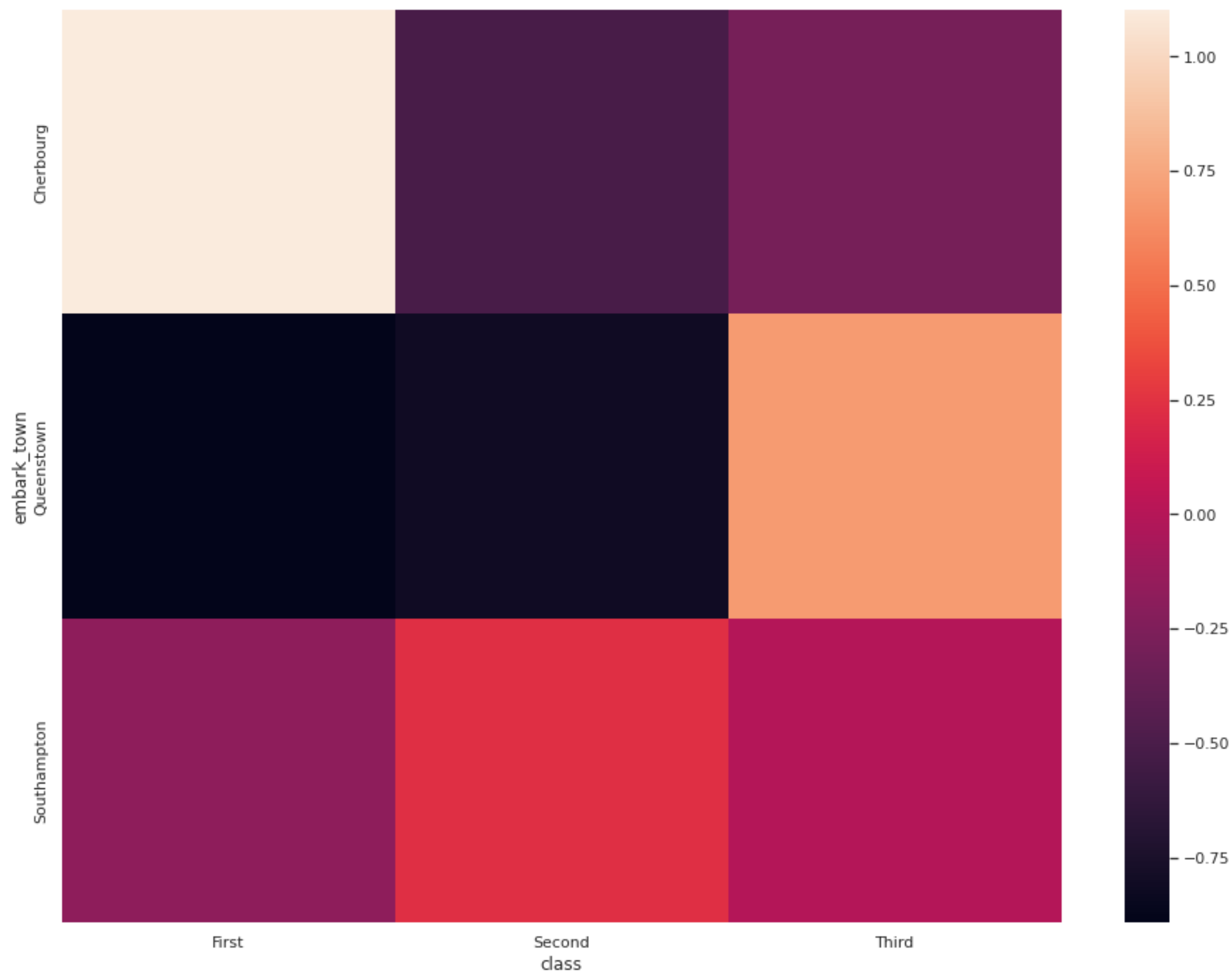
in order of appearance:

- the test statistic chi is very high, indicating a correlation
- the p value is low, so this is definitely not by chance
- there are 4 'degrees of freedom'
- the expected frequency table shows what it thinks the proporties should look like

What we could do now is create a heatmap with the contingency table but subtract the expected non-biased values and scale using the expected values (real - expected)/expected. This gives us the biggest changes in respect with 'random' values.

```
sns.heatmap(
    pd.DataFrame((contingency_table-exp)/exp, index=contingency_table.index,␣
 ↪columns=contingency_table.columns)
)
```

```
<AxesSubplot:xlabel='class', ylabel='embark_town'>
```



There seems to be much more people from first class that have embarked in Cherbourg, and the lower classes are more represented from Queenstown. The population from southampton only sees a positive deviation in second class.

To demonstrate that there can also be no correlation we now calculate the proportions of survival for each town and class combination.

```
survived_df = titanic_df.groupby(['embark_town', 'class']).survived.sum().unstack(
↪'class')/contingency_table
survived_df
```

```
class            First    Second     Third
embark_town
Cherbourg     0.694118  0.529412  0.378788
Queenstown    0.500000  0.666667  0.375000
Southampton   0.582677  0.463415  0.189802
```

If we would do a Cramer V test now, we assume there would be no significance, as it would not make sense that the embarked town has no influence on the chances (proportion of survived persons) of survival.

```
chi, p, dof, exp = stats.chi2_contingency(survived_df, correction=True)
p
```

```
0.9989353452702686
```

As you can see, the p value is 0.99, indicating that the differences in embarkment are purely coincidental!

27.2 Categorical vs continuous

The most interesting exploration (in my opinion) happens when we combine categorical and continuous data, as more graphing opportunities are present. When doing this comparison, we usually use the student t-test or Z-test, you can spend hours arguing the difference and which to use, yet I will stick for simplicity with the t-test for robuustness.

we can use the t-test to check if a continuous variable changes between 2 categories of a categorical variable.

let us seperate the men from the women and see if they had to pay a different fare amount

```
t, p = stats.ttest_ind(
    titanic_df.fare[titanic_df.who=='man'],
    titanic_df.fare[titanic_df.who=='woman']
)
t, p
```

```
(-5.817465335062089, 8.614583735152227e-09)
```

Our p-value again is very low, indicating there is a difference in the groups. The t statistic is -5.82, meaning that the second group (women) are paying more for fares.

We print out the means to verify

```
print('mean male fare')
print(titanic_df.fare[titanic_df.who=='man'].mean())
print('mean female fare')
print(titanic_df.fare[titanic_df.who=='woman'].mean())
```

```
mean male fare
24.864181750465548
mean female fare
46.570711070110704
```

By the looks of this, the fares are heavily gender biased. To put this into more detail, we pivot the means of each group including class into a table, as female might be more in the upper classes.

```
titanic_df.groupby(['who', 'class']).fare.mean().unstack('class')
```

```
class        First     Second      Third
who
child   139.382633  28.323905  23.220190
man      65.951086  19.054124  11.340213
woman   104.317995  20.868624  15.354351
```

This already makes more sense, it is mainly the first class difference that drives up the prices, yet the difference seems to be still present.

Can you perform a t-test on the gender fare gap in the third class, is it still significant?

A t-test is ideal if you would like to compare 2 groups, yet often we have multiple groups. For this we can use a (one_way) ANOVA or ANalysis Of VAriance.

We seperate on class and check if the fare is significantly different.

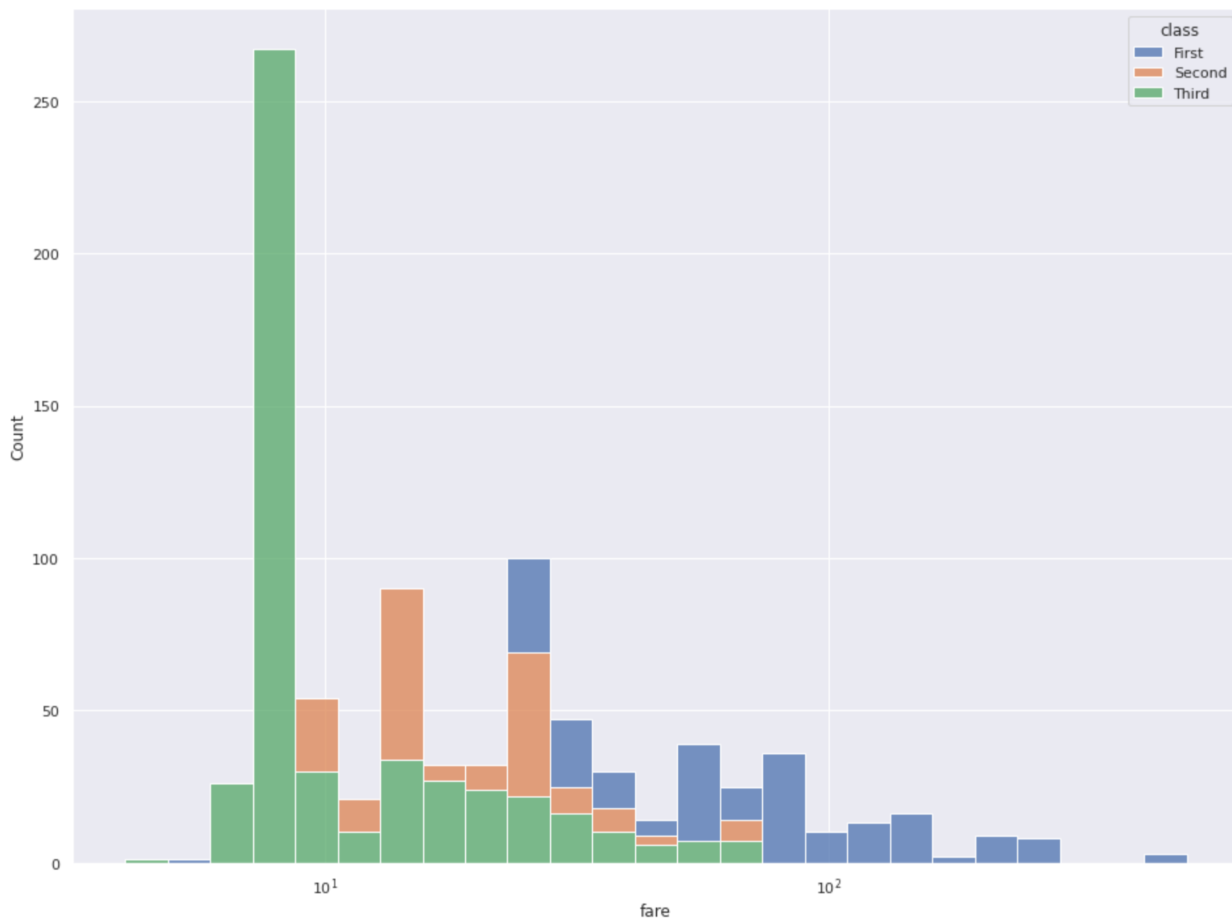
```
F, p = stats.f_oneway(
    titanic_df.fare[titanic_df.pclass==1],
    titanic_df.fare[titanic_df.pclass==2],
    titanic_df.fare[titanic_df.pclass==3]
)
F, p
```

```
(242.34415651744814, 1.0313763209141171e-84)
```

This was more or less a no-brainer, as it is advertised that higher classes come with a higher pricetag. We can use a nice histogram to show this division of class.

