Exercise 2

1. Merge all data sets

Start a new Python script on your local machine. You can also work on the FASRC, e.g., using a Jupyter Notebook. In this case, all case study related data is publicly available (from within the FASRC network) at `/n/holyscratch01/cga/python-workshop-gis-big-data` However, working in your go-to environment might be faster for you.

If you are working on your own device, ensure to have a copy of all data set files ready. Also, create a new Python environment, providing the same packages as we installed in Chapter 1. You can use the same `requirements.txt` file: https://raw.githubusercontent.com/cga-harvard/python-workshop-gis-big-data/main/Chapter%201/requirements.txt

Now, create a new Python file. You'll find the following example code in the GitHub repo: https://github.com/cga-harvard/python-workshop-gis-big-data/tree/main/Chapter%202

```
#!/usr/bin/env python

import pandas as pd
import numpy as np
from datetime import datetime
from tqdm import tqdm
import xarray as xr


DATASET_ROOT = 'PATH/TO/DATASET'
YEAR = 2022
```

1.1 Load the TSGI dataset

To load a CSV file, use Pandas' `pd.read_csv(filename)` function. This reads the file and returns a Pandas data frame (DF) object:

```
tweets_df = pd.read_csv(f'{DATASET_ROOT}/twitter_sentiment_geo_index/
num_posts_and_sentiment_summary_{YEAR}.csv')
```

For better readability, I suggest to rename the columns of the file, e.g., to

```
tweets_df.columns = ['date', 'country', 'state', 'county',
'sentiment_score', 'tweets']
```

Lastly, state and county names are concatenated, e.g., `United States_New Mexico_Torrance`. To have only the state name and only the county name in the corresponding columns, extract the relevant names using string-processing, e.g.,

```
tweets_df['state'] = tweets_df.state.apply(lambda x: x.split('_')[-1])
tweets_df['county'] = tweets_df.county.apply(lambda x:
x.split('_')[-1])
```

Now, select a few states for our toy example. This limitation can later easily be removed, but for now it allows us to work with a much more manageable subset:

```
state_subset = ['Massachusetts', 'Connecticut', 'Rhode Island']
tweets_subset = tweets_df[tweets_df.state.isin(state_subset)].copy()
```

If you now inspect the variable `tweets_subset`, it should contain 9671 rows. Let's add geo coordinates to the table. As discussed, the geocoding was already done and can be loaded as a separate DF:

```
county_coordinates =
pd.read_csv(f'{DATASET_ROOT}/county_coordinates/lookup.csv')
```

To merge this new DF with our existing `tweets_subset`, employ Pandas' merge function:

```
tweets_subset = tweets_subset.merge(county_coordinates, on=['country',
'state', 'county'], how='left')
tweets_subset = tweets_subset.dropna(subset=['lat', 'lon'])
```

The last row ensures that we only consider rows that have a lat & lon value, and drop all rows for which the merge wasn't successful. If you now inspect the `tweets_subset` variable, the DF should still have 9671 rows, but reach row should have two more columns: lat and lon coordinates.

1.2 Load the NOAA CPC dataset & augment the tweets DF

To load the NOAA CPC dataset, we use the xarray package that provides functionalities specific to the NetCDF data format. The code is very similar to what we're used to when working with the Pandas library:

```
noaa_cpc_dataset =
xr.open_dataset(f"{DATASET_ROOT}/precipitation/precip.{YEAR}.nc")
```

Now we have access to the precipitation values through the `noaa_cpc_dataset` object. While the object also stores a lot of meta data, the `noaa_cpc_dataset.precip.values` provides the three dimensional array with the `[day, latitude, longitude]` dimensions.

However, the coordinates of our counties don't line up perfectly with the 0.5 x 0.5 grid of the NOAA CPC dataset. Hence, when we want to augment the tweets with precipitation data, we need to find the closest NOAA CPC coordinates per county. Also, the tweets come with a date in the format "2022-05-27" to indicate the day of the year, while our NOAA dataset expects a single value in `[0, 364]` (note: zero-based numbering) for the day of the year. Converting the day is probably the easiest. Assuming `row` contains one row of the `tweets_subset` DF (so we can easily iterate over the DF):

```
# compute the array index for the day of the year
day_idx = datetime.strptime(row.date, "%Y-%m-%d").
timetuple().tm_yday - 1
```

Here, we initialize a new datetime object using the `strptime` constructor. This allows us to create a datetime object from a string of the format "2022-05-27". Next, we use the `timetuple()` function of datetime objects. This returns many helpful values for working with time. Out of the lot, we select the `tm_yday` attribute, which contains the day of the year. Lastly, we remove one from the value, since our array is zero-based.

