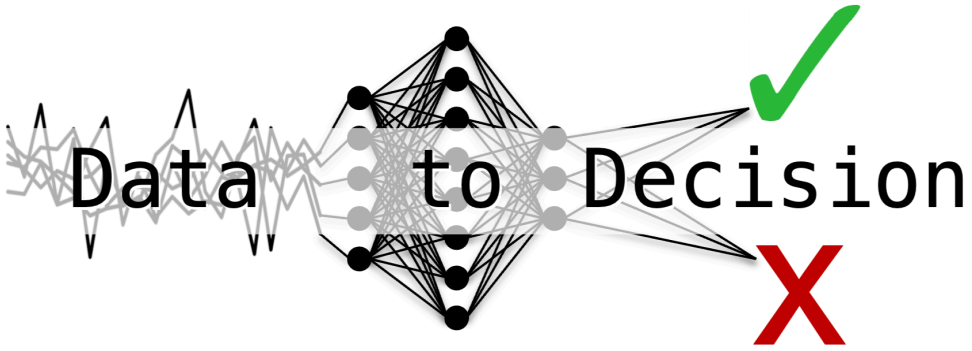
Data to Decision - IND320

Kristian Hovde Liland

Aug 16, 2023

CONTENTS

The contents of this Jupyter Book is the basis for the course "Data to decision" codenamed IND320 given at the Norwegian University of Life Sciences for the first time in the autumn of 2023.



Status

This book is under construction and will be filled and updated during the autumn semester of 2023.

Audience

- The main audience will be those who follow DAT320 as regular students.
- Oters are free to use the book.

Structure

The structure and organisation of this book means it can be used in several ways:

- A compiled static book with text, graphics and Python examples.
- A collection of interactive Jupyter Notebooks.
- A collection of presentations (through support for the RISE extension).

Technical basis

Keys and passwords

Code in the book sometimes refer to the folder "No_sync" which is on the same folder level as D2D (the book). This folder is excluded from the GitHub sync due to containing API keys and passwords. If you want to run examples referring

to the folder, please create one on your computer and fill with appropriate keys and passwords for your own licenses.

Software requirements

- Basic functionality requires Python (originally built with version 3.10.6) and packages *matplotlib*, *numpy*, *json*, *datetime*.
- For databases:
  - E.g., docker with Cassandra database having the tables expected.
  - Python packages *cassandra-driver*, *cql*
- For APIs:
  - Python packages *requests*, *flask*
- For machine learning:
  - Python packages *pandas*, *scikit-learn* (or quicker/more scalable. implementation)
- Compiling the full book requires several programs and packages, but all Jupyter Notebooks can be run separately.
  - Python packages: *jupyter-book* and all above.
  - A running Flask server with the *flask_API.py* active.
  - Docker with Cassandra running locally.

Part I

Data-Driven Decision Making

CHAPTER

ONE

DATA PIPELINE

1.1 DDDM

- We will only cover the basics of DDDM.
- Many facets of "Data to Decision" can be placed under the DDDM umbrella.

1.1.1 Data-Driven

- Make informed decisions that are based on facts rather than intuition or guesswork, i.e., without bias or emotion.
- Clive Humby 2006 -"Data is the new oil"
  - Big data
  - Blindly collecting data and looking for patterns is not likely to result in long term profits and revolutionary discoveries.

1. Know your objectives

- A well-rounded data analyst:
  - Understands the business and the competitive market.
  - Asks the right questions about the problems in the industry.
  - Identifies and understands the problems thoroughly.
  - Makes better inferences with data based on the foundational knowledge.
- Before collecting data, a data analyst should:
  - Identify the business questions that need to be answered to achieve organizational goals.
  - Determine the precise questions that need to be answered to inform the strategy.
  - Streamline the data collection process and avoid wasting resources.

2. Find relevant data

- **Identify the data sources**. This could include databases, web-driven feedback forms, social media, and other sources.
- **Coordinate the data sources**. This may involve identifying common variables and ensuring that the data is in a consistent format.
- **Consider the future use of the data**. Is it only needed for this project, or could it be used for other projects in the future? If so, you should develop a strategy to present the data in a way that is accessible in other scenarios.
- In "Data to Decision" we will call this part **Data sources**.

3. Pre-process data

- **Data cleaning is important:** 80% of a data analyst's time is spent cleaning and organizing data, while only 20% is spent analyzing it. This shows that having clean data is essential for accurate analysis.
- **Data cleaning is the process of preparing raw data for analysis.** Data cleaning involves a variety of tasks, such as:
  - Identifying and removing incorrect data
  - Filling in missing data
  - Formatting the data in a consistent way
  - Standardizing the data
- **To start data cleaning, build tables to organize and catalog the data:** This will help you to better understand the data and identify any problems. Create a data dictionary: This is a table that catalogs each of your variables and translates them into what they mean to you in the context of this particular project.
- In "Data to Decision" we will call this **Data quality**

4. Analyse data

- **Once you've cleaned the data, you can start to analyze it using statistical models.** This involves building models to test the data and answer the business questions you identified earlier in the process.
- **There are three different ways to present your findings:**
  - **Descriptive information:** This is just the facts.
  - **Inferential information:** This includes the facts, plus an interpretation of what those facts indicate in the context of a particular project.
  - **Predictive information:** This is an inference based on facts and advice for further action based on your reasoning.
- Clarifying how the information will be most effectively presented will help you remain organized when it comes time to interpret the data.
- We will use graphics to condense data into interpretable information.
- In "Data to Decision" we will call this **Machine Learning**

5. Make decisions

- Did the analysis:
  - confirm suspicions?
  - shed new light on previously accepted "facts"?
  - find something deviating from expectations?
  - reveal flaws or weeknesses?
- In "Data to Decision" we might not make decisions ourselves, but we will perform **Deployment**

1.1.2 Data and decision making

- Simon Jackson defines the following Five flavours of decision making:
  - **Data-driven**: Commiting to a decision that will be taken from data before seing it. This means trusting that the DDDM pipeline is well designed and working.
  - **Data-inspired**: Making decision after looking at data or visualisations, believing that you see trends, patterns or anomalies. This means you have used data, but haven't made your choices on testable assumptions but more vague notions.
  - **Data-aspired**: Making decisions without data, but having a plan of collecting and analysing data at a later stage. This means you at least wish to collect data, but do not commit to changing your decision if data-based evidence comes at a later time.
  - **Data-ignorant**: Making decisions without considering data. This means you either do not have data available, haven't been able to leverage data or go against the evidence found by analysis.
  - **Data-tortured**: First making a decision, then looking through data until you find something that seems to support your case. This means you are biased by your own beliefs when looking at data, possibly grasping for straws. "If you torture the data long enough, it will confess to anything" ~ losely after Ronald Coase in the early 60's.