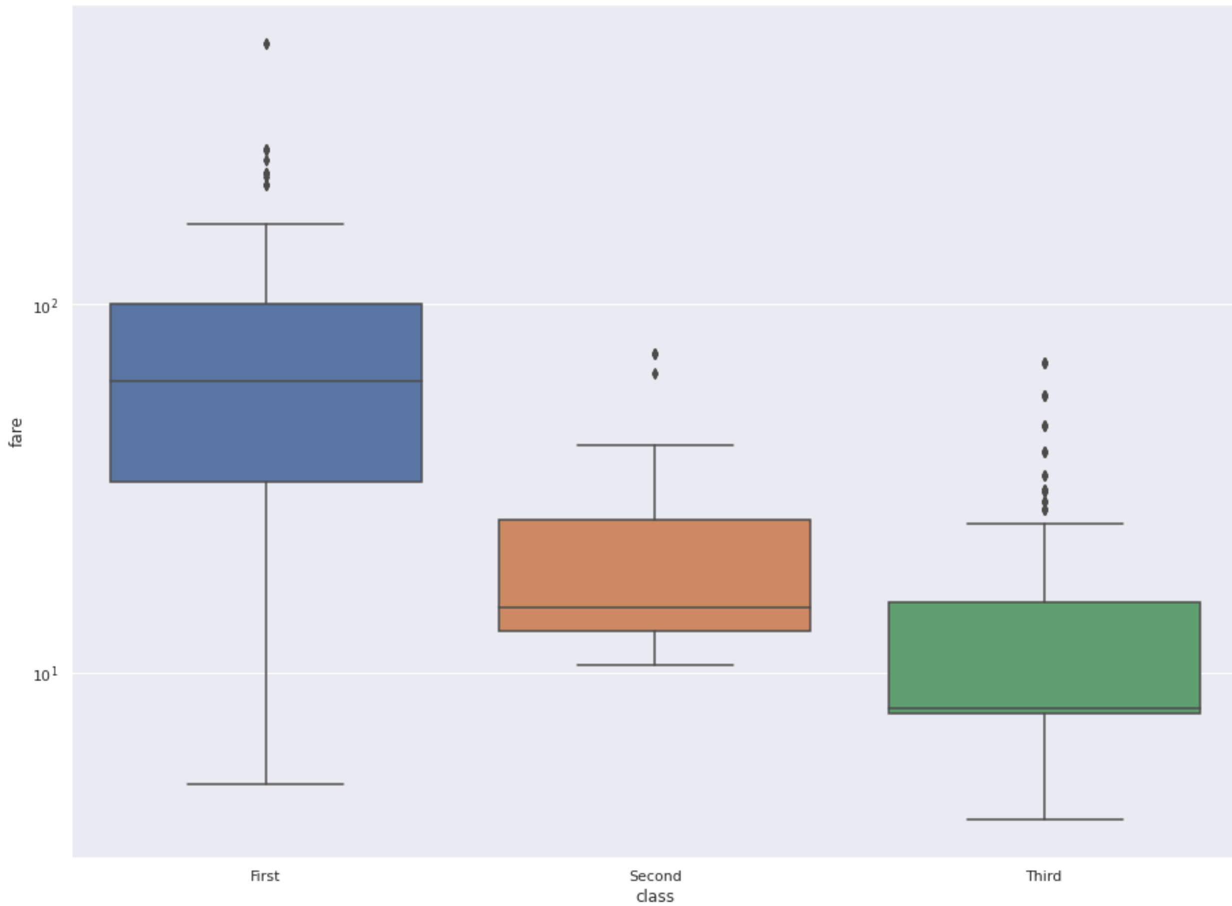
```
sns.histplot(data=titanic_df[titanic_df.fare!=0], x='fare', hue='class', log_
↪scale=True, multiple='stack', bins=25)
```

```
<AxesSubplot:xlabel='fare', ylabel='Count'>
```



A less cluttered plot would be to use a boxplot, containing less information about the distribution, yet still showing simple statistics.

```
ax = sns.boxplot(data=titanic_df[titanic_df.fare!=0], x='class', y='fare')
ax.set_yscale("log")
```



We could do something similar, but taking the age instead of the fare, giving us the following result.

```
F, p = stats.f_oneway(
    titanic_df.age[titanic_df.pclass==1].dropna(),
    titanic_df.age[titanic_df.pclass==2].dropna(),
    titanic_df.age[titanic_df.pclass==3].dropna()
)
F, p
```

```
(57.443484340676214, 7.487984171959904e-24)
```

The p value indicates there is surely a difference in age between classes, how about we look at the means for each class.

```
titanic_df.groupby('pclass').age.mean()
```

```
pclass
1    38.233441
2    29.877630
3    25.140620
Name: age, dtype: float64
```

What about any statistical significant differences in ages for the groups that survived and didn't, could you perform this analysis? Report your findings in a histogram.

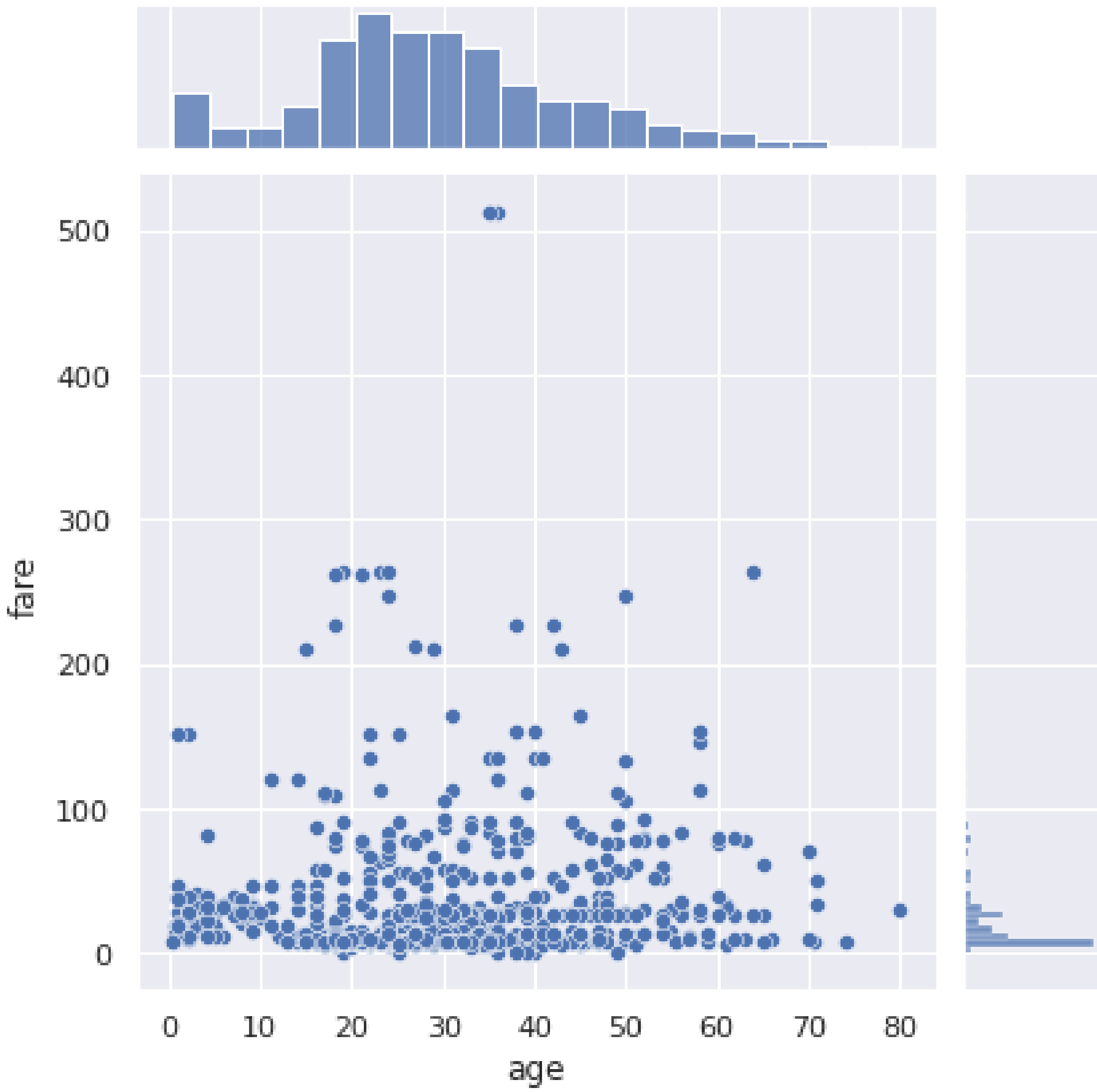
27.3 Continuous vs continuous

A thirds option to explore the interactions within your dataset is by comparing 2 continuous variables.

