7CCSMCMP: Coursework 2

Computer Programming for Data Scientists

The 7CCSMCMP instructors

2022

Contents

0 Instructions

This document describes Coursework 2 of the 7CCSMCMP module. The coursework has three activities, which focus on the main concepts covered in Weeks 7 to 11. We start with some general instructions on how to format your answers for submission. **Please read these instructions carefully before you start coding**. They explain the correct format and documentation your solutions must comply with. Coursework assessors may be not able to execute and assess code that does not follow the instructions, which may lead to considerable penalties to your coursework mark.

Each of the three activities in the coursework must be addressed in a separate IPython notebook file; you should submit a *zipped file* containing *three notebook files with extension .ipynb*, one per

each activity.

Each notebook must be organised as shown in Figure 1. First, create a cell for all your library imports, and give it the title *Imports* with a markdown header. Then, if the activity has sub-activities, organise the notebook into separate sequences of cells for each sub-activity using markdown headers. If the activity is not split in sub-activities, create a single markdown header called *Activity* instead. Answer each task in *a single cell*, and give it a markdown header with the task number.

Tasks require you to write some code. Some tasks in Activity 1 ask you to provide explanations as free text - write the explanations as comments **at the end** of the relevant cell.

Pleas,e use the proposed markdown headers to clearly signal each task solution cell to help the markers assess your solutions correctly.

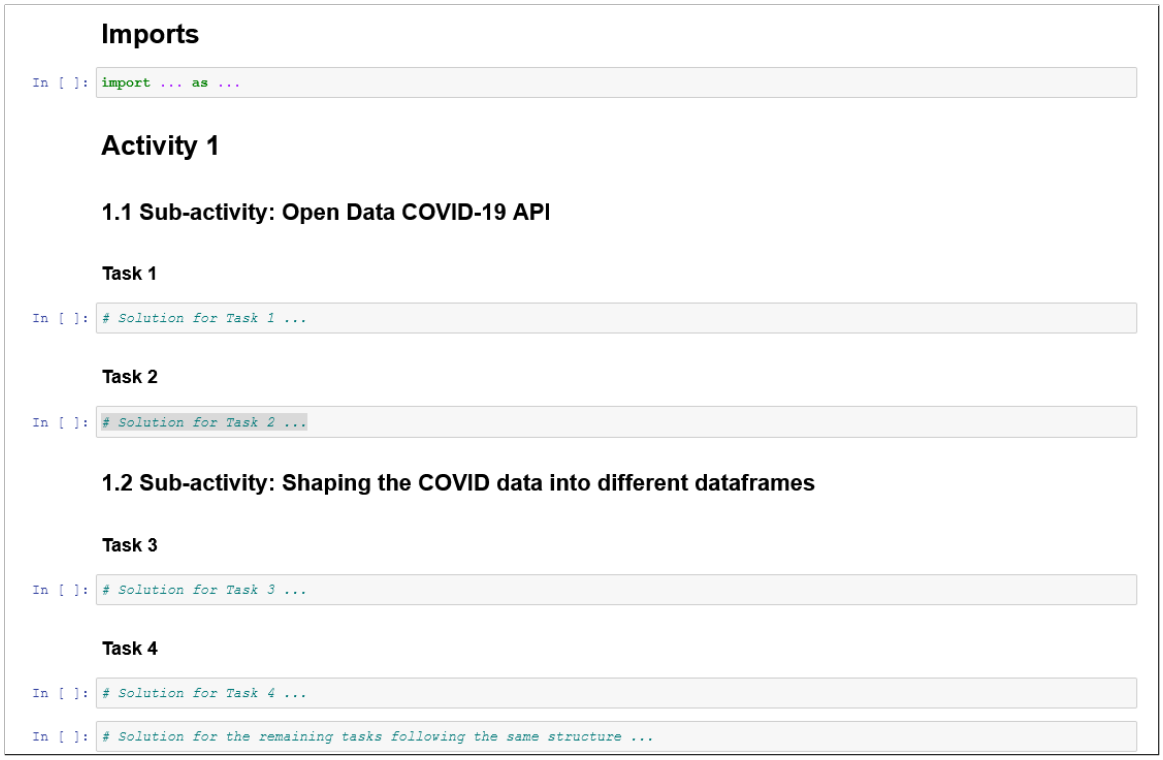

Figure 1: Example of a correctly organised IPython notebook for Activity 1

1 Activity 1 (60 points)

In this activity, you will be asked to do three things:

1. Query an open data API for public COVID information;
2. Mold this information into several dataframes according to our instructions;
3. Create quick data transformations and plots to answer specific questions.