1.1.3 Resources

- DDDM: A Primer for beginners (Inspiration for this presentation)
- Five flavours of decision making
- Data-driven decision making: A step-by-step guide
- Making Data-driven Decisions in Business | Google Data Analytics Certificate (23m:41s)

1.2 Dashboards vs Reports

- Dashboards are often updated live
  - Mostly graphs, plots, etc.
  - Key performance indicators (KPIs)
  - Current trends, accumulation in minutes, hours, days, weeks, …
- Reports can be:

- Summaries of activities, economy, etc. up to a point
- Investigations into a subject
- Suggestive of future path or action
- Often printable, but digital reports can contain interactive graphics

Part II

Data Sources

CHAPTER

TWO

STATIC AND STREAMING DATA

The main topics here are:

- *Data formats*
- *Streaming data*

2.1 Data formats

Data come in many forms and formats, depending on source and function.

2.1.1 "Flat" text files

- .txt, .dat, .csv, .fasta, …
- Often with a fixed format
  - Comma separated columns
  - Tabulator separated columns
  - Fixed width columns
  - Headers, subheaders
  - Sections with keywords

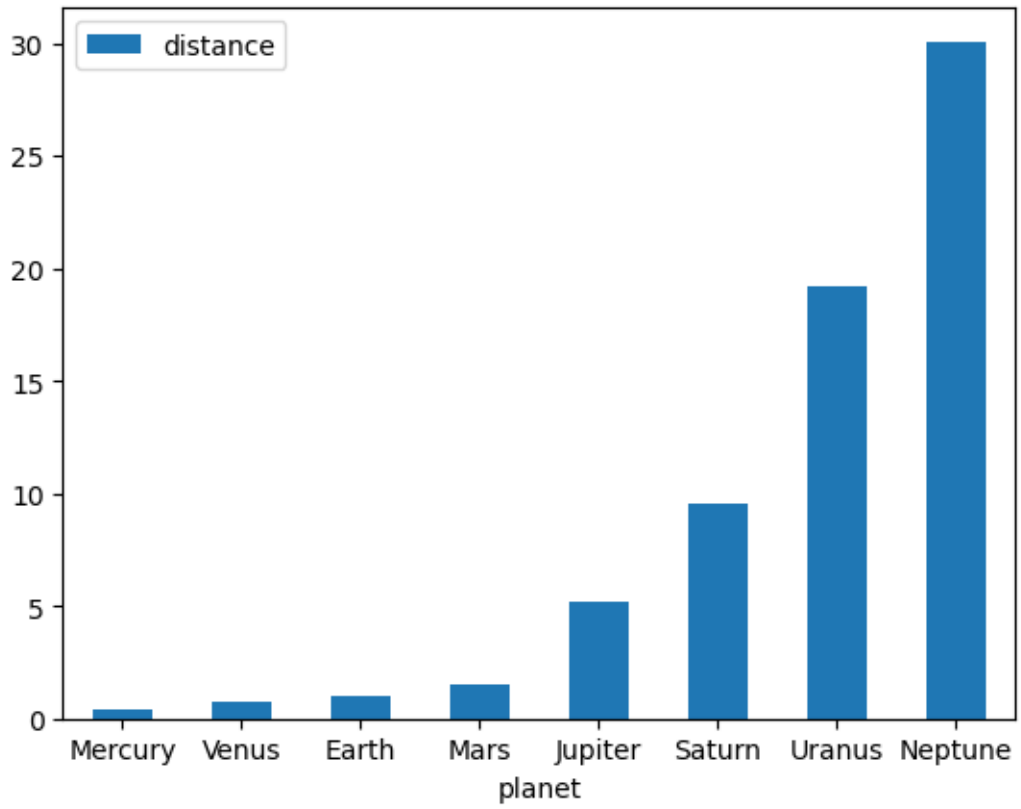
```
import pandas as pd
planets_DF = pd.read_csv('../../data/planets.csv')
planets_DF.head()
```

```
    planet  distance
0  Mercury  0.387 AU
1    Venus  0.723 AU
2    Earth  1.000 AU
3     Mars  1.524 AU
4  Jupiter  5.203 AU
```

```
# Using planets_DF as input, remove " AU" from the "distance" column and convert it␣
↪to a float
planets_DF['distance'] = planets_DF['distance'].str.replace(' AU', '').astype(float)
planets_DF.head()
```

```
    planet  distance
0  Mercury     0.387
1    Venus     0.723
2    Earth     1.000
3     Mars     1.524
4  Jupiter     5.203
```

```
# Plotting directly from Pandas
import matplotlib.pyplot as plt
planets_DF.plot.bar(x='planet', y='distance', rot=0)
plt.show()
```