Seaborn has a nice functionality where can perform a jointplot that not only shows us the scatter plot but also the distributions, When we perform this plot we notice the inbalanced distribution of the fares.

```
sns.jointplot(data=titanic_df, x='age', y='fare')
```

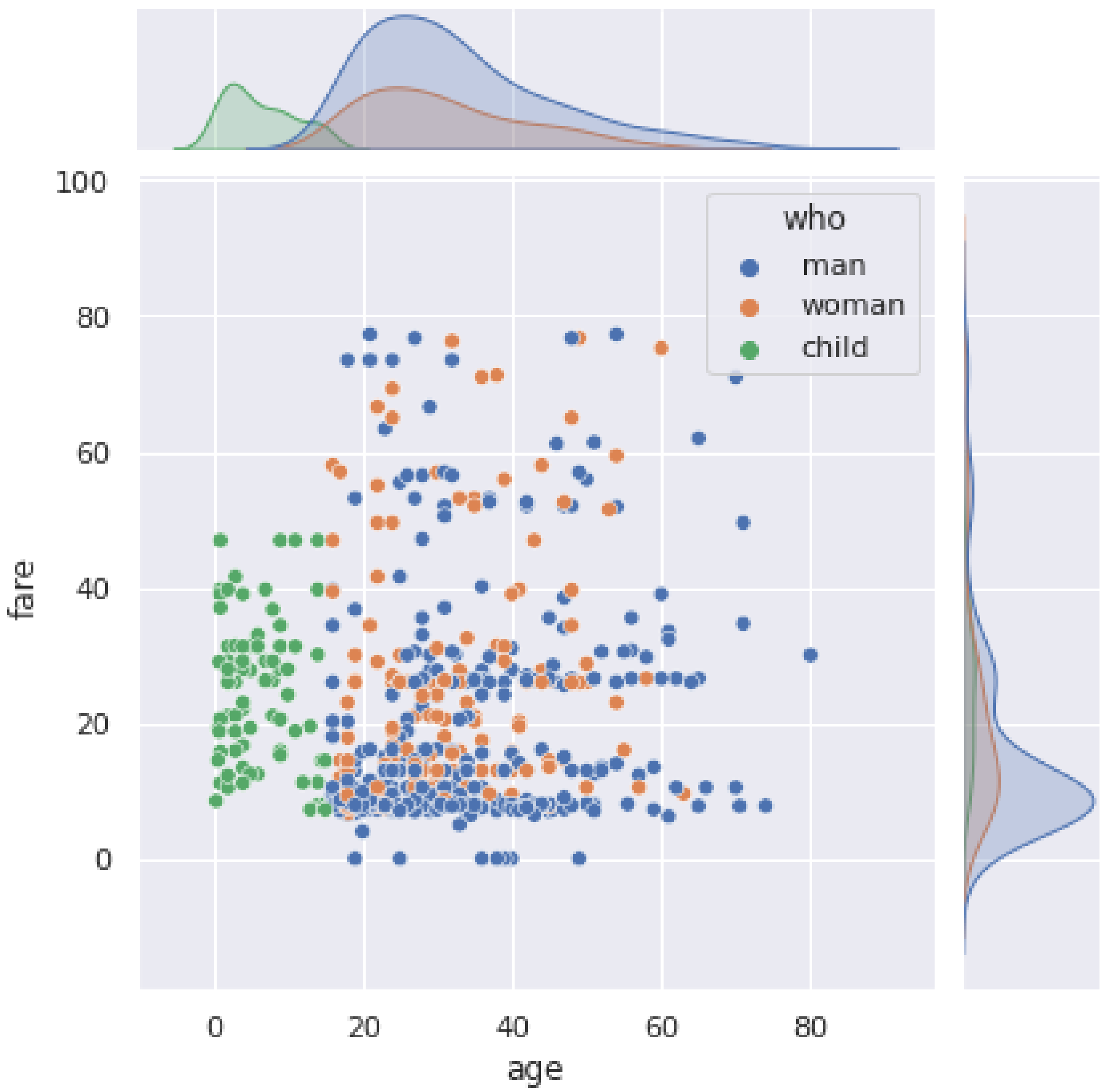
```
<seaborn.axisgrid.JointGrid at 0x7f9fa6f080d0>
```



What we could do is remove outliers, if I recall correclty we set a upper bound of 77.5, let's do that here and replot. I've also added the type of person as a color, you can here see that women and children pay more as we saw earlier.

```
sns.jointplot(data=titanic_df[titanic_df.fare<77.5], x='age', y='fare', hue='who')
```

```
<seaborn.axisgrid.JointGrid at 0x7f9fa6ac27f0>
```



To make this more mathematically sound, we are using the spearman rank correlation test, not the pearson as we are dealing with non normal data. You could check that with a shapiro wilk test but i'll leave that up to you!

```
corr, p = stats.spearmanr(a=titanic_df[['age','fare']].dropna())
corr, p
```

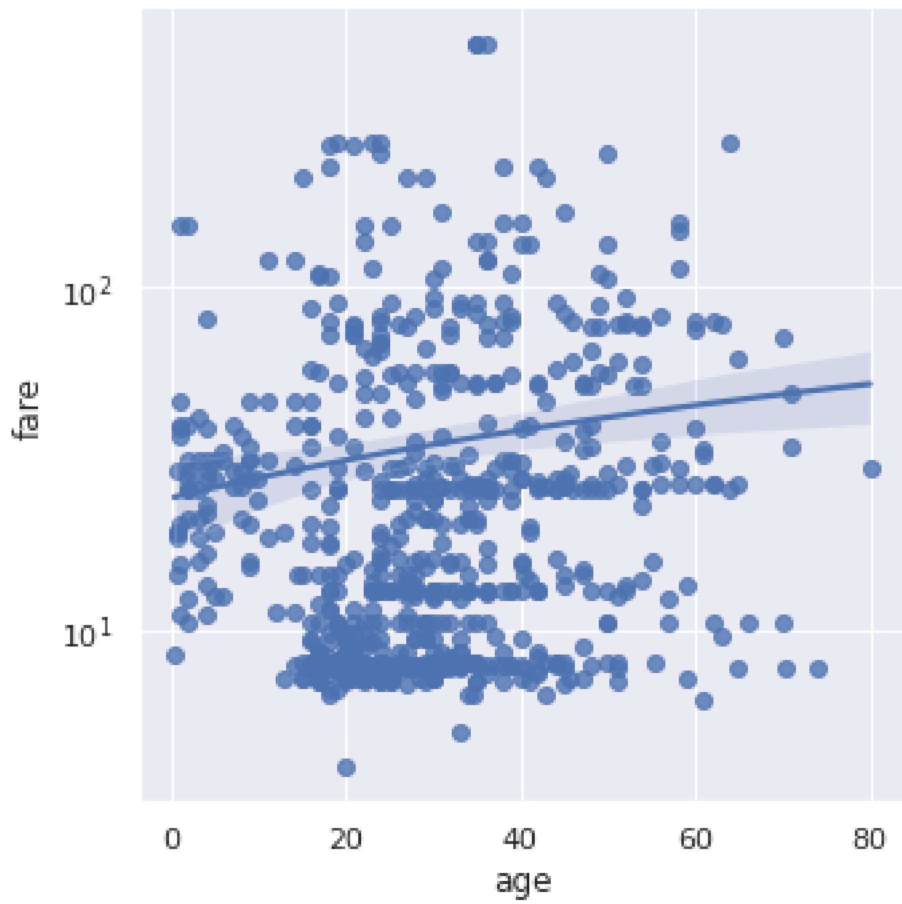
```
(0.13505121773428 78, 0.00029580903243060916)
```

with a p-value of only 0.000296 we can safely reject the null-hypothesis, meaning there is a correlation. The correlation coefficient here is only 0.135, meaning for any person each year of age would make their fare about 0.135 dollars more expensive on average, which in that time was a fair amount of money.

To make this more visual, I added a lmplot that performs a linear regression, you can see how the line goes up in fare as the age goes up. I had to use a logarithmic y-scale as the distribution is still not normal.

```
ax = sns.lmplot(data=titanic_df, x='age', y='fare')
ax.set(yscale='log')
```

```
<seaborn.axisgrid.FacetGrid at 0x7f9fa4916100>
```



Now this correlation of 0.135 dollar is relevant for ANY person, man, female, child, first class, second,...

Perhaps we could find several subgroups with a higher or lower correlation, I will perform the correlation with the outliers removed.

```
corr, p = stats.spearmanr(a=titanic_df[titanic_df.fare<77.5][['age','fare']].dropna())
corr, p
```

```
(0.09269934275477329, 0.019762193968013368)
```

When we remove outliers, we have a less strong correlations, indicating that the outliers - with high fares - are in general older persons.

Try to experiment with subsetting the data and find a group where age matters more for the correlation.

CHAPTER

TWENTYEIGHT

NEW DATA SOURCES

In this notebook we are going to look into adding new data to your dataset. We start out with a taxi dataset describing all pickup points from taxis in a specific date interval, notice that the dataset is divided up into months. Each month has their specific csv file saved in an AWS location.

```
import pandas as pd
import seaborn as sns
from urllib.request import urlopen
```

```
data_url_files = urlopen('https://raw.githubusercontent.com/toddwschneider/nyc-taxi-
↪data/master/setup_files/raw_data_urls.txt')
data_urls = data_url_files.read().decode('utf-8').split('\n')
data_urls[0:12]
```

```
['https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-01.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-02.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-03.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-04.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-05.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-06.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-07.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-08.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-09.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-10.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-11.csv',
 'https://s3.amazonaws.com/nyc-tlc/trip+data/fhv_tripdata_2015-12.csv']
```

Due to slow parsing of data we will here only parse the uber data from jan-mar 2015

```
datasets = [pd.read_csv(url) for url in data_urls[0:3]]
```

```
cab_df = pd.concat(datasets)
```

```
print('shape: ' + str(cab_df.shape))
cab_df.head()
```

```
shape: (9153861, 3)
```

```
  Dispatching_base_num          Pickup_date  locationID
0               B00013  2015-01-01 00:30:00         NaN
1               B00013  2015-01-01 01:22:00         NaN
2               B00013  2015-01-01 01:23:00         NaN
```

(continues on next page)

(continued from previous page)

```
3               B00013  2015-01-01 01:44:00         NaN
4               B00013  2015-01-01 02:00:00         NaN
```

