Contents

1 Making a torch dataloader **2**
1.1 Dataloader wishlist/description . . . 2
1.2 Resulting data model . . . 2

2 Pipeline **3**
2.1 Actual implementation details of the pipeline . . . 3
2.1.1 Configuration files . . . 4
2.2 Performance of the pipeline . . . 4
2.3 Tools used in the pipeline . . . 5
2.4 Difficulties . . . 5

3 Airflow **5**

4 Appendix **7**
4.1 Running the pipeline instructions . . . 7
4.2 Table of configuration files . . . 7
4.3 Wrangling functions . . . 8
4.3.1 process_nc_file . . . 8
4.3.2 insert_nc_filerows . . . 9
4.4 Schema of the Measurement table: . . . 10

1 Making a torch dataloader

I will improve the quality of the data by improving how easy the data will be to load. To do this I want to create a torch dataloader.

When I was taught machine learning we always had to use torch data loaders. Sometimes you were lucky and one already exisited but most of the time you were unlucky. I remember the processes of creating your own data loader to always be quite annoying. Presenting the data scientist with a dataloader will make it the easiest to use the data to make models. Thus, I think I would make the data scientist the happiest by providing a dataloader as the main way to load the data from knmi.

1.1 Dataloader wishlist/description

In the constructor of this dataloader the data scientist should be able to say which keys they want, from which stations and from which date range (from, until). The dataloader will then go download and cache the datasets from that date range and the keys you downloaded.

The resulting shape of the data for the data scientist will be (n, s, k) where

- n is based on how many readings there are in the specified date range
- s is the number of valid stations asked for
- k is the number of different keys specified

The label of each row in the pytorch dataset will be the time the value was recorded. The labels dataset will have n rows.

Effectivity the dataloader will provide the data scientist with the ability to only load the part of the dataset they wish for. This means that we might only send a subset of the entire dataset to the data scientist.

1.2 Resulting data model

Because I want the data scientist to query the dataset using the data loader I think a relational data model is the best fit. Using a relation database I can easily query the central database for exactly the data that the data scientist requested. A relational database additionally provides an easy way to combine the data from multiple nc files because it can just be added to the table. Each row in the table holds the measurements of a single station at a single point in time. The database schema will only have a single table. The schema of this table can be found in the appendix. I do not need more than one table. You can already filter the columns you want in the select statement. Having more tables will only needlessly increase complexity.

When the data scientist loads the data using the torch dataloader the server shall answer the query by querying this central database. Then once the data has been collected on the server it will be sent back to the data scientist in the form of a parquet file.

To create this relational database that will facilitate the experience with the dataloader I will create a pipeline. This pipeline will add the new data from the knmi stream (in the form of nc files) to a central relational database hosted somewhere. The rest of this document will only discuss creating the pipeline and not the actual torch dataloader. It makes more sense to discuss the actual data loader in assignment 4 which is about actually serving the data.

2 Pipeline

Remember, these nc files are basically measurements from s stations of (currently) 94 variables at a single point in time. To be able to query the data later, I want to save the measurements of each station at a point in time as a separate row in the database. This means I have to transform the nc file into a list of table rows that can be inserted into the database. This is the job of the pipeline.

The pipeline is called upon when an item actually comes in from the knmi stream. The pipeline consists of the following steps:

1. Download and save the nc file by sending the required requests.
2. Load the nc file
3. Get the time value (there is only 1 per nc file)
4. Create a list of rows to insert into the database from the nc file
5. Create each row by iterating over the column names and for each station:
    1. Deal with the fact that the vectors are of shape (69,1) instead of just (69,)
    2. Deal with the masked values and add missing values as None to the row to insert
    3. Do not touch string as these vectors are actually (69,) so just put them in the row to insert
6. Insert the list of rows into a relational database

It might look a bit simple now, but I spend a lot of time thinking about this, exploring the data and considering different options to finally get to this simplicity. For instance, I also considered vector databases or making a larger h5 file, but I finally settled on a relational model.