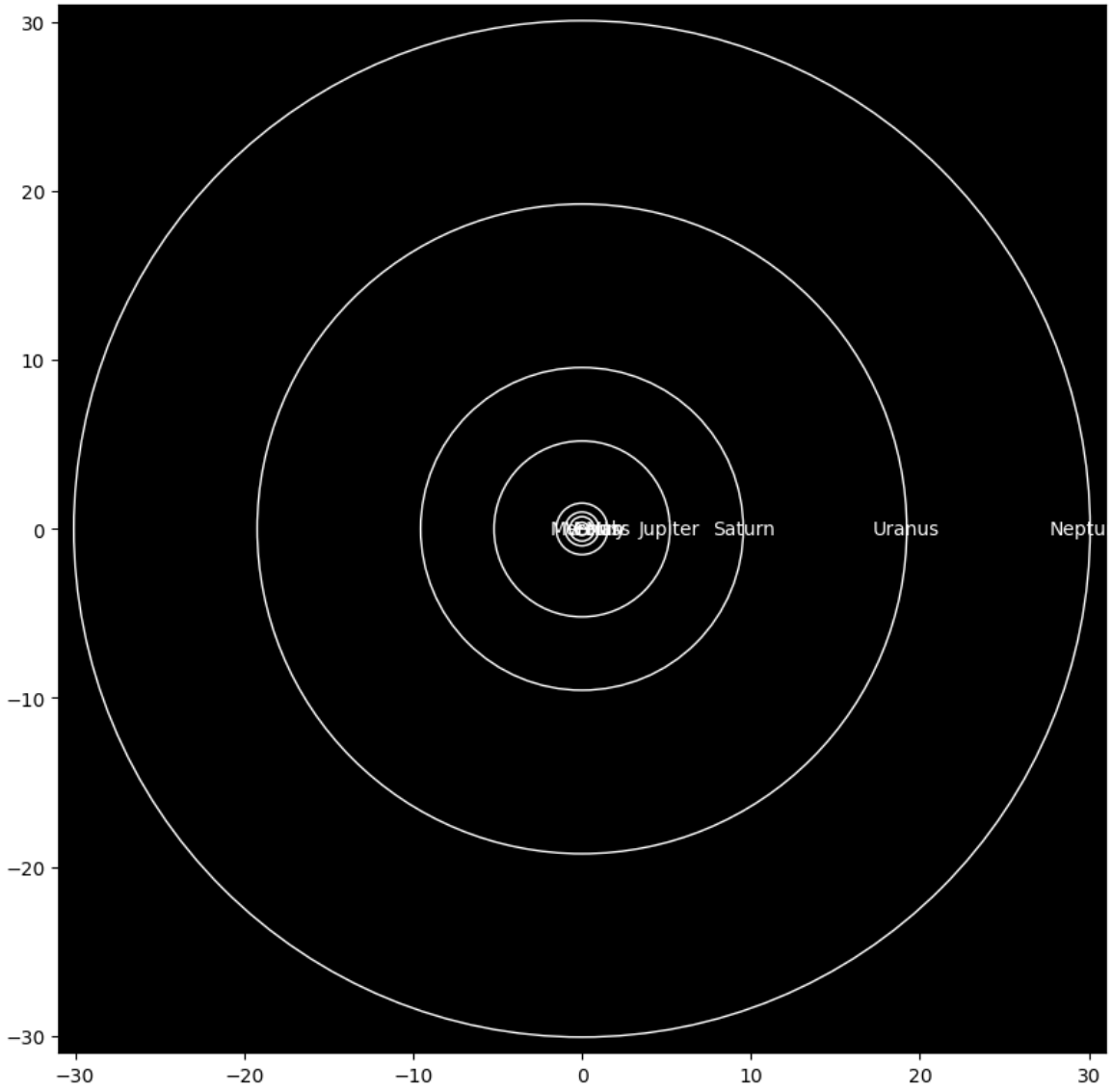



```
# Plot concentric ellipses for each planet where the size of the ellipse is␣
 ↪proportional to
# the planet's distance. Add planet names to the plot. Let the xlim and ylim extend␣
 ↪from -31 to 31.
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots(figsize=(10,10))
ax.set_xlim(-31, 31)
ax.set_ylim(-31, 31)
ax.set_aspect('equal')
ax.set_facecolor('black')
for i in range(len(planets_DF)):
    ax.add_artist(plt.Circle((0,0), planets_DF['distance'][i], color='white',␣
 ↪fill=False))
```

(continues on next page)

(continued from previous page)

```
    ax.text(planets_DF['distance'][i], 0, planets_DF['planet'][i], color='white', ha=
↪'center', va='center')
plt.show()
```



2.1.2 Excel files

2.1.3 Other proprietary formats

2.1.4 Media files

2.1.5 JSON

2.1.6 Database posts

2.1.7 HDF5

2.2 Streaming data

CHAPTER

THREE

DATABASES

This chapter will give a brief introduction to databases on a level sufficient for usage in the course. As many databases use SQL (Structured Query Language / "sequel") or a variation of this, we will also show SQL examples.

3.1 Main categories

- Relational databases/SQL, e.g., MySQL, Microsoft SQL, Oracle DB
- NoSQL, e.g., Cassandra, Spark
- Graph databases

This list is non-exhaustive both with regard to categories and examples.

3.1.1 Relational databases

- Tables with fixed attributes
  - Each column has a name and a storage type.
  - One column is defined as a key, only accepting unique values.
- Tables are connected (relations) in one of three ways:
  - **One-to-many**, one-to-one, and many-to-many

Key	Name	Type	…
+	id	int(11)	
	first_name	utf8	
	last_name	utf8	
	position	utf8	
	address	utf8	

1

Key	Name	Type	…
+	id	int(11)	
+	date	date	
	salary	float8	
	tax	float8	

3.2 MySQL

A true and tested free relational database.

```
# Make a connection object 'conn' for the MySQL database 'u492297623_student1' on␣
↪port 3306 at server 145.14.151.151 with username 'u492297623_stud1'
# and password 'Student1'.
import mysql.connector
PWD = open('../../../No_sync/password_khliland','r').read()

# Establishing a connection to the MySQL database
connection = mysql.connector.connect(
    host='145.14.151.151',
    port=3306,
    user='u492297623_stud1',
    password=PWD,
    database='u492297623_student1'
)

# Creating a cursor object to execute SQL queries
cursor = connection.cursor()
```

```
# Executing the SQL query to select all data from the 'students' table
query = "SELECT * FROM students;"
cursor.execute(query)

# Fetching all the rows from the result set
result = cursor.fetchall()

# Printing the retrieved data
for row in result:
    print(row)
```

```
(1, 'John', 'Industrial Economics')
(2, 'Sandra', 'Building Physics')
```

```
# Insert a new row into the first_test table with the values 3, 'Pete', 'Data Science'
query = "INSERT INTO students VALUES (3, 'Pete', 'Data Science');"
cursor.execute(query)
```

```
# Increment the main_key and insert new data
query = "INSERT INTO students SELECT COALESCE(MAX(main_key) + 1, 1), 'Anita',
↪'Paramedic' FROM students;"
cursor.execute(query)
```

```
# Closing the cursor and the database connection
cursor.close()
connection.close()
```

3.2.1 Exercise

Make a Python function that takes an SQL statement as input, opens a connection, executes the statement and closes the connection.

3.3 Cassandra

Assumptions:

- Docker installed on system
- "cassandra:latest" image installed in docker
- Python/Conda environment with python 3.8 (or newer*)

*As of 9 August 2023, the officially built cassandra-driver package was at version 3.25 and did not work well with Python 3.10 on Mac.
To install a newer version from the terminal you can:

```
pip install git+https://github.com/datastax/python-driver.git
pip install cql
```

Or use older Python:

```
conda create --name cassy38 python=3.8 jupyter scikit-learn
conda activate cassy38
pip install cassandra-driver cql
```

3.3.1 Spinning up a local cassandra instance in a terminal

`docker run --name my_cassandra cassandra:latest -p 9042:9042`

If this works for you, hourra! Otherwise, try the following:

- Open Docker Desktop
- Run the cassandra image with optional settings, opening 9042 port (left-hand side).

```
# Connecting to Cassandra
from cassandra.cluster import Cluster
cluster = Cluster(['localhost'], port=9042)
session = cluster.connect()
```

```
# Set up new keyspace (first time only)
#                                                name of keyspace                    ␣
 ↪ replication strategy           replication factor
session.execute("CREATE KEYSPACE IF NOT EXISTS my_first_keyspace WITH REPLICATION = {
 ↪'class' : 'SimpleStrategy', 'replication_factor' : 1 };")
```

```
<cassandra.cluster.ResultSet at 0x118922da0>
```

```
# Create a new table (first time only)
session.set_keyspace('my_first_keyspace')
session.execute("CREATE TABLE IF NOT EXISTS my_first_table (ind int PRIMARY KEY,␣
 ↪company text, model text);")
```

```
<cassandra.cluster.ResultSet at 0x1097c2890>
```

```
# Insert some data (ind is the primary key, must be unique)
session.execute("INSERT INTO my_first_table (ind, company, model) VALUES (1, 'Tesla',
 ↪'Model S');")
session.execute("INSERT INTO my_first_table (ind, company, model) VALUES (2, 'Tesla',
 ↪'Model 3');")
session.execute("INSERT INTO my_first_table (ind, company, model) VALUES (3, 'Polestar
 ↪', '3');")
```

```
<cassandra.cluster.ResultSet at 0x1097a4f70>
```

```
# Query the data
rows = session.execute("SELECT * FROM my_first_table;")
for i in rows:
    print(i)
```

```
Row(ind=1, company='Tesla', model='Model S')
Row(ind=2, company='Tesla', model='Model 3')
Row(ind=3, company='Polestar', model='3')
```

```
# More specific query
prepared_statement = session.prepare("SELECT * FROM my_first_table WHERE company=?;")
 ↪# <- will fail as company is not a key
teslas = session.execute(prepared_statement, ['Tesla'])
for i in teslas:
    print(i)
```

```
---------------------------------------------------------------------------
InvalidRequest                            Traceback (most recent call last)
Cell In [6], line 2
      1 # More specific query
----> 2 prepared_statement = session.prepare("SELECT * FROM my_first_table WHERE␣
 ↪company=?;") # <- will fail as company is not a key
      3 teslas = session.execute(prepared_statement, ['Tesla'])
      4 for i in teslas:

File ~/miniforge3/envs/tf_M1/lib/python3.10/site-packages/cassandra/cluster.
 ↪py:3088, in Session.prepare(self, query, custom_payload, keyspace)
   3086 try:
   3087     future.send_request()
-> 3088     response = future.result().one()
   3089 except Exception:
   3090     log.exception("Error preparing query:")

File ~/miniforge3/envs/tf_M1/lib/python3.10/site-packages/cassandra/cluster.
 ↪py:4920, in ResponseFuture.result(self)
```

(continues on next page)

(continued from previous page)

```
   4918     return ResultSet(self, self._final_result)
   4919 else:
-> 4920     raise self._final_exception

InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot␣
 ↪execute this query as it might involve data filtering and thus may have␣
 ↪unpredictable performance. If you want to execute this query despite the␣
 ↪performance unpredictability, use ALLOW FILTERING"
```

3.3.2 Cassandra filtering

Cassandra is inherently a distributed production database. Selecting as above may require downloading all data from a node, then filtering based on the WHERE part (only PRIMARY KEYs are centrally known). Solutions:

- If the table is small or most of the data will satisfy the query, add `ALLOW FILTERING` at the end of the query (not recommended if not known).
- Or make sure the WHERE clause points to one of the keys (see below).

```
# Create a new table (observe keys)
session.execute("CREATE TABLE IF NOT EXISTS car_table (company text, model text,␣
 ↪PRIMARY KEY(company, model));")
```

```
<cassandra.cluster.ResultSet at 0x118925090>
```

```
# Insert some data (combination of company and model must be unique)
session.execute("INSERT INTO car_table (company, model) VALUES ('Tesla', 'Model S');")
session.execute("INSERT INTO car_table (company, model) VALUES ('Tesla', 'Model 3');")
session.execute("INSERT INTO car_table (company, model) VALUES ('Polestar', '3');")
```

```
<cassandra.cluster.ResultSet at 0x11957c430>
```

```
# More specific query now works
prepared_statement = session.prepare("SELECT * FROM car_table WHERE company=?;")
teslas = session.execute(prepared_statement, ['Tesla'])
for i in teslas:
    print(i)
```

```
Row(company='Tesla', model='Model 3')
Row(company='Tesla', model='Model S')
```