Next, let's compute the best matching x (or longitude) index of the array, for the given longitude value of the county in our row variable. Here, we leverage that the dataset comes with some metadata, including a list of all the longitude values of the coordinate grid it provides. The strategy here is to compute the difference between our actual county longitude variable, and all the dataset longitude values. The one pair with the smallest difference is the closest coordinate. If we know which element has the smallest difference, we can simply ask for the index in the list. But first, according to the dataset description, the dataset expects longitude values in the range between [0, 360]. Hence, we have to convert our negative longitude values first.

```
# compute array index for longitude value
lon_values = noaa_cpc_dataset.indexes['lon']
if row.lon < 0:
      lon_0_to_360 = row.lon + 360
else:
      lon_0_to_360 = row.lon
x = np.abs(lon_values - lon_0_to_360).argmin()
```

The conversion of the latitude values follows the same logic. The only difference is that they don't have to be converted first, as the NOA CPC dataset and our tweets both save latitude values in the range [-90, 90].

```
# compute array index for latitude value
lat_values = noaa_cpc_dataset.indexes['lat']
y = np.abs(lat_values - row.lat).argmin()
```

Put all together, we can write this as one function: it expects one element of our tweet DF and returns the precipitation amount in millimeters for this element:

```
def precipitation_for_row(row):

    # compute the array index for the day of the year
    day_idx = datetime.strptime(row.date, "%Y-%m-%d").
    timetuple().tm_yday - 1

    # compute array index for longitude value
    lon_values = noaa_cpc_dataset.indexes['lon']
    if row.lon < 0:
        lon_0_to_360 = row.lon + 360
    else:
        lon_0_to_360 = row.lon
    x = np.abs(lon_values - lon_0_to_360).argmin()

    # compute array index for latitude value
    lat_values = noaa_cpc_dataset.indexes['lat']
    y = np.abs(lat_values - row.lat).argmin()

    # read the precipitation value using the three computed indexes
    return noaa_cpc_dataset.precip.values[day_idx, y, x]
```

With such a function in place, adding precipitation information to all the tweets in our subset can be done elegantly in one line:

```
# augment tweets table with the NOAA CPC precipitation data
tweets_subset['precipitation'] = tweets_subset.apply(lambda row:
precipitation_for_row(row), axis=1)

# remove nan values
tweets_subset = tweets_subset.dropna(subset=['precipitation'])
```

2. Analyze the results

To investigate our research question, we can now – that we have all relevant information in one table – analyze the results. Similarly, to 1.2, we write a function that generates all relevant results at once. However, this time, we don't feed a single row into the function, but rather the entire subset of a county we're investigating. After all, we want to contrast all the days of rain against all the days of no rain, but only per county. Assume `grouped_df` contains all the rows of a given county and of the entire year. Furthermore, to only count rainy days with a significant amount of rain, let's define

```
RELEVANT_PRECIPITATION_THRESHOLD = 12 * 2.5
```

This is equal to 12h of 2.5mm rain, typically what is considered as medium intense rain. Then

```
no_rain_df = grouped_df[grouped_df.precipitation == 0]
```

gives us all the days of no rain of this county, and, similarly,

```
rain_df = grouped_df[grouped_df.precipitation >=
RELEVANT_PRECIPITATION_THRESHOLD]
```

gives us all the rainy days. These two subsets can then be used to compute the average sentiment score per group:

```
no_rain_mean = no_rain_df.sentiment_score.mean()
rain_mean = rain_df.sentiment_score.mean()
group_diff_percent = (no_rain_mean - rain_mean) * 100.0
```

All this together can then be called as a function. As its input, we can group our tweet dataset by country, state, and county, and feed these groups, one by one, to our analysis function. Altogether, this code looks like the following:

```
RELEVANT_PRECIPITATION_THRESHOLD = 12 * 2.5

def compute_statistics(grouped_df):
    no_rain_df = grouped_df[grouped_df.precipitation == 0]
    rain_df = grouped_df[grouped_df.precipitation >=
    RELEVANT_PRECIPITATION_THRESHOLD]

    no_rain_mean = no_rain_df.sentiment_score.mean()
    rain_mean = rain_df.sentiment_score.mean()
    group_diff_percent = (no_rain_mean - rain_mean) * 100

    return no_rain_mean, rain_mean, group_diff_percent

# group the DF and compute statistics
grouped_statistics = (
    tweets_subset
    .groupby(['country', 'state', 'county'])
    .apply(compute_statistics)
)

# create a summary DF
summary_df = pd.DataFrame(grouped_statistics.tolist(),
columns=['no_rain_mean', 'rain_mean', 'group_diff'],
index=grouped_statistics.index)
```

Now, `summary_df` contains the sentiment difference between rainy and non-rainy days per county. If we'd want to aggregate this per state or country, we can simply call:

```
summary_df.groupby('state').group_diff.mean()
summary_df.groupby('country').group_diff.mean()
```

What you learned in this exercise:

- How to load CSV and NetCDF files into Python
- How to merge DFs using Pandas
- How to filter DFs and work with `groupby()` and `apply()`