We would like to find out how many uber rides were performed each day so we:

- parse the date string to a datetime format
- set the date as index
- resample to '1D' or one day (and chose count as aggregation)

```
cab_df['datetime'] = pd.to_datetime(cab_df['Pickup_date'], format="%Y/%m/%d %H:%M:%S")
```

```
cab_df = cab_df.set_index('datetime')
```

```
cab_df.head()
```

```
                    Dispatching_base_num          Pickup_date  locationID
datetime
2015-01-01 00:30:00               B00013  2015-01-01 00:30:00         NaN
2015-01-01 01:22:00               B00013  2015-01-01 01:22:00         NaN
2015-01-01 01:23:00               B00013  2015-01-01 01:23:00         NaN
2015-01-01 01:44:00               B00013  2015-01-01 01:44:00         NaN
2015-01-01 02:00:00               B00013  2015-01-01 02:00:00         NaN
```

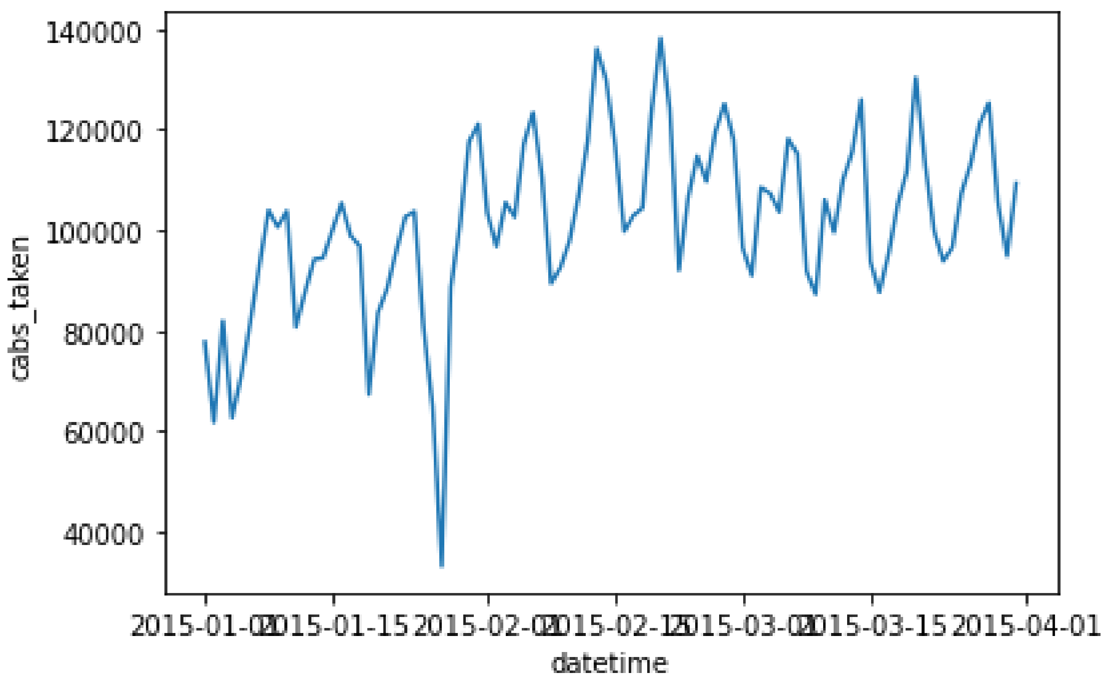
```
cabs_taken = cab_df['Dispatching_base_num'].resample('1D').count().rename('cabs_taken
↪')
cabs_taken.head()
```

```
datetime
2015-01-01    77789
2015-01-02    61832
2015-01-03    81955
2015-01-04    62691
2015-01-05    71063
Freq: D, Name: cabs_taken, dtype: int64
```

great! now we have an idea on how many ubers were taken each day, let us use a simple line plot to show the results.

```
sns.lineplot(data=cabs_taken)
```

```
<AxesSubplot:xlabel='datetime', ylabel='cabs_taken'>
```



This dataset is nice, but by itself pretty useless, why don't we look up some weather information to see if this influences our traffic.

```
url = 'https://raw.githubusercontent.com/toddwschneider/nyc-taxi-data/master/data/
↪central_park_weather.csv'
weather = pd.read_csv(url)
```

```
weather.head()
```

```
       STATION                         NAME        DATE   AWND  PRCP  SNOW  \
0  USW00094728  NY CITY CENTRAL PARK, NY US  2009-01-01  11.18   0.0   0.0
1  USW00094728  NY CITY CENTRAL PARK, NY US  2009-01-02   6.26   0.0   0.0
2  USW00094728  NY CITY CENTRAL PARK, NY US  2009-01-03  10.07   0.0   0.0
3  USW00094728  NY CITY CENTRAL PARK, NY US  2009-01-04   7.61   0.0   0.0
4  USW00094728  NY CITY CENTRAL PARK, NY US  2009-01-05   6.93   0.0   0.0

   SNWD  TMAX  TMIN
0   0.0    26    15
1   0.0    34    23
2   0.0    38    29
3   0.0    42    25
4   0.0    43    38
```

you can see a variaty of information, more info on the column names can be found hereagain we need to:

- parse the date
- set it to the index
- resampling is not needed as it is already in day-to-day intervals

```
weather['DATE'] =  pd.to_datetime(weather['DATE'], format="%Y/%m/%d")
weather = weather.set_index('DATE')
```

```
weather.head()
```

```
                  STATION                          NAME   AWND  PRCP  SNOW  SNWD  \
DATE
2009-01-01  USW00094728  NY CITY CENTRAL PARK, NY US  11.18   0.0   0.0   0.0
2009-01-02  USW00094728  NY CITY CENTRAL PARK, NY US   6.26   0.0   0.0   0.0
2009-01-03  USW00094728  NY CITY CENTRAL PARK, NY US  10.07   0.0   0.0   0.0
2009-01-04  USW00094728  NY CITY CENTRAL PARK, NY US   7.61   0.0   0.0   0.0
2009-01-05  USW00094728  NY CITY CENTRAL PARK, NY US   6.93   0.0   0.0   0.0

            TMAX  TMIN
DATE
2009-01-01    26    15
2009-01-02    34    23
2009-01-03    38    29
2009-01-04    42    25
2009-01-05    43    38
```

Having 2 dataset, now we need to merge them. Since we already prepared the date as index, this should be easy.

```
merged_df = pd.merge(cabs_taken, weather, left_index=True, right_index=True)
```

```
merged_df.head()
```

```
            cabs_taken      STATION                          NAME   AWND  PRCP  \
2015-01-01       77789  USW00094728  NY CITY CENTRAL PARK, NY US   7.16  0.00
2015-01-02       61832  USW00094728  NY CITY CENTRAL PARK, NY US   7.16  0.00
2015-01-03       81955  USW00094728  NY CITY CENTRAL PARK, NY US   6.49  0.71
2015-01-04       62691  USW00094728  NY CITY CENTRAL PARK, NY US   6.49  0.30
2015-01-05       71063  USW00094728  NY CITY CENTRAL PARK, NY US  10.51  0.00

            SNOW  SNWD  TMAX  TMIN
2015-01-01   0.0   0.0    39    27
2015-01-02   0.0   0.0    42    35
2015-01-03   0.0   0.0    42    33
2015-01-04   0.0   0.0    56    41
2015-01-05   0.0   0.0    49    21
```

One would assume that when it is a rainy day, people would use more cabs. so let us seperate based on precipitation.

```
rained = merged_df[merged_df['PRCP']>0]
no_rain = merged_df[merged_df['PRCP']==0]
```

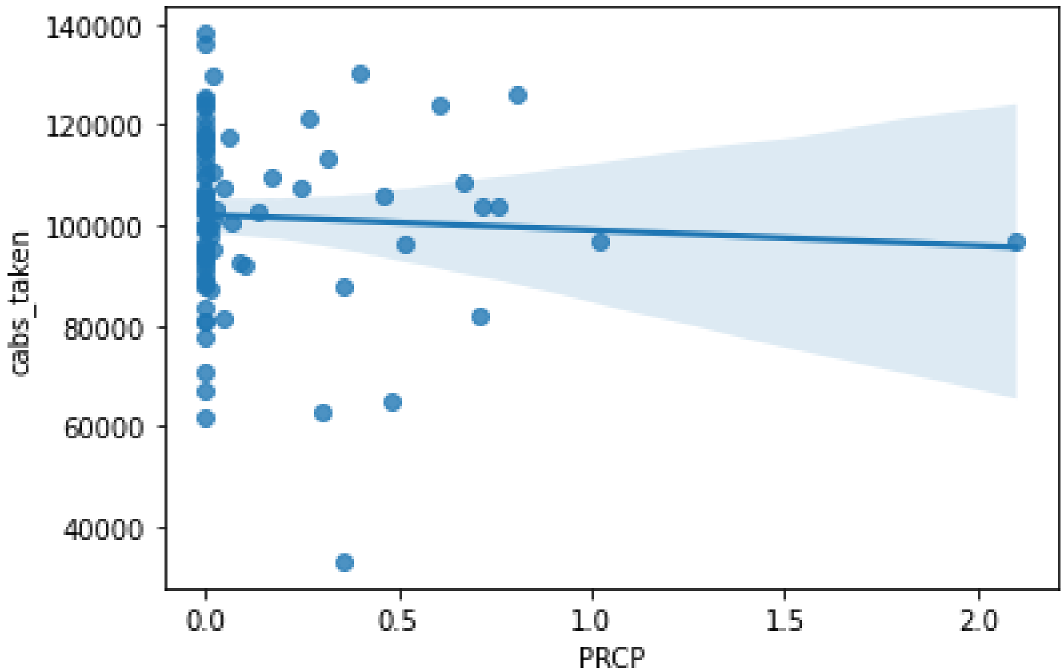
```
print('average uber rides on a rainy day')
print(rained['cabs_taken'].mean())
print('average uber rides on a dry day')
print(no_rain['cabs_taken'].mean())
```

```
average uber rides on a rainy day
99837.29411764706
average uber rides on a dry day
102846.30357142857
```

ouch! it looks like the average new yorker doesn't mind getting wet, or they take a cab any day…using a regression plot we can see it more clear

```
sns.regplot(data=merged_df, x='PRCP', y='cabs_taken')
```