3.3.3 JSON in Cassandra

```
# Imports
import datetime as dt
import requests
import json
```

Read previously saved JSON file forecast.json to memory

```
with open('../APIs/downloads/forecast.json', 'r') as f:
    forecast = json.load(f)
```

```
# Inspect JSON file
forecast.__str__()
```

```
"{'cod': '200', 'message': 0, 'cnt': 40, 'list': [{'dt': 1692187200, 'main': {'temp
↪': 292.16, 'feels_like': 292.31, 'temp_min': 292.16, 'temp_max': 292.41,
↪'pressure': 1015, 'sea_level': 1015, 'grnd_level': 998, 'humidity': 84, 'temp_kf
↪': -0.25}, 'weather': [{'id': 500, 'main': 'Rain', 'description': 'light rain',
↪'icon': '10d'}], 'clouds': {'all': 92}, 'wind': {'speed': 1.17, 'deg': 299, 'gust
↪': 1.37}, 'visibility': 10000, 'pop': 0.43, 'rain': {'3h': 0.46}, 'sys': {'pod':
↪'d'}, 'dt_txt': '2023-08-16 12:00:00'}, {'dt': 1692198000, 'main': {'temp': 292.
↪84, 'feels_like': 292.85, 'temp_min': 292.84, 'temp_max': 294.19, 'pressure':␣
↪1015, 'sea_level': 1015, 'grnd_level': 999, 'humidity': 76, 'temp_kf': -1.35},
↪'weather': [{'id': 500, 'main': 'Rain', 'description': 'light rain', 'icon': '10d
↪'}], 'clouds': {'all': 73}, 'wind': {'speed': 0.68, 'deg': 328, 'gust': 1.44},
↪'visibility': 10000, 'pop': 0.49, 'rain': {'3h': 0.15}, 'sys': {'pod': 'd'}, 'dt_
↪txt': '2023-08-16 15:00:00'}, {'dt': 1692208800, 'main': {'temp': 291.21, 'feels_
↪like': 291.21, 'temp_min': 290.73, 'temp_max': 291.21, 'pressure': 1016, 'sea_
↪level': 1016, 'grnd_level': 1000, 'humidity': 82, 'temp_kf': 0.48}, 'weather': [{
↪'id': 500, 'main': 'Rain', 'description': 'light rain', 'icon': '10d'}], 'clouds
↪': {'all': 47}, 'wind': {'speed': 0.4, 'deg': 303, 'gust': 1.4}, 'visibility':␣
↪10000, 'pop': 0.68, 'rain': {'3h': 0.52}, 'sys': {'pod': 'd'}, 'dt_txt': '2023-
↪08-16 18:00:00'}, {'dt': 1692219600, 'main': {'temp': 287.1, 'feels_like': 287.
↪03, 'temp_min': 287.1, 'temp_max': 287.1, 'pressure': 1019, 'sea_level': 1019,
↪'grnd_level': 1002, 'humidity': 95, 'temp_kf': 0}, 'weather': [{'id': 500, 'main
↪': 'Rain', 'description': 'light rain', 'icon': '10n'}], 'clouds': {'all': 15},
↪'wind': {'speed': 1.55, 'deg': 353, 'gust': 2.35}, 'visibility': 10000, 'pop': 0.
↪21, 'rain': {'3h': 0.26}, 'sys': {'pod': 'n'}, 'dt_txt': '2023-08-16 21:00:00'},
↪{'dt': 1692230400, 'main': {'temp': 285.77, 'feels_like': 285.54, 'temp_min':␣
↪285.77, 'temp_max': 285.77, 'pressure': 1021, 'sea_level': 1021, 'grnd_level':␣
↪1004, 'humidity': 94, 'temp_kf': 0}, 'weather': [{'id': 801, 'main': 'Clouds',
↪'description': 'few clouds', 'icon': '02n'}], 'clouds': {'all': 13}, 'wind': {
↪'speed': 1.99, 'deg': 358, 'gust': 4.84}, 'visibility': 10000, 'pop': 0.05, 'sys
↪': {'pod': 'n'}, 'dt_txt': '2023-08-17 00:00:00'}, {'dt': 1692241200, 'main': {
↪'temp': 284.36, 'feels_like': 283.99, 'temp_min': 284.36, 'temp_max': 284.36,
↪'pressure': 1022, 'sea_level': 1022, 'grnd_level': 1005, 'humidity': 94, 'temp_kf
↪': 0}, 'weather': [{'id': 800, 'main': 'Clear', 'description': 'clear sky', 'icon
↪': '01n'}], 'clouds': {'all': 8}, 'wind': {'speed': 2.18, 'deg': 359, 'gust': 5.
↪76}, 'visibility': 10000, 'pop': 0, 'sys': {'pod': 'n'}, 'dt_txt': '2023-08-17␣
↪03:00:00'}, {'dt': 1692252000, 'main': {'temp': 286.68, 'feels_like': 286.28,
↪'temp_min': 286.68, 'temp_max': 286.68, 'pressure': 1024, 'sea_level': 1024,
↪'grnd_level': 1007, 'humidity': 84, 'temp_kf': 0}, 'weather': [{'id': 800, 'main
↪': 'Clear', 'description': 'clear sky', 'icon': '01d'}], 'clouds': {'all': 7},
↪'wind': {'speed': 2.19, 'deg': 19, 'gust': 4.9}, 'visibility': 10000, 'pop': 0,
↪'sys': {'pod': 'd'}, 'dt_txt': '2023-08-17 06:00:00'}, {'dt': 1692262800, 'main
↪': {'temp': 292.07, 'feels_like': 291.64, 'temp_min': 292.07, 'temp_max': 292.07,
↪ 'pressure': 1024, 'sea_level': 1024, 'grnd_level': 1008, 'humidity': 62, 'temp_
↪kf': 0}, 'weather': [{'id': 800, 'main': 'Clear', 'description': 'clear sky',
↪'icon': '01d'}], 'clouds': {'all': 3}, 'wind': {'speed': 2.61, 'deg': 43, 'gust
↪': 3.94}, 'visibility': 10000, 'pop': 0, 'sys': {'pod': 'd'}, 'dt_txt': '2023-08-
```

(continues on next page)

(continued from previous page)

3.3.4 Raw JSON

A simple, but not very efficient way of storing JSON data is to treat it as a text and save it directly to the database. More efficient, with regard to transfer, is to compress the JSON data to a blob first.

```
session.execute("DROP TABLE IF EXISTS my_first_keyspace.forecast_table;")
```

```
<cassandra.cluster.ResultSet at 0x11955fbb0>
```

Create a new table which treats the whole JSON as a blob, using the city id and the first dt as keys

```
session.set_keyspace('my_first_keyspace')
session.execute("CREATE TABLE IF NOT EXISTS forecast_table (city_id int, dt int,␣
↪forecast blob, PRIMARY KEY(city_id, dt));")
```

```
<cassandra.cluster.ResultSet at 0x11a42ea10>
```

Insert the forecast data into the table as text

```
session.execute("INSERT INTO forecast_table (city_id, dt, forecast) VALUES (%s, %s,␣
↪textAsBlob(%s));", (forecast['city']['id'], forecast['list'][0]['dt'], forecast.__
↪str__()))
```

```
<cassandra.cluster.ResultSet at 0x1097a4bb0>
```

```
# Query the data
forecast_rows = session.execute("SELECT * FROM forecast_table;")
print(forecast_rows.one()) # <- only one row
```

```
Row(city_id=3139081, dt=1692187200, forecast=b"{'cod': '200', 'message': 0, 'cnt':␣
↪40, 'list': [{'dt': 1692187200, 'main': {'temp': 292.16, 'feels_like': 292.31,
↪'temp_min': 292.16, 'temp_max': 292.41, 'pressure': 1015, 'sea_level': 1015,
↪'grnd_level': 998, 'humidity': 84, 'temp_kf': -0.25}, 'weather': [{'id': 500,
↪'main': 'Rain', 'description': 'light rain', 'icon': '10d'}], 'clouds': {'all':␣
↪92}, 'wind': {'speed': 1.17, 'deg': 299, 'gust': 1.37}, 'visibility': 10000, 'pop
↪': 0.43, 'rain': {'3h': 0.46}, 'sys': {'pod': 'd'}, 'dt_txt': '2023-08-16␣
↪12:00:00'}, {'dt': 1692198000, 'main': {'temp': 292.84, 'feels_like': 292.85,
↪'temp_min': 292.84, 'temp_max': 294.19, 'pressure': 1015, 'sea_level': 1015,
↪'grnd_level': 999, 'humidity': 76, 'temp_kf': -1.35}, 'weather': [{'id': 500,
↪'main': 'Rain', 'description': 'light rain', 'icon': '10d'}], 'clouds': {'all':␣
↪73}, 'wind': {'speed': 0.68, 'deg': 328, 'gust': 1.44}, 'visibility': 10000, 'pop
↪': 0.49, 'rain': {'3h': 0.15}, 'sys': {'pod': 'd'}, 'dt_txt': '2023-08-16␣
↪15:00:00'}, {'dt': 1692208800, 'main': {'temp': 291.21, 'feels_like': 291.21,
↪'temp_min': 290.73, 'temp_max': 291.21, 'pressure': 1016, 'sea_level': 1016,
↪'grnd_level': 1000, 'humidity': 82, 'temp_kf': 0.48}, 'weather': [{'id': 500,
↪'main': 'Rain', 'description': 'light rain', 'icon': '10d'}], 'clouds': {'all':␣
↪47}, 'wind': {'speed': 0.4, 'deg': 303, 'gust': 1.4}, 'visibility': 10000, 'pop
↪': 0.68, 'rain': {'3h': 0.52}, 'sys': {'pod': 'd'}, 'dt_txt': '2023-08-16␣
↪18:00:00'}, {'dt': 1692219600, 'main': {'temp': 287.1, 'feels_like': 287.03,
↪'temp_min': 287.1, 'temp_max': 287.1, 'pressure': 1019, 'sea_level': 1019, 'grnd_
↪level': 1002, 'humidity': 95, 'temp_kf': 0}, 'weather': [{'id': 500, 'main':
↪'Rain', 'description': 'light rain', 'icon': '10n'}], 'clouds': {'all': 15},
```