2.1 Actual implementation details of the pipeline

To create this pipeline I created two functions. The first function `process_nc_file` with the type $Path \to [[str \mid int \mid float]]$. This function takes a file path and executes the first five steps. Then the

second function, `insert_nc_filerows` with type $[[str \mid int \mid float]] \to ()$ inserts the result from `process_nc_file` into the relational database. In doing so it completes step 6 of the pipeline. The two functions can be found in the appendix or in the git repository.

I wrote the functions in such a way that they can deal with a dynamic shape of the nc file. The code can for instance handle if the number of stations changes.

When knmi decides to add more data to their nc files the wrangler also should not break. But you do have to add the new nc database name to the `nc_keys_to_save` tuple in wrangle.py and to the database schema to get the pipeline to actually save this new data.

2.1.1 Configuration files

Making things data driving is good because it makes the code much more resilient to change. With data driven I mean read arbitrary options from a config file instead of from the code. During the wrangling step of the project I got the idea to make `nc_keys_to_save` data driven. And, if I was going to be doing that, I might as well make the configuration for the stream knmi connection data driven as well. This allows sharing the code without sharing the secret tokens because you have to add those yourself.

To this end I created two config files `auth.toml` and `config.toml`. The idea is that `auth.toml` holds secret data which should not be commited to source control while `config.toml` only holds public knowledge data which can be commited to source control. A table with the keys of the configuration can be found in the appendix. However, in the end I decided against putting `nc_keys_to_save` into a configuration file. I decided against this because I dynamically generate the sql insert statement from the key names. Do not worry I still use prepared statements, but I construct the number of `?` holes and column names dynamically from the `nc_keys_to_save` list. For this reason I judged it better for security if it is a bit more difficult to edit the column names. This only leaves the knmi stream configuration to be data driven.

2.2 Performance of the pipeline

A single nc file is about 419145 bytes on average. It takes my pipeline on average 1.43 seconds to process and nc file and then on average 2.54 seconds to save the rows into the database. These are just simple benchmarks using python `time.time` and this is just running on my laptop.

Clearly, the processing of the nc file could be sped up significantly as this is a naive python script. But I do not think that this is necessary as a nc file comes in only every 10 minutes.

2.3 Tools used in the pipeline

I wrote the two wrangle functions in python 3.10 as python is a langauge that is easy to write with many dependencies. I used poetry to manage the python environment and dependencies.

For the database I decided to use duckdb. I wanted to try out this technology after the lectures. After trying it I can say that I had a good experience with it. I love the documentation it is really well written.

I also used the netCDF4 library to actually load the nc files. I use paho-mqtt to listen to the data stream from knmi.

2.4 Difficulties

Everything went smoothly except for one thing. There are two fields within the nc files that have an − in them. Duckdb does not support this directly. I had two options. Either change to _ or you quote the column names like `"W10−10"` and `"ww−10"`. I decided for quoting as changing the name seems like a bad idea because no one using the dataset will expect that the names have been changed. I suspect that changing the names of columns will cause confusion. Quoting is a bit inconvenient for me, but preventing confusion is more important.

I did the quoting manually for the table creation, but I dynamically generate the sql query based on the keys specified to save in `key_name_order`. So to fix it here I used the following regular expression substitution: `/([A−Z0−9a−z]+−[A−Z0−9a−z]+)/"\1"/`

3 Airflow

During lecture 4 directed acyclic graphs (dag) were discussed. Two tools were mentioned: Airflow and DBT. I found this concept very interesting and I decided to investigate Airflow because that tool appealed to me more.