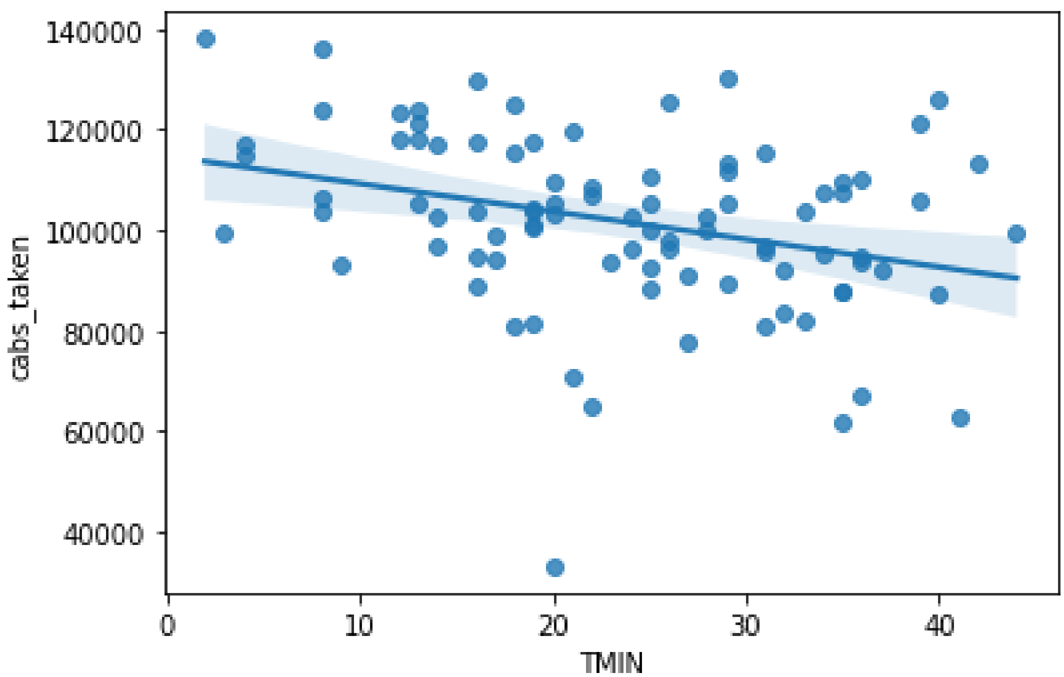
```
<AxesSubplot:xlabel='PRCP', ylabel='cabs_taken'>
```



Ok, here we see that it might just be because a lot of days are dry and the dataset is skewed. Not reliable info.What about temperatures, can we see a difference if the lowest temperature changes?

```
sns.regplot(data=merged_df, x='TMIN', y='cabs_taken')
```

```
<AxesSubplot:xlabel='TMIN', ylabel='cabs_taken'>
```



Appearantly when the temperature lowers, yorkers seem to be taking more cab rides. So global warming might be disastrous for capitalism after all?

CHAPTER

TWENTYNINE

FEATURE ENHANCING

This rather simple notebook is a small illustration how feature enhancing might work in specific cases, we have a dataset containing cars and their fuel efficiency. What we will try to illustrate here is that sometimes combinations or formulas using the original data might display patterns not visible with the previous data.

```
import pandas as pd
import seaborn as sns
from scipy.stats import spearmanr
```

we load the mpg dataset and have a look at it.

```
mpg = sns.load_dataset('mpg')
```

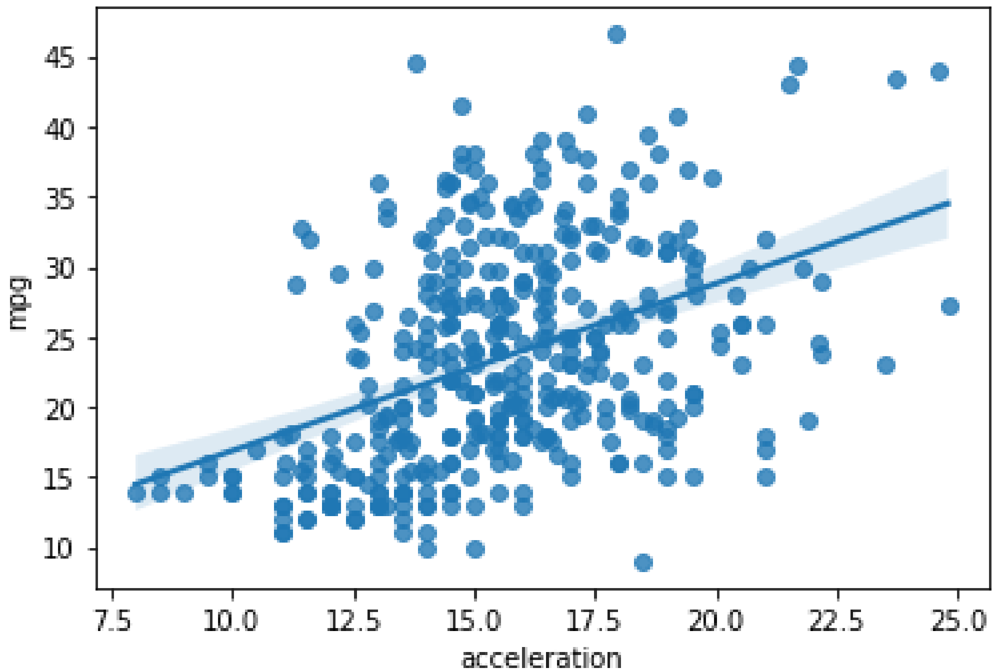
```
mpg.head()
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year origin                       name
0          70    usa  chevrolet chevelle malibu
1          70    usa          buick skylark 320
2          70    usa         plymouth satellite
3          70    usa              amc rebel sst
4          70    usa                ford torino
```

We'll try to explore our dataset by printing out some regression plots between features of the car and the mileage per gallon.

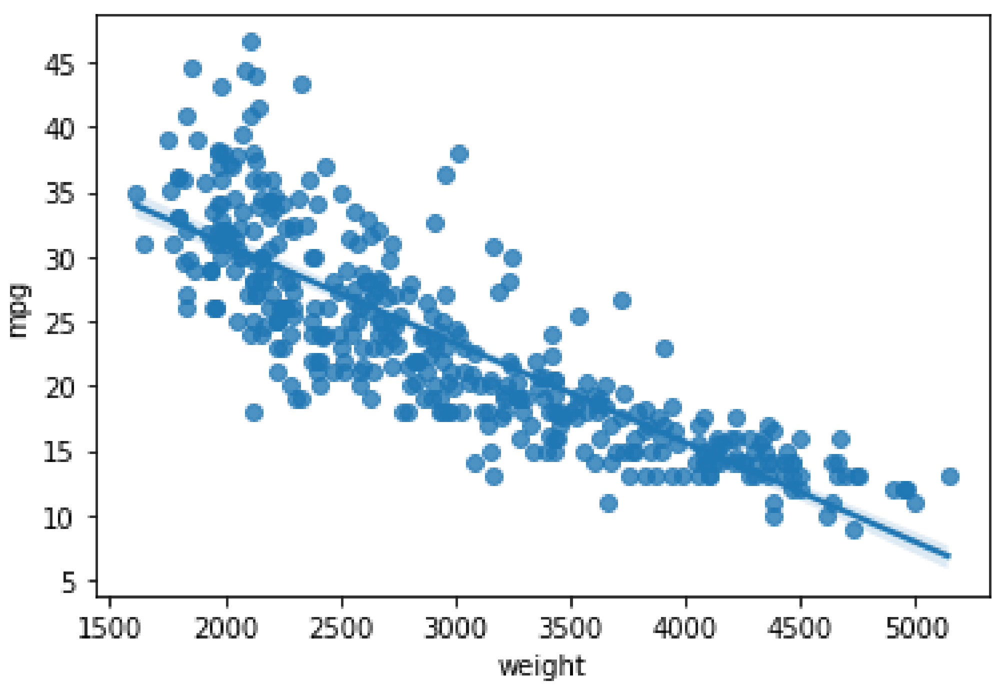
```
sns.regplot(x=mpg['acceleration'], y=mpg['mpg'])
corr, p = spearmanr(mpg['mpg'], mpg['acceleration'])
print('acceleration correlation: ' + str(100*round(corr,4)) + "%")
```

```
acceleration correlation: 43.87%
```




```
sns.regplot(x=mpg['weight'], y=mpg['mpg'])
corr, p = spearmanr(mpg['mpg'], mpg['weight'])
print('weight correlation: ' + str(100*round(corr,4)) + "%")
```

```
weight correlation: -87.49%
```



We can see that the acceleration has a positive influence on the miles per gallon, whilst the weight has a negative influence, what about the acceleration per weight?