This activity's solutions should be provided in a single IPython Notebook file, named `CW2_A1.ipynb`.

1.1 Sub-activity: Open Data COVID-19 API

The UK government has a portal with data about the Coronavirus in the UK; it contains data on cases, deaths, and vaccinations on national and local levels. The portal is available on this page: https://coronavirus.data.gov.uk. You can acquire the data by querying its API.

We ask you to use the **requests** library in order to communicate with the API. The documentation is available at: https://docs.python-requests.org/en/latest. Read carefully the API documentation at:
https://coronavirus.data.gov.uk/details/developers-guide/main-api.

Then complete the following tasks in order to acquire the data.

Task 1: Create a function `get_API_data(filters, structure)` that sends a specific query to the API and retrieves all the data provided by the API that matches the query. The function requires two arguments:

- `filters (dictionary)` are the filters to be passed to the query, as specified in the API documentation. This will be a dictionary where keys are filter metrics and values are the values of those metrics. For example, you may want to filter the data by nation, date etc. As seen in the API documentation, filters are passed to the API's URL as a *URL parameter*. This means you will have to format *filters* inside get_API_data in a way that the API can accept it as an argument.
- `structure (dictionary)` will specify what information the query should return, again as specified in the API documentation. This will be a dictionary where the keys are the names you wish to give to each metric, and the values are the metrics as specified in the API. The structure argument specifies what attributes from the records that match the filters you wish to obtain, such as date, region, daily casualties etc. The argument is *passed as an URL parameter* to the API's URL. This means you will have to format *structure* inside get_API_data in a way that the API can accept it as an argument.

The function `get_API_data` should return a list of dictionaries answering the query.

To ensure you receive all data matching your query, use the page URL parameter. The function should get data from all pages and return everything as a single list of dictionaries.

An example of the full URL with filter, structure, and page parameters defined can be seen in Listing 1; this URL, when queried, returns the first page (pages begin at 1) with

data at a regional level, retrieving only the date and new cases by publishing date, and naming them *date* and *newCases*, respectively.

```
https://api.coronavirus.data.gov.uk/v1/data?filters=areaType=region&structure={"date":"date",
"newCases":"newCasesByPublishDate"}&page=1
```

Listing 1: Example of API query URL with filters, structure, and page.

Task 2: Write a script that calls the function `get_API_data` twice, producing **two** lists of dictionaries: `results_json_national` and `results_json_regional`. Both lists should consist of dictionaries with the following key-value pairs:

- `date (string)`: The date to which this observation corresponds to;
- `name (string)`: The name of the area covered by this observation (could be a nation, region, a local authority, etc);
- `daily_cases (numeric)`: The number of new cases at that date and in that area by specimen date;
- `cumulative_cases (numeric)`: The cumulative number of cases at that date and in that area area by specimen date;
- `daily_deaths (numeric)`: The number of new deaths at that date and in that area after 28 days of a positive test, by publishing date;
- `cumulative_deaths (numeric)`: The cumulative number of deaths at that date and in that area after 28 days of a positive test, by publishing date;
- `cumulative_vaccinated (numeric)`: The cumulative number of people who completed their vaccination (both doses) by vaccination date;
- `vaccination_age (dictionary or list of dictionaries)`: A demographic breakdown of cumulative vaccinations by age intervals for all people.

The first list of dictionaries obtained (`results_json_national`) should have data at the *national level* (England, Wales, Scotland, Northern Ireland). The second (`results_json_regional`) should have data at a *regional level* (London, North West, North East, etc). Both should contain data for all dates covered by the API.

Attention: Do not query the API too often, as you might be blocked or compromise the API's service. The API service is used by many other organisations, which rely on it for vital tasks. It is your responsibility to query the API by respecting its rules. *We ask students to keep requests under* 10 *requests every* 100 *seconds, and* 100 *total requests every hour*. When querying the API, if your response has a 429 status code (or a similar code indicating your query failed), check for a header called "Retry-After", which indicates how much time you have to **wait** before doing another query; you should wait that long.