(continues on next page)

(continued from previous page)

CHAPTER

FOUR

APIS

4.1 REST API

- REST stands for Representational State Transfer.
- Architectural style for designing web services.
- REST services are stateless, which means that they do not maintain any state between requests. This makes them scalable and reliable.
- For us they are mainly interfaces for information retrieval.

4.1.1 Accessing REST

- REST services are based on the HTTP protocol, and they use a set of well-defined verbs to manipulate resources.
- Some are open and free, some need an API key (free or subscription)
- The four main verbs in REST are:
  - GET: Retrieve a resource.
  - POST: Create a resource.
  - PUT: Update a resource.
  - DELETE: Delete a resource.

Retrieving data (GET)

- A simple example without an API key (Strømpris API):

```
import requests
# Power prices in the NO5 zone on a particular date
url = "https://www.hvakosterstrommen.no/api/v1/prices/2023/08-09_NO5.json"
response = requests.get(url)
print(response.json())
```

```
[{'NOK_per_kWh': 0.00546, 'EUR_per_kWh': 0.00049, 'EXR': 11.1485, 'time_start':
↪'2023-08-09T00:00:00+02:00', 'time_end': '2023-08-09T01:00:00+02:00'}, {'NOK_per_
↪kWh': -0.01249, 'EUR_per_kWh': -0.00112, 'EXR': 11.1485, 'time_start': '2023-08-
↪09T01:00:00+02:00', 'time_end': '2023-08-09T02:00:00+02:00'}, {'NOK_per_kWh': -0.
↪01282, 'EUR_per_kWh': -0.00115, 'EXR': 11.1485, 'time_start': '2023-08-
↪09T02:00:00+02:00', 'time_end': '2023-08-09T03:00:00+02:00'}, {'NOK_per_kWh': -0.
↪01304, 'EUR_per_kWh': -0.00117, 'EXR': 11.1485, 'time_start': '2023-08-
↪09T03:00:00+02:00', 'time_end': '2023-08-09T04:00:00+02:00'}, {'NOK_per_kWh': -0.
↪01316, 'EUR_per_kWh': -0.00118, 'EXR': 11.1485, 'time_start': '2023-08-
↪09T04:00:00+02:00', 'time_end': '2023-08-09T05:00:00+02:00'}, {'NOK_per_kWh': -0.
↪01304, 'EUR_per_kWh': -0.00117, 'EXR': 11.1485, 'time_start': '2023-08-
↪09T05:00:00+02:00', 'time_end': '2023-08-09T06:00:00+02:00'}, {'NOK_per_kWh': -0.
↪00033, 'EUR_per_kWh': -3e-05, 'EXR': 11.1485, 'time_start': '2023-08-
```

(continues on next page)

(continued from previous page)

Creating data (POST)

- For this example we assume there is an API server running locally.
- We rely on the Flask framework (see flask_API.py) in the current folder.
  - Simple POST, GET, "UPDATE", DELETE

```
# POST data (stored locally in a dictionary)
id = 'test'
data = {'key': 'value'}
response = requests.post('http://localhost:8000/api/post/{}'.format(id), json=data)
print(response.json())
```

```
{'key': 'value'}
```

```
# GET data
id = 'test'
response = requests.get('http://localhost:8000/api/get/{}'.format(id))
print(response.json())
# You can also test this in a browser since this is an HTTP based (and we use no␣
↪passwords here).
```

```
{'key': 'value'}
```

Change data (UPDATE)

- This command can mostly be exchanged with POST.
- The Flask framework does not have a separate UPDATE (see implementation in flask_API.py).

```
# UPDATE data
id = 'test'
data = {'key': 'new value'}
response = requests.post('http://localhost:8000/api/update/{}'.format(id), json=data)
print(response.json())
```

```
{'key': 'new value'}
```

Remove data (DELETE)

- Let's remove some data and try to read it again.

```
# DELETE data
id = 'test'
response = requests.delete('http://localhost:8000/api/delete/{}'.format(id))
print(response.json())
```

```
{'key': 'new value'}
```

```
# GET data again
id = 'test'
response = requests.get('http://localhost:8000/api/get/{}'.format(id))
print(response.json())
```

```
---------------------------------------------------------------------------
JSONDecodeError                           Traceback (most recent call last)
File ~/miniforge3/envs/tf_M1/lib/python3.10/site-packages/requests/models.py:971,␣
 ↪in Response.json(self, **kwargs)
    970 try:
--> 971     return complexjson.loads(self.text, **kwargs)
    972 except JSONDecodeError as e:
    973     # Catch JSON-related errors and raise as requests.JSONDecodeError
    974     # This aliases json.JSONDecodeError and simplejson.JSONDecodeError

File ~/miniforge3/envs/tf_M1/lib/python3.10/json/__init__.py:346, in loads(s, cls,␣
 ↪object_hook, parse_float, parse_int, parse_constant, object_pairs_hook, **kw)
    343 if (cls is None and object_hook is None and
    344         parse_int is None and parse_float is None and
    345         parse_constant is None and object_pairs_hook is None and not kw):
--> 346     return _default_decoder.decode(s)
    347 if cls is None:

File ~/miniforge3/envs/tf_M1/lib/python3.10/json/decoder.py:337, in JSONDecoder.
 ↪decode(self, s, _w)
    333 """Return the Python representation of ``s`` (a ``str`` instance
    334 containing a JSON document).
    335
    336 """
--> 337 obj, end = self.raw_decode(s, idx=_w(s, 0).end())
    338 end = _w(s, end).end()

File ~/miniforge3/envs/tf_M1/lib/python3.10/json/decoder.py:355, in JSONDecoder.
 ↪raw_decode(self, s, idx)
    354 except StopIteration as err:
--> 355     raise JSONDecodeError("Expecting value", s, err.value) from None
    356 return obj, end

JSONDecodeError: Expecting value: line 1 column 1 (char 0)

During handling of the above exception, another exception occurred:

JSONDecodeError                           Traceback (most recent call last)
Cell In [6], line 4
```

(continues on next page)

(continued from previous page)

```
      2 id = 'test'
      3 response = requests.get('http://localhost:8000/api/get/{}'.format(id))
----> 4 print(response.json())

File ~/miniforge3/envs/tf_M1/lib/python3.10/site-packages/requests/models.py:975,␣
 ↪in Response.json(self, **kwargs)
    971     return complexjson.loads(self.text, **kwargs)
    972 except JSONDecodeError as e:
    973     # Catch JSON-related errors and raise as requests.JSONDecodeError
    974     # This aliases json.JSONDecodeError and simplejson.JSONDecodeError
--> 975     raise RequestsJSONDecodeError(e.msg, e.doc, e.pos)

JSONDecodeError: Expecting value: line 1 column 1 (char 0)
```

4.1.2 Exercise

- Look at flask_API.py.
- Add "try-except" to GET to return an empty JSON when an 'id' does not exist.
- Are there other potential sources of error here?