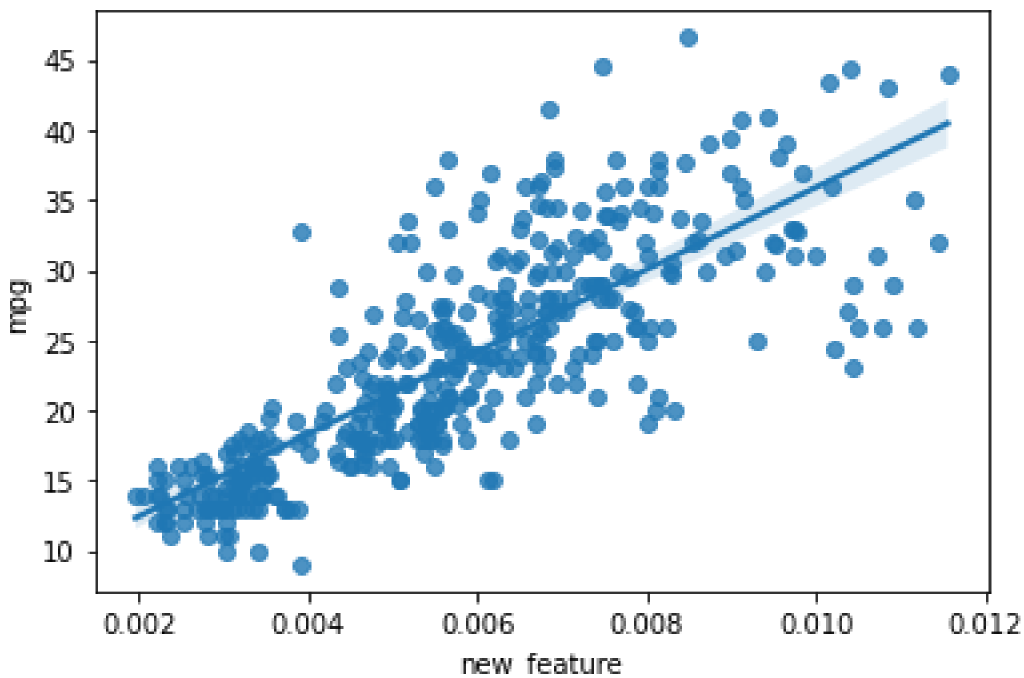
```
mpg['new_feature'] = mpg['acceleration']/mpg['weight']
```

```
mpg.head()
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year origin                       name  new_feature
0          70    usa  chevrolet chevelle malibu     0.003425
1          70    usa          buick skylark 320     0.003114
2          70    usa         plymouth satellite     0.003201
3          70    usa              amc rebel sst     0.003495
4          70    usa                ford torino     0.003044
```

```
sns.regplot(x=mpg['new_feature'], y=mpg['mpg'])
corr, p = spearmanr(mpg['mpg'], mpg['new_feature'])
print('new feature correlation: ' + str(100*round(corr,4)) + "%")
```

```
new feature correlation: 84.19%
```



It seems we are not able to create a new feature with even more correlation, not every story has to be a success. We can report this to our boss and explain the results.

CHAPTER

THIRTY

CLUSTER ANALYSIS

Before starting this notebook I would like to state that what is explained here will be elaborated later in the course and might look complicated at this point. If you do not feel familiar with these concepts that is perfectly fine.

```
import pandas as pd
import seaborn as sns
```

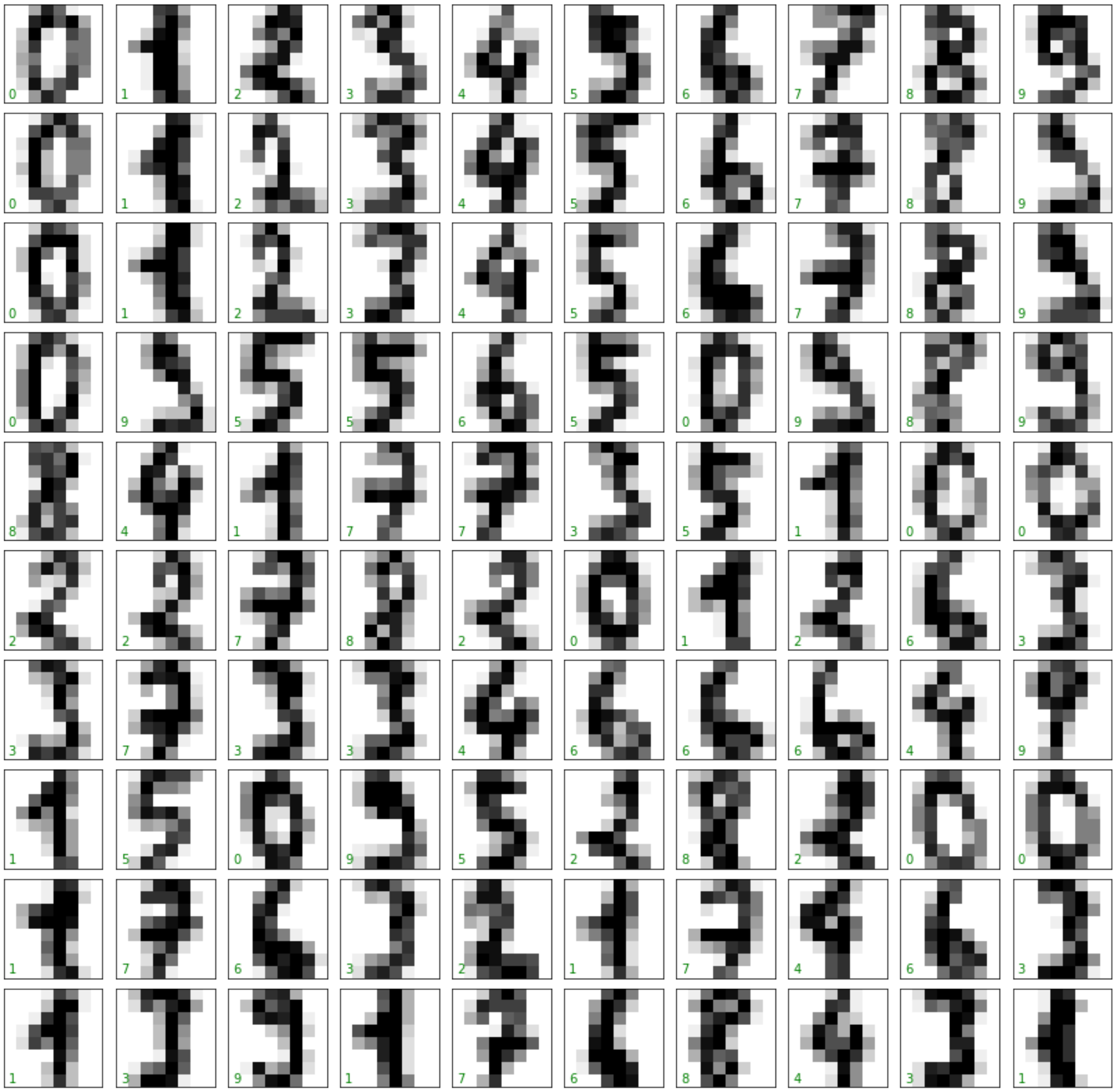
We will load a digits dataset from sklearn, the machine learning library, these are 8x8 pixel images showing handwritten digits with the correct answer.In the dataset there are 1797 images giving the dataset a dimension of (1797, 8*8)

```
from sklearn.datasets import load_digits
digits = load_digits()
digits.data.shape
```

```
(1797, 64)
```

Before we start, let's print out a few of them, the following cell will do that. Again, plotting is not yet seen, so the following cells might be overwhelming.

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(10, 10, figsize=(16, 16),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap='binary', interpolation='nearest')
    ax.text(0.05, 0.05, str(digits.target[i]),
            transform=ax.transAxes, color='green')
```



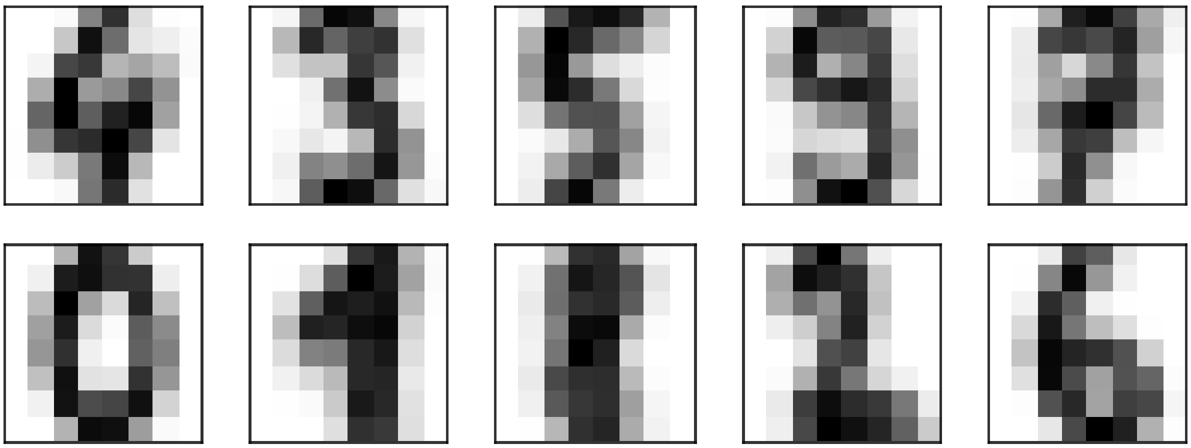
In cluster analysis we will try to figure out clusters within the dataset, keep in mind that these cluster are constructed without knowning the correct answer. Here we use the Isomap algorithm to create clusters, by using fit and transform methods we can create the clusters

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=10, random_state=0)
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