1.2 Sub-activity: Shaping the COVID data into different dataframes

These two lists of dictionaries from before are a good start. However, they are not the easiest way to turn data into insight.

In the following, you will take the data from these lists of dictionaries and turn it into *Pandas* dataframes. Dataframes have quick transformation, summarising, and plotting functionalities which let you analyse the data more easily.

The code should use native *Pandas* methods. Implementing the functionality manually (e.g. using loops or directly accessing the arrays inside the dataframes) will be penalised. Follow the library's documentation. Remember that *Pandas* methods can very often be chained; use that to your advantage.

Task 3: Concatenate the two lists of dictionaries (`results_json_national` and `results_json_regional`) into a single list.

Task 4: Transform this list into a dataframe called *covid_data*, which should now have one column for each metric retrieved from the API (*date*, *name*, *daily_cases*, *cumulative_cases*, *daily_deaths*, *cumulative_deaths*, *cumulative_vaccinated*, *vaccination_age*).

Task 5: The regional portion of the dataframe is a breakdown of the data from England. Thus, all observations in England are contained in the dataframe twice. Hence you can erase all rows in which the `name` column have the value "England".

Task 6: The column *name* has an ambiguous title. Change it to *area*.

Task 7: The `date` column is of type *object*, which is for strings and other types. This makes it harder to filter/select by month, year, to plot according to time, etc. Convert this entire column to the *datetime* type.

Task 8: Print a summary of the dataframe, which includes the amount of missing data. How you measure the amount of missing data is up to you. Please document your decision in the code.

Task 9: For the cumulative metrics columns (`cumulative_deaths`, `cumulative_cases`, `cumulative_vaccinated`), replace missing values with the most recent (up to the date corresponding to that missing value) existing values for that area. If none exist, leave it as it is. For example, if there is a missing value in the `cumulative_deaths` column at the date *08-02-2021*, look at all non-missing values in the `cumulative_deaths` columns whose date is lower than *08-02-2021* and take the most recent.

Task 10: Now, remove the rows that still have missing values in the cumulative metrics columns mentioned in the last question.

Task 11: Rolling averages are often better indicators of daily quantitative metrics than raw daily measures. Create two new columns. One, with the 7-day rolling average of new daily cases in that area, including the current day, and one with the same calculation but for daily deaths. Name them `daily_cases_roll_avg` and `daily_deaths_roll_avg`.

Task 12: Now that we have the rolling averages, drop the columns `daily_deaths` and `daily_cases` as they contain redundant information.

Task 13: A column in the dataframe `covid_data` has dictionaries as values. We can transform this column into a separate dataframe. Copy the columns `date`, `area`, and

`vaccination_age` into a new dataframe named `covid_data_vaccinations`, and drop the `vaccination_age` column from `covid_data`.

Task 14: Transform `covid_data_vaccinations` into a new dataframe called `covid_data_vaccinations_wide`. Each row must represent available vaccination metrics for a specific date, in a specific area, and for a specific age interval. The dataframe must have the following columns:

- `date`: The date when the observation was made;
- `area`: The region/nation where the observation was made;
- `age`: The age interval that the observation applies to;
- `VaccineRegisterPopulationByVaccinationDate`: Number of people registered for vaccination;
- `cumPeopleVaccinatedCompleteByVaccinationDate`: Cumulative number of people who completed their vaccination;
- `newPeopleVaccinatedCompleteByVaccinationDate`: Number of new people completing their vaccination;
- `cumPeopleVaccinatedFirstDoseByVaccinationDate`: Cumulative number of people who took their first dose of vaccination;
- `newPeopleVaccinatedFirstDoseByVaccinationDate`: Number of new people taking their first dose of vaccination;
- `cumPeopleVaccinatedSecondDoseByVaccinationDate`: Cumulative number of people who took their second dose of vaccination;
- `newPeopleVaccinatedSecondDoseByVaccinationDate`: Number of new people taking their second dose of vaccination;
- `cumVaccinationFirstDoseUptakeByVaccinationDatePercentage`: Percentage of people out of that demographic who took their first dose of vaccination;
- `cumVaccinationCompleteCoverageByVaccinationDatePercentage`: Percentage of people out of that demographic who took all their doses of vaccination;
- `cumVaccinationSecondDoseUptakeByVaccinationDatePercentage`: Percentage of people out of that demographic who took their second dose of vaccination.

