Data Science - A practical Approach

Lorenz Feyen

Sep 28, 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.

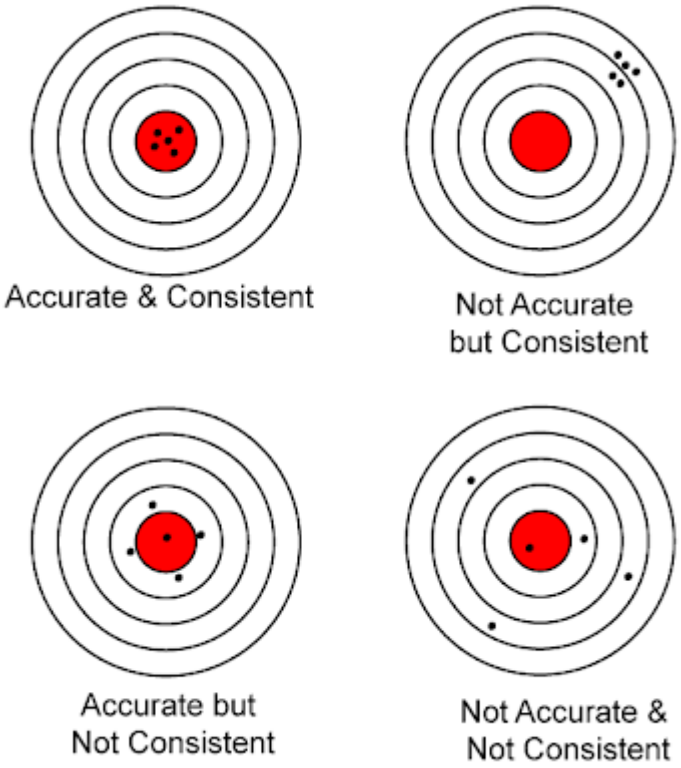
2.1.1 Accuracy

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

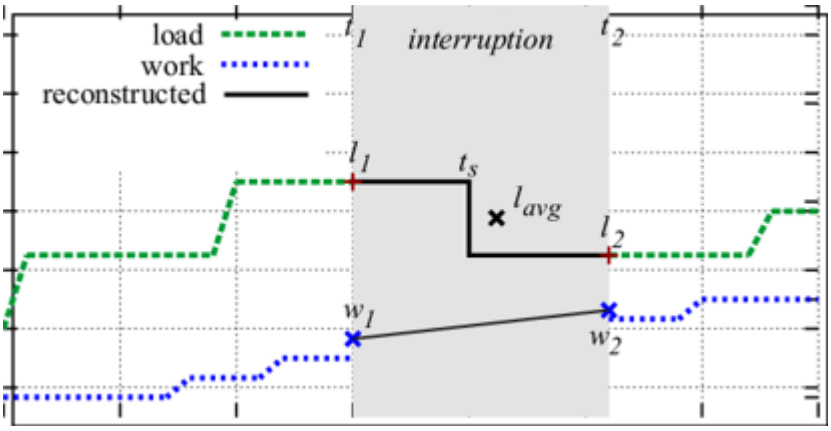
As hinted in the previous point, completeness is something you have to be aware of. Having 'complete' data is crucial for you narrative to give a correct answer, as you might otherwise lose detail. Note that you never will know if your data is complete as there might always be more data to mine. Yet 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 psysical 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_25543/1298831137.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

```
import pandas as pd
```

```
wafer_df = pd.read_csv('https://openmv.net/file/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
```

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

```
range_df = (wafer_df-wafer_df.quantile(0.5))/iqr
```

```
range_df[(range_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
```

```
range_df[(range_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
```

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

```
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
```

CHAPTER

SIX

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

SEVEN

DATA PREPROCESSING

this is an introduction

CHAPTER

EIGHT

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
```

Part IV

4. Data Exploration

CHAPTER

NINE

DATA EXPLORATION

this is an introduction

Part V

5. Data Visualisation

CHAPTER

TEN

DATA VISUALISATION

this is an introduction

Part VI

6. Machine Learning

CHAPTER

ELEVEN

MACHINE LEARNING

this is an introduction