4.1.3 Resources

- Wikipedia: REST
- Flask local HTTP server in Python.
- Strømpris API (only in Norwegian)
- YouTube: What is a REST API? (9m:11s)
- YouTube: REST API Crash Course - Introduction + Full Python API Tutorial (52m:49s)

4.2 Demonstration of API using OpenWeather.org

This demonstration is heavily inspired by NeuralNine's video.
The VS Code extensino *JSON viewer* is recommended for viewing downloaded JSON content.

```
# Imports
import datetime as dt
import requests
import json
```

4.2.1 Common definitions to use for all requests

Current weather

```
BASE_URL = "http://api.openweathermap.org/data/2.5/weather?"
API_KEY = open('../../../No_sync/api_key_OpenWeather','r').read()
CITY = "Ski"

url = BASE_URL + "q=" + CITY + "&appid=" + API_KEY
```

4.2.2 Request current weather in chosen city

```
response = requests.get(url).json()
# print(response)
```

```
# Write JSON to file for viewing
with open('downloads/weather.json', 'w') as f:
    json.dump(response, f, indent=4)
```

4.2.3 Conversion functions

```
# Kelvin to Celsius
def kelvin_to_celsius(temp):
    return temp - 273.15

# Meters per second to knots
def mps_to_knots(speed):
    return speed * 1.943844
```

4.2.4 Print some weather properties

```
# Current temperature
temp_kelvin = response['main']['temp']
temp_celsius = kelvin_to_celsius(temp_kelvin)
print(f"The current temperature in {CITY} is {temp_celsius:.2f}°C")
```

```
The current temperature in Ski is 19.01°C
```

```
# Sunrise and sunset today in local time
sunrise = dt.datetime.fromtimestamp(response['sys']['sunrise'])
sunset = dt.datetime.fromtimestamp(response['sys']['sunset'])
print(f"Sunrise today is at {sunrise:%H:%M} and sunset is at {sunset:%H:%M}")
```

```
Sunrise today is at 05:33 and sunset is at 21:08
```

4.2.5 Common definitions to use for all requests

Forecasted weather

```
BASE_URL = "http://api.openweathermap.org/data/2.5/forecast?"
CITY = "Ski"

urlF = BASE_URL + "q=" + CITY + "&appid=" + API_KEY
```

4.2.6 Request current weather in chosen city

```
responseF = requests.get(urlF).json()
# print(responseF)
```

```
# Write JSON to file for viewing
with open('downloads/forecast.json', 'w') as f:
    json.dump(responseF, f, indent=4)
```

When and what?

Check contents and time stamps

```
# Content of responseF
responseF.keys()
```

```
dict_keys(['cod', 'message', 'cnt', 'list', 'city'])
```

```
# Number of forecasts
print(len(responseF["list"]))
```

```
40
```

```
# Print forecast times
for forecast in responseF["list"]:
    print(forecast["dt_txt"])
```

```
2023-08-16 12:00:00
2023-08-16 15:00:00
2023-08-16 18:00:00
2023-08-16 21:00:00
2023-08-17 00:00:00
2023-08-17 03:00:00
2023-08-17 06:00:00
2023-08-17 09:00:00
2023-08-17 12:00:00
2023-08-17 15:00:00
2023-08-17 18:00:00
```

(continues on next page)

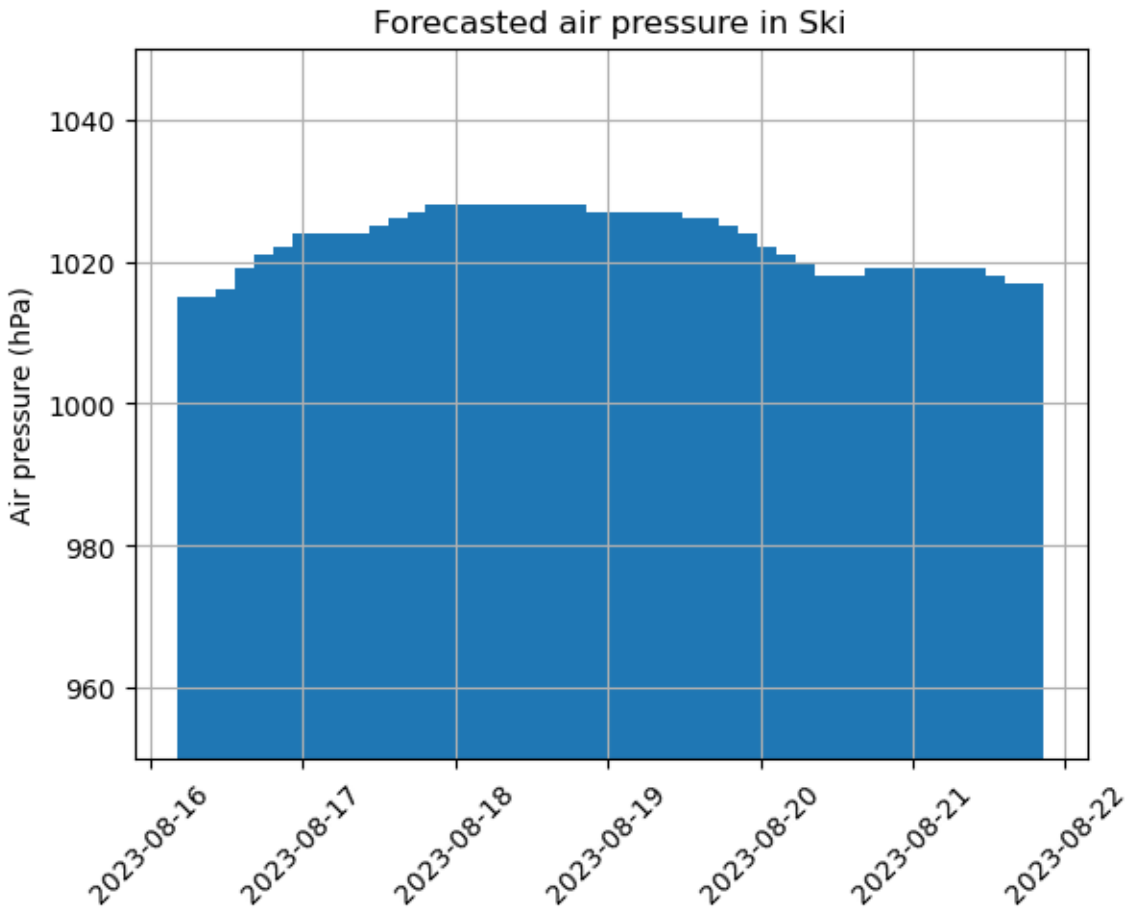
(continued from previous page)

```
2023-08-17 21:00:00
2023-08-18 00:00:00
2023-08-18 03:00:00
2023-08-18 06:00:00
2023-08-18 09:00:00
2023-08-18 12:00:00
2023-08-18 15:00:00
2023-08-18 18:00:00
2023-08-18 21:00:00
2023-08-19 00:00:00
2023-08-19 03:00:00
2023-08-19 06:00:00
2023-08-19 09:00:00
2023-08-19 12:00:00
2023-08-19 15:00:00
2023-08-19 18:00:00
2023-08-19 21:00:00
2023-08-20 00:00:00
2023-08-20 03:00:00
2023-08-20 06:00:00
2023-08-20 09:00:00
2023-08-20 12:00:00
2023-08-20 15:00:00
2023-08-20 18:00:00
2023-08-20 21:00:00
2023-08-21 00:00:00
2023-08-21 03:00:00
2023-08-21 06:00:00
2023-08-21 09:00:00
```