1.3 Sub-activity: Aggregating, plotting, and analysing

We have created dataframes for our analysis. We will ask you to answer several questions with the data from the dataframes. For each question, follow the same three steps:

1. aggregate and/or shape the data to answer the question and save it as an intermediate dataframe;
2. apply plot methods on the dataframe to create a *single plot* to visualise the transformed data;
3. write your conclusion as comments or markdown.

Figure 2 shows an example of how your code should be organised for each question.

Some questions will use data and plots from a previous question and require you only to answer the question; in this case, either have a cell with only comments or only a markdown cell with the answer.

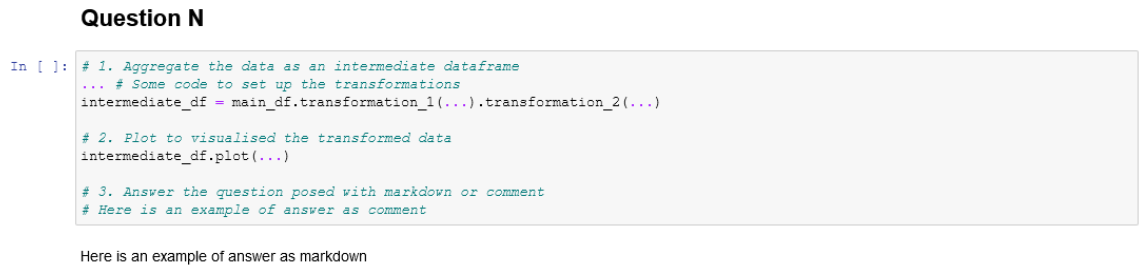

Figure 2: Example of a cell that answers a question from sub-activity 1.3

Plotting should be done exclusively using native Pandas visualisation methods, described here. To make these answers clear for us, we ask you to use concise and clear transformations and to add comments to your code.

Task 15: Show the cumulative cases in London as they evolve through time.

Question: Is there a period in time in which the cases plateaued?

Task 16: Show the evolution through time of cumulative cases summed over all areas.

Question: How does the pattern seen in London hold country-wide?

Task 17: Now, instead of summing the data over areas, show us the evolution of cumulative cases of different areas as different lines in a plot.

Question: What patterns do all nations/regions share?

Task 18: *Question*: As a data scientist you will often need to interpret data insights, based on your own judgement and expertise. Considering the data and plot from the last question, what event could have taken place in June-July that could justify the trend seen from there onward?

Task 19: Show us the evolution of cumulative deaths in London through time.

Question: Is there a noticeable period in time when the ongoing trend is broken? When?

Task 20: *Question*: Based on the data and plot from the last question, is there any similarity between trends in cumulative cases and cumulative deaths?

Task 21: Create a new column, `cumulative_deaths_per_cases`, showing the ratio between cumulative deaths and cumulative cases in each row. Show us its sum over all regions/nations as a function of time.

Question: What overall trends can be seen?

Task 22: *Question*: Based on the data and plot from the last question, it seems like, in June-July, the graph's inclination gets steeper. What could be a reasonable explanation?

Task 23: Show us the sum of cumulative vaccinations over all areas as a function of time.

Question: Are there any relationships between the trends seen here and the ones seen in Task 21?

Task 24: Show us the daily cases rolling average as a function of time, separated by areas.

Question: Is there a specific area that seems to escape the general trend in any way? Which one and how?

Task 25: Show us the daily cases rolling average as a function of time for the area identified in the previous question alongside another area that follows the general trend, in order to compare them.

Question: What reasons there might be to justify this difference?

Task 26: To be able to compare numbers of cases and deaths, we should normalise them. Create two new columns, `daily_cases_roll_avg_norm` and `daily_deaths_roll_avg_norm`, obtained by performing a simple normalisation on all values in the `daily_cases_roll_avg` and `daily_deaths_roll_avg` columns; for each column, you divide all values by the maximum value in that column.