```
(10, 64)
```

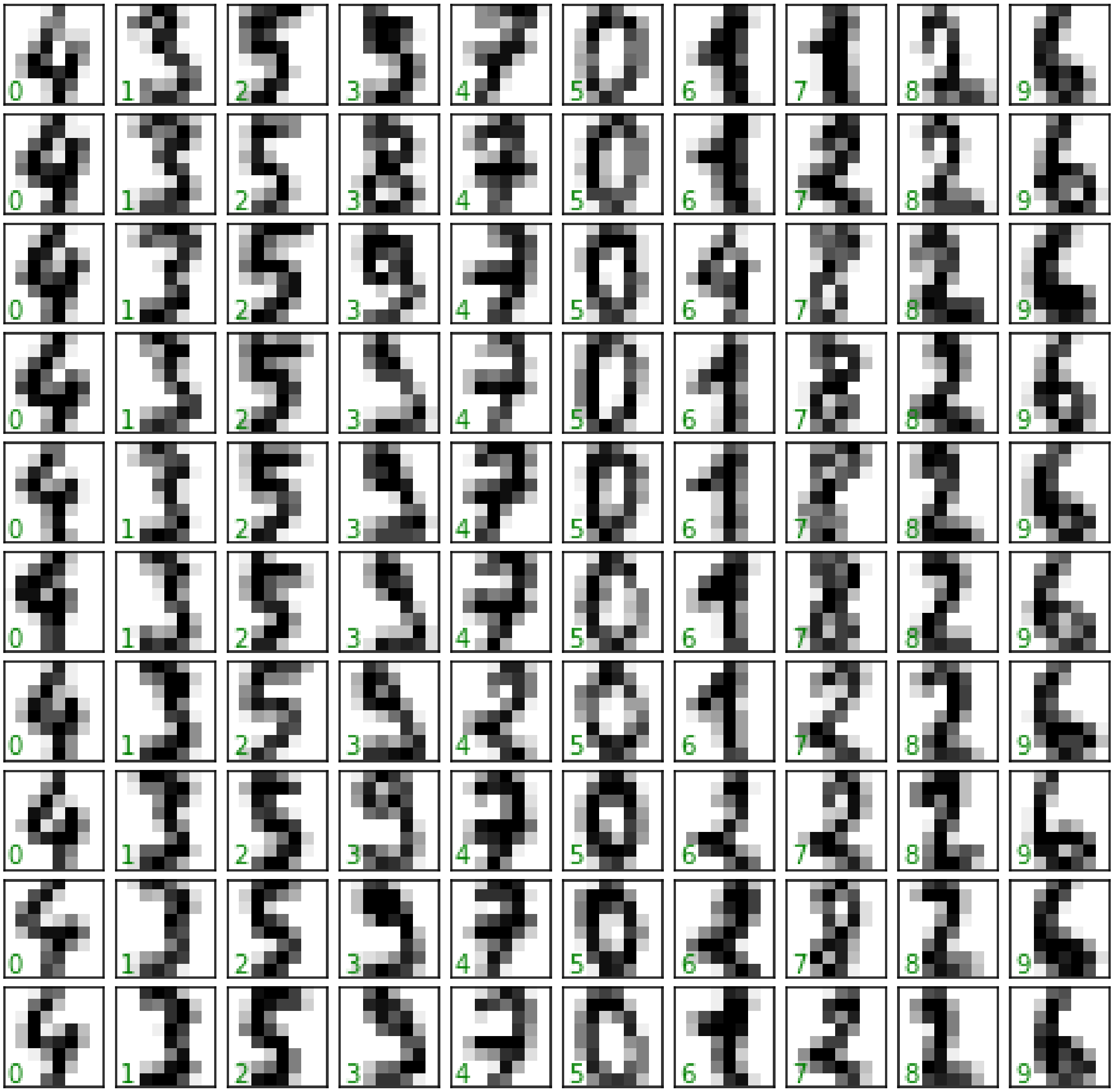
Now that the algorithm seperated the dataset into 10 clusters, we can ask it to print the center of each cluster.This gives us an idea how the average digit in that cluster looks like.

```
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
    axi.set(xticks=[], yticks=[])
    axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```



Those look similar to the actual numbers, confirming that arabic numbers have good visual seperation inbetween.Aside from the centers we can also print a few examples from the clusters.

```
fig, axes = plt.subplots(10, 10, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[clusters==i%10][int(i/10)], cmap='binary', interpolation=
↪'nearest')
    ax.text(0.05, 0.05, str(i%10),
            transform=ax.transAxes, color='green')
```



You can see that the cluster number does not match the actual number, that's because our algorithm does not understand which numbers there are.It does however understand the differences between the numbers! This technique can also be used for other datasets where no outcome is given, but we would like to separate our dataset into clusters.

To make this more visible we will use another example of a dataset about the leafs of 3 types of iris flowers.

```
iris_df = sns.load_dataset('iris')
iris_df.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

What we could do here is ask the algorithm to create 3 clusters of records, as the dataset contains 3 types of iris flowers.

We do not supply the algorithm with the information of the species, yet it has to figure out by itself how to seperate the records.

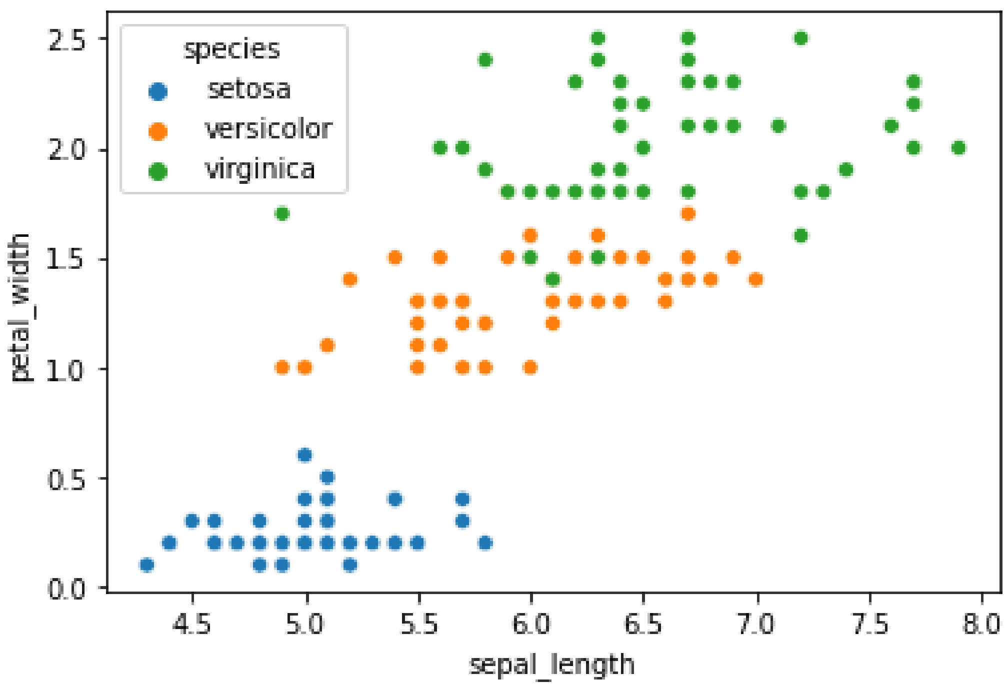
```
kmeans = KMeans(n_clusters=3, random_state=0)
iris_df['cluster'] = kmeans.fit_predict(iris_df.drop(columns='species'))
iris_df.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width species  cluster
0           5.1          3.5           1.4          0.2  setosa        1
1           4.9          3.0           1.4          0.2  setosa        1
2           4.7          3.2           1.3          0.2  setosa        1
3           4.6          3.1           1.5          0.2  setosa        1
4           5.0          3.6           1.4          0.2  setosa        1
```

We can see our data now has an additional feature cluster which contains either 0, 1 or 2. If the clustering has been performed as expected, the clusters should coincide with the species. Using a plot we can find out.

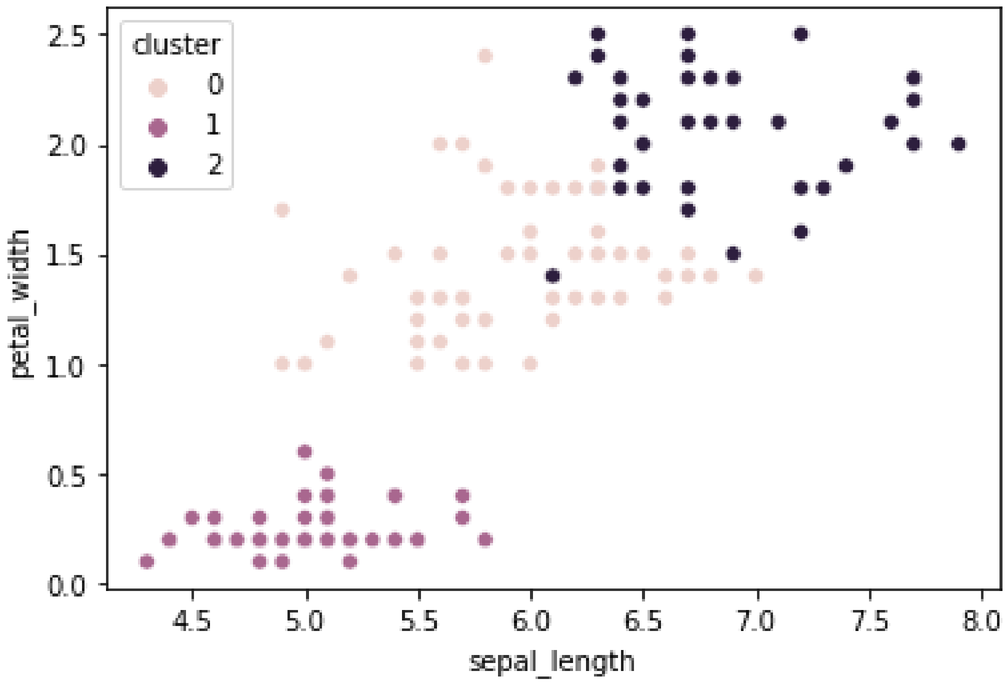
```
sns.scatterplot(data=iris_df, x='sepal_length', y='petal_width', hue='species')
```

```
<AxesSubplot:xlabel='sepal_length', ylabel='petal_width'>
```




```
sns.scatterplot(data=iris_df, x='sepal_length', y='petal_width', hue='cluster')
```

```
<AxesSubplot:xlabel='sepal_length', ylabel='petal_width'>
```



For some reason seaborn thinks it is useful to change color scheme, yet you can see that there is an uncanny similarity between the clusters and the species, the algorithm was succesful in finding the different species.

Without giving the information we were able to cluster the different species of iris flowers yet we have no idea which cluster belongs to which species. It is the reasers responsibility to take conclusion in what the different clusters mean!

CHAPTER

THIRTYONE

VIF: VARIANCE INFLATION FACTOR

in this notebook we will investigate the variance inflation which can occur in a dataset. As an example here, we will use the 'Mile Per Gallon' dataset contianing a set of cars and their fuel efficiency. Some columns in the dataset might

```
import pandas as pd
import seaborn as sns
from statsmodels.stats.outliers_influence import variance_inflation_factor
mpg = sns.load_dataset('mpg')
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_13569/3881140987.py in <module>
      1 import pandas as pd
      2 import seaborn as sns
----> 3 from statsmodels.stats.outliers_influence import variance_inflation_factor
      4 mpg = sns.load_dataset('mpg')