4.2.7 Make plots of omnipresent measurements and events

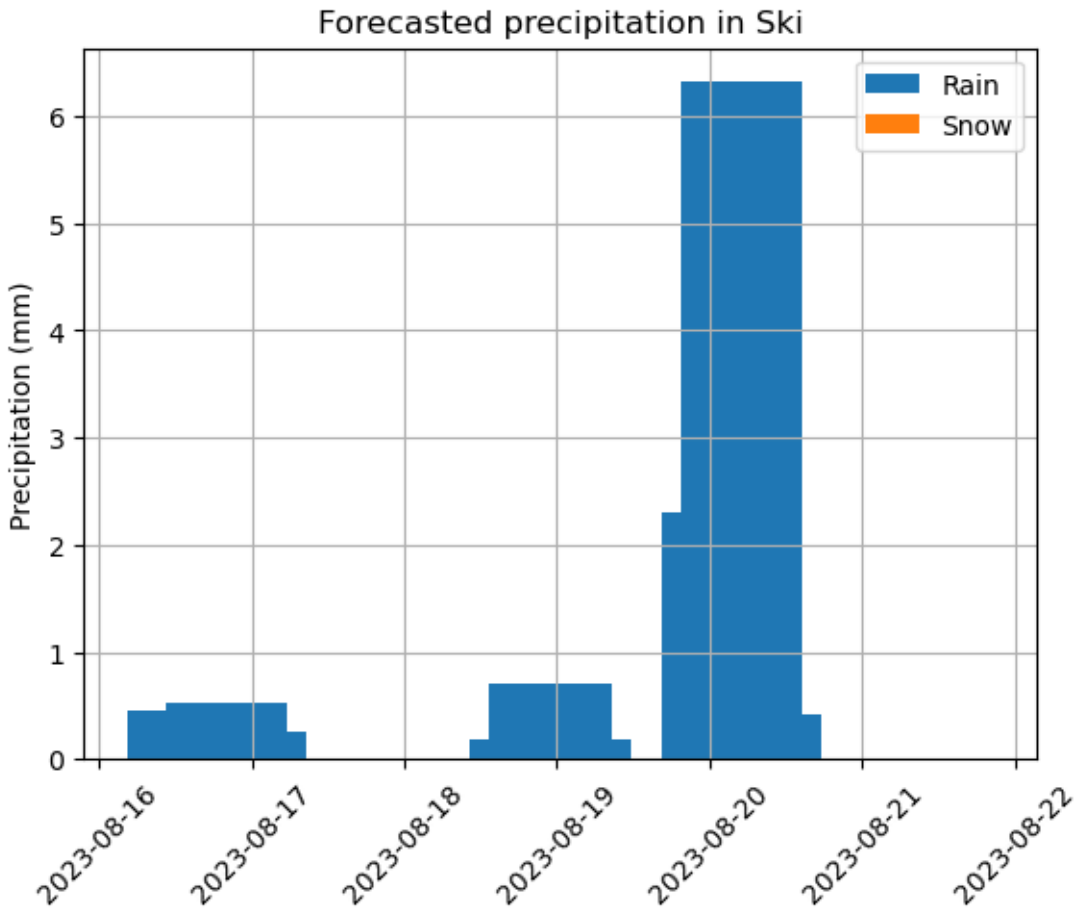
```
# Air pressure per period
pressures = []
timestamps = []
for forecast in responseF["list"]:
    pressures.append(forecast["main"]["pressure"])
    timestamps.append(dt.datetime.fromtimestamp(forecast["dt"]))
```

```
import matplotlib.pyplot as plt
plt.bar(timestamps, pressures)
plt.xticks(rotation=45)
plt.ylim(950, 1050)
plt.grid()
plt.ylabel("Air pressure (hPa)")
plt.title(f"Forecasted air pressure in {CITY}")
plt.show()
```




```
rain = []
snow = []
for forecast in responseF["list"]:
    try: # Check if rain is present in forecast
        rain.append(forecast["rain"]["3h"])
    except KeyError:
        rain.append(0)
    try: # Check if snow is present in forecast
        snow.append(forecast["snow"]["3h"])
    except KeyError:
        snow.append(0)
```

```
# Stacked bar chart with rain and snow
plt.bar(timestamps, rain, label="Rain")
plt.bar(timestamps, snow, label="Snow")
plt.xticks(rotation=45)
plt.grid()
plt.ylabel("Precipitation (mm)")
plt.title(f"Forecasted precipitation in {CITY}")
plt.legend()
plt.show()
```


Forecasted precipitation in Ski
Rain
Snow
Precipitation (mm)
0
1
2
3
4
5
6
2023-08-16
2023-08-17
2023-08-18
2023-08-19
2023-08-20
2023-08-21
2023-08-22

Part III

Data quality

CHAPTER

FIVE

TBA

Part IV

Machine Learning

CHAPTER

SIX

TBA

6.1 Resources

- An Introduction to Statistical Learning - with Applications in Python [Gareth et al. 2023]

Part V

Deployment

CHAPTER

SEVEN

TBA