However, using Airflow did not go well for me. After trying to install Airflow for about 2 hours using pip it seemed like it installed correctly, but then I still got weird import errors. Airflow is the only python package I have ever seen that needs a constraints file to install. So I gave up on the pip install and tried with docker. Installing with docker went a lot better but this installation came with many examples. All these examples were honestly very a bit overwhelming. Writing Airflow scripts itself is qutie easy, but Airflow is bad at hiding that there is a large amount of complexity within. I did make an Airflow python file that executed part of my pipeline but using Airflow to solve my problem just felt overly complicated as opposed to just writing some python code myself and managing it using poetry. I

felt like the pipeline I wanted to make did not need all the complexity that Airflow brings. Besides I could not actually find a way to trigger the Airflow pipeline based on an incoming stream from a socket connection.

Instead of Aiflow I just wrote a simple Python package. The dependencies can be easy installed using poetry and you can run the project with poetry too. If you would really want to you could even put the package in a container.

I can definitely see that if you have many pipelines with tens to hundreds to thousands of steps that Airflow could be very useful, but it was just overkill for my project.

4 Appendix

4.1 Running the pipeline instructions

1. Clone the code from the Github reposity
2. Install poetry using `pipx install poetry` or using the instructions from their website
3. Run `poetry install` in the root directory of the repository to install the dependencies
4. Fill in the auth.toml configuration file using your tokens from knmi.nl
5. Run `poetry run listen-nc` to start the pipeline

4.2 Table of configuration files

Key	File	Description
CLIENT_ID	`auth.toml`	Identify yourself to the mqtt stream
TOPIC	`config.toml`	Topic of the mqtt stream
BROKER_DOMAIN	`config.toml`	Domain of the mqtt stream
TOKEN	`auth.toml`	Secret token used to connect to the mqtt stream
FILE_DOWNLOAD_TOKEN	`auth.toml`	Secret token used to get nc file download links

4.3 Wrangling functions

4.3.1 process_nc_file

```
from pathlib import Path
import netCDF4 as nc
import numpy as np
from typing import List

nc_keys_to_save = ( "station", "time", "wsi", "stationname", "lat", "lon",
 ↪  "height", "D1H", "dd", "dn", "dr", "dsd", "dx", "ff", "ffs", "fsd",
 ↪  "fx", "fxs", "gff", "gffs", "h", "h1", "h2", "h3", "hc", "hc1", "hc2",
 ↪  "hc3", "n", "n1", "n2", "n3", "nc", "nc1", "nc2", "nc3", "p0", "pp",
 ↪  "pg", "pr", "ps", "pwc", "Q1H", "Q24H", "qg", "qgn", "qgx", "qnh",
 ↪  "R12H", "R1H", "R24H", "R6H", "rg", "rh", "rh10", "Sav1H", "Sax1H",
 ↪  "Sax3H", "Sax6H", "sq", "ss", "Sx1H", "Sx3H", "Sx6H", "t10", "ta", "tb",
 ↪  "tb1", "Tb1n6", "Tb1x6", "tb2", "Tb2n6", "Tb2x6", "tb3", "tb4", "tb5",
 ↪  "td", "td10", "tg", "tgn", "Tgn12", "Tgn14", "Tgn6", "tn", "Tn12",
 ↪  "Tn14", "Tn6", "tsd", "tx", "Tx12", "Tx24", "Tx6", "vv", "W10",
 ↪  "W10-10", "ww", "ww-10", "zm")

Measurements = List[List[str | int | float]]

def process_nc_file(filename: str | Path) -> Measurements:
    """ Process a single NetCDF file and return a list of lists with values
     ↪  in the order of nc_keys_to_save."""

    db = nc.Dataset(filename) # Step 2
    try:
        time = db.variables['time'][:].item()  # step 3

       to_insert = []  # List of data to insert into the database in the end.
        # For each station iterate each key in the nc file
        for index, station in enumerate(db.variables['station']):
            to_insert.append([])  # Step 4
            for key in nc_keys_to_save: # Step 5

                if key == "time":
                    to_insert[index].append(time)
                    continue

                station_reading = db.variables[key][index]
```

```
                is_string = isinstance(station_reading, str)
                if is_string:
                    to_insert[index].append(station_reading)
                    continue

                # Else check for a masked array
                is_masked = False
                if isinstance(station_reading, np.ma.core.MaskedArray) and
                 ↪  station_reading.mask:
                    is_masked = station_reading.mask[0]

                if is_masked:
                    to_insert[index].append(None)
                    continue

               value = station_reading.item()  # Otherwise just get the item
                to_insert[index].append(value)

    finally:
        db.close()

    return to_insert
```