ModuleNotFoundError: No module named 'statsmodels'
```

```
mpg.head()
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year origin                       name
0          70    usa  chevrolet chevelle malibu
1          70    usa          buick skylark 320
2          70    usa         plymouth satellite
3          70    usa              amc rebel sst
4          70    usa                ford torino
```

as you can see, we also imported a function 'variance_inflation_factor' which will help us calculate this, more information can be found on wikipedia.

to use the function, we refer to the documentation. The function is a bit stubborn and requires the following:

- only numerical values (so we to drop the categories)
- no nan values (dropping nans)
- as a numpy array instead of a pandas dataframe

```
cols_to_keep = ['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration',
↪'model_year']
vif_compatible_df = mpg[cols_to_keep]
vif_compatible_df = vif_compatible_df.dropna(axis='index')
vif_compatible_df = vif_compatible_df.values
vif_compatible_df
```

```
array([[   8. ,  307. ,  130. , 3504. ,   12. ,   70. ],
       [   8. ,  350. ,  165. , 3693. ,   11.5,   70. ],
       [   8. ,  318. ,  150. , 3436. ,   11. ,   70. ],
       ...,
       [   4. ,  135. ,   84. , 2295. ,   11.6,   82. ],
       [   4. ,  120. ,   79. , 2625. ,   18.6,   82. ],
       [   4. ,  119. ,   82. , 2720. ,   19.4,   82. ]])
```

this looks a lot different! we don't know anymore what all of that means, but the computer does, now we run it through the function. Notice how we have to specify a specific column, the resulting inflation factor is that for the chosen column

```
# we pick column 0 which is 'cylinders' according to cols_to_keep
variance_inflation_factor(vif_compatible_df, 0)
```

```
115.97777160980726
```

```
for idx, col in enumerate(cols_to_keep):
  print(col + ": \t" + str(variance_inflation_factor(vif_compatible_df, idx)))
```

```
cylinders:         115.97777160980726
displacement:         86.48595590611876
horsepower:         60.25657462146676
weight:         137.4717563697324
acceleration:         69.40087667701684
model_year:         109.3200159587966
```

31.1 TODO

The variance inflation gives a numerical value to how little variation there is between one column and the others in a dataset, you will see how the numbers will gradually go down as you remove more and more columns.This way we have a quantifyable method of removing data from our dataset in case there is too much 'duplicate' information.There is no real cut-off value that specifies of a column should or should not be removed, so make sure you can argument your decision.

- experiment with removing columns in the cols_to_keep list
- What do you think would be the ideal dataset here? we would like to predict the fuel economy (mpg) of a car.

CHAPTER

THIRTYTWO

PRINCIPLE COMPONENT ANALYSIS

In this notebook we will not try to remove data from our dataset, but transform the variation in our features (columns) into less features.We will do this using the concept of PCA (principle component analysis). The dataset we will be using here is about the dimensions of iris flowers, in total 150 flowers were measured of 3 species.

```
import pandas as pd
import seaborn as sns
from sklearn.decomposition import PCA
iris = sns.load_dataset('iris')
```

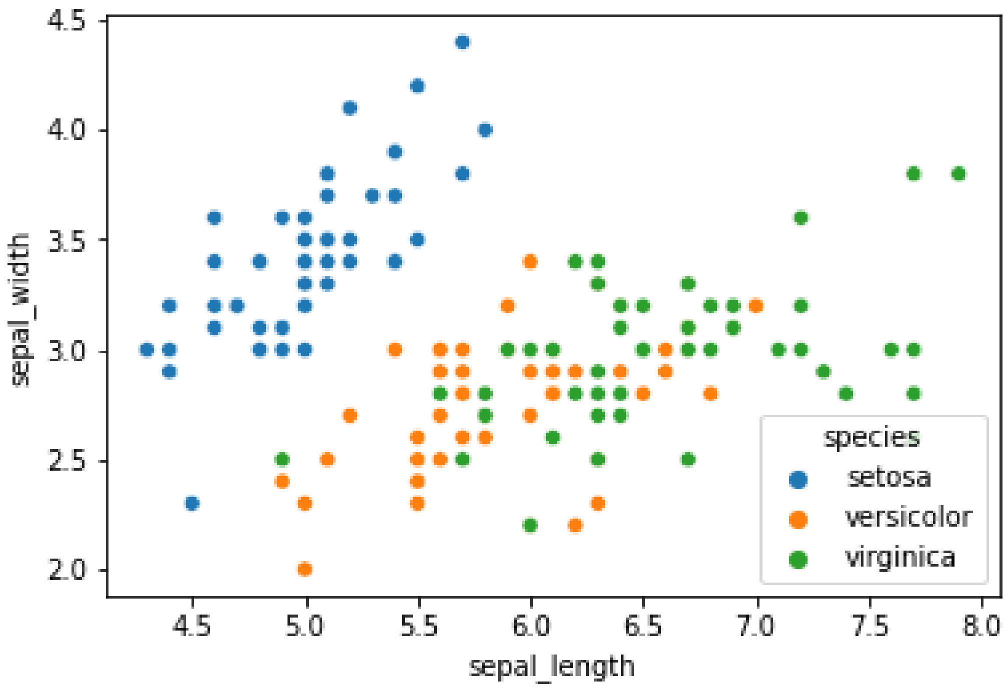
you can see that we imported a function PCA from sklearn, this will do the calculations for us, but we still need to specify some parameters.Before we do that, let us use the first 2 columns of the dataset to plot a scatter and see if we can distinguish the different species of flowers.

```
iris.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

```
sns.scatterplot(x=iris['sepal_length'], y=iris['sepal_width'], hue=iris['species'])
```

```
<AxesSubplot:xlabel='sepal_length', ylabel='sepal_width'>
```



That already looks pretty good, but versicolor and virginica are still hard to differentiate. Let's see if we can compress the variation of all 4 columns into 2 axi.We do this by creating a PCA transformer and specifying we want only 2 output components

```
pca = PCA(n_components=2)
```

We also need to prepare our dataframe, we do this by only dropping our outcome (that which we do not need for the transform)

```
X = iris.drop(columns='species')
X.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width
0           5.1          3.5           1.4          0.2
1           4.9          3.0           1.4          0.2
2           4.7          3.2           1.3          0.2
3           4.6          3.1           1.5          0.2
4           5.0          3.6           1.4          0.2
```

```
iris_pca = pca.fit_transform(X)
pd.DataFrame(iris_pca, columns=['PC1', 'PC2'])
```

```
          PC1       PC2
0   -2.684126  0.319397
1   -2.714142 -0.177001
2   -2.888991 -0.144949
3   -2.745343 -0.318299
4   -2.728717  0.326755
..        ...       ...
145  1.944110  0.187532
146  1.527167 -0.375317
147  1.764346  0.078859
```

(continues on next page)

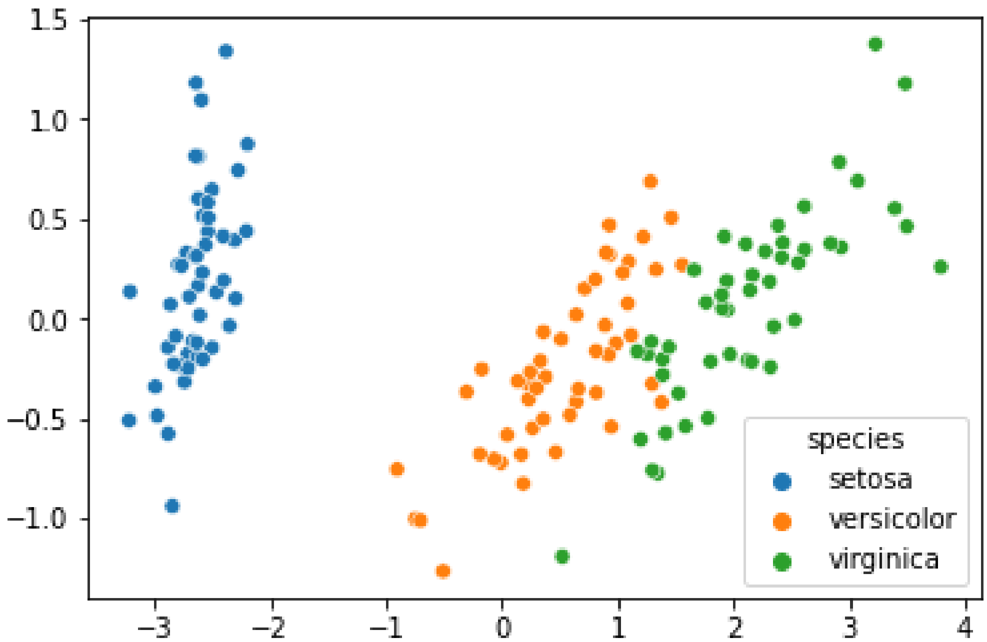
(continued from previous page)

```
148  1.900942  0.116628
149  1.390189 -0.282661

[150 rows x 2 columns]
```

Running it through the PCA transformer using the fit_transform function gives us a numpy 2 dimensional array (which is similar to a pandas dataframe) with 2 columns.When inserted into a scatter plot they show us (nearly) all variance of 4 columns compressed into a 2 dimensional plot.

```
sns.scatterplot(x=iris_pca[:,0], y=iris_pca[:,1], hue=iris['species'])
```

```
<AxesSubplot:>
```



32.1 TODO

it is clear that this function is very potent concerning data visualisation, do you think you can improve on the mpg dataset?

- experiment with the PCA transformer using the mpg dataset

```
mpg = sns.load_dataset('mpg')
mpg.head()
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0    3504          12.0
1  15.0          8         350.0       165.0    3693          11.5
2  18.0          8         318.0       150.0    3436          11.0
3  16.0          8         304.0       150.0    3433          12.0
4  17.0          8         302.0       140.0    3449          10.5

   model_year origin                       name
```

(continues on next page)

(continued from previous page)

```
0          70    usa  chevrolet chevelle malibu
1          70    usa          buick skylark 320
2          70    usa         plymouth satellite
3          70    usa              amc rebel sst
4          70    usa                ford torino
```

Part VI

6. Machine Learning

CHAPTER

THIRTYTHREE

MACHINE LEARNING

this is an introduction