4.3.2 insert_nc_filerows

```
def insert_nc_filerows(result: Measurements): # Step 6
    con = get_connection()
    con.executemany(insert_statement, result)
```

4.4 Schema of the Measurement table:

```
CREATE TABLE Measurement (
    station      TEXT NOT NULL,
    time         BIGINT NOT NULL CHECK ( time > 0 ),
    wsi          TEXT,
    stationname  TEXT,
    lat        DOUBLE,
    lon        DOUBLE,
    height     DOUBLE,
    D1H        DOUBLE,
    dd         DOUBLE,
    dn         DOUBLE,
    dr         DOUBLE,
    dsd        DOUBLE,
    dx         DOUBLE,
    ff         DOUBLE,
    ffs        DOUBLE,
    fsd        DOUBLE,
    fx         DOUBLE,
    fxs        DOUBLE,
    gff        DOUBLE,
    gffs       DOUBLE,
    h          DOUBLE,
    h1         DOUBLE,
    h2         DOUBLE,
    h3         DOUBLE,
    hc         DOUBLE,
    hc1        DOUBLE,
    hc2        DOUBLE,
    hc3        DOUBLE,
    n          DOUBLE,
    n1         DOUBLE,
    n2         DOUBLE,
    n3         DOUBLE,
    nc         DOUBLE,
    nc1        DOUBLE,
    nc2        DOUBLE,
    nc3        DOUBLE,
    p0         DOUBLE,
    pp         DOUBLE,
    pg         DOUBLE,
    pr         DOUBLE,
```

```
    ps          DOUBLE,
    pwc         DOUBLE,
    Q1H         DOUBLE,
    Q24H        DOUBLE,
    qg          DOUBLE,
    qgn         DOUBLE,
    qgx         DOUBLE,
    qnh         DOUBLE,
    R12H        DOUBLE,
    R1H         DOUBLE,
    R24H        DOUBLE,
    R6H         DOUBLE,
    rg          DOUBLE,
    rh          DOUBLE,
    rh10        DOUBLE,
    Sav1H       DOUBLE,
    Sax1H       DOUBLE,
    Sax3H       DOUBLE,
    Sax6H       DOUBLE,
    sq          DOUBLE,
    ss          DOUBLE,
    Sx1H        DOUBLE,
    Sx3H        DOUBLE,
    Sx6H        DOUBLE,
    t10         DOUBLE,
    ta          DOUBLE,
    tb          DOUBLE,
    tb1         DOUBLE,
    Tb1n6       DOUBLE,
    Tb1x6       DOUBLE,
    tb2         DOUBLE,
    Tb2n6       DOUBLE,
    Tb2x6       DOUBLE,
    tb3         DOUBLE,
    tb4         DOUBLE,
    tb5         DOUBLE,
    td          DOUBLE,
    td10        DOUBLE,
    tg          DOUBLE,
    tgn         DOUBLE,
    Tgn12       DOUBLE,
    Tgn14       DOUBLE,
    Tgn6        DOUBLE,
```

```
    tn          DOUBLE,
    Tn12        DOUBLE,
    Tn14        DOUBLE,
    Tn6         DOUBLE,
    tsd         DOUBLE,
    tx          DOUBLE,
    Tx12        DOUBLE,
    Tx24        DOUBLE,
    Tx6         DOUBLE,
    vv          DOUBLE,
    W10         DOUBLE,
    "W10-10"    DOUBLE,
    ww          DOUBLE,
    "ww-10"     DOUBLE,
    zm          DOUBLE,
    PRIMARY KEY (time, station)
    );
```