Now, on the same line plot with *date* as the x-axis, plot two lines: the normalised rolling average of deaths and the normalised rolling average of cases summed over all areas.

Question: Are daily trends of cases and deaths increasing and decreasing at the same rates? What part of the plot tells you this?

Task 27: The dataframe `covid_data_vaccinations_wide` has some columns expressed as percentage of population. First, split this dataframe into two dataframes, one for London, one for Scotland.

Now, mould the London dataframe such that each row corresponds to a date, each column corresponds to an age interval, and the data in a dataframe cell is the value of *cumVaccinationFirstDoseUptakeByVaccinationDatePercentage* for that age interval and date.

Plot the London dataframe as a line chart with multiple lines, each representing an age interval, showing the growth in vaccination coverage per age group.

Because this plot will generate over ten lines, colours will repeat. Add this argument to your call of the `plot()` method: `style=['--' for _ in range(10)]`. This will force the first ten lines to become dashed.

Question: Were all age groups vaccinated equally and at the same time, or was there a strategy employed? What strategy does the plot indicate and why?

Task 28: Do the same transformations asked in the last question, but for the Scotland dataframe.

Question: In both plots, compare how vaccination evolved for two sections of population: 50-64 years and 65-79 years. Were there any differences in the strategies employed between London and Scotland for dealing with both sections?

2 Activity 2 (15 points)

This activity is about graphs and requires using the *NetworkX Python* package (refer to the documentation if needed). The goal of the activity is to create a script that randomly generates a graph like the one shown in Figure 3. Additionally, you will be asked to compute some metrics on the generated graph and to export it as a file in *JSON* format.

This activity's solutions should be provided in a single IPython Notebook file, named `CW2_A2.ipynb`.

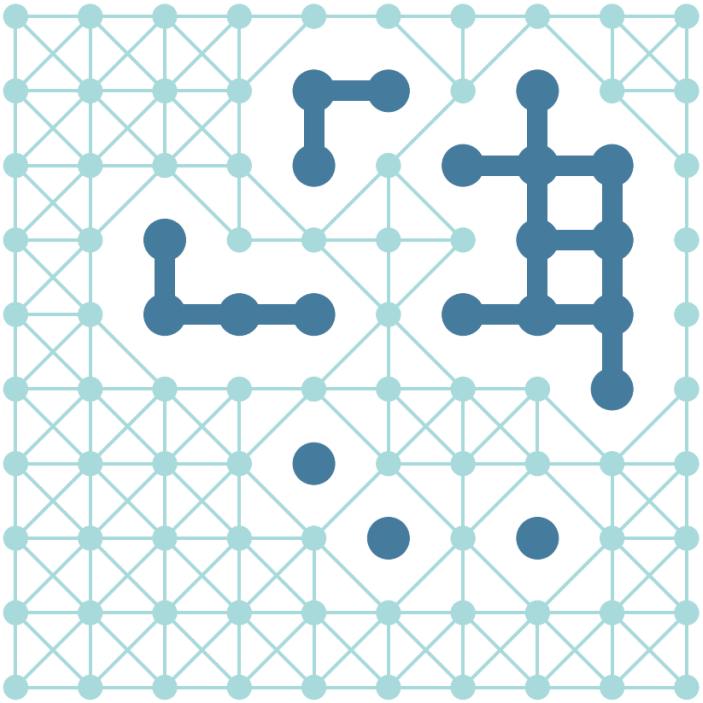

Figure 3: Example of a randomly generated graph

The graph has its nodes placed in a matrix fashion. Each node has a type (background or foreground) and can be connected only with neighbouring nodes of the same type:

- **Background** nodes are connected to all its neighbours of the same type (horizontally, vertically, diagonally).
- **Foreground** nodes are fully connected as well, again to neighbouring nodes of the same type, but only horizontally and vertically.

Edges between background and foreground nodes are not allowed.

Before you start coding, declare some constants at the beginning of your Notebook file; put them right after the *imports* in the import cell. Figure 3 lists the constants, with default values in brackets:

- `MATRIX_SIDE` (10), the number of nodes per each side of the matrix;

- `NO_FG_NODES` (20), the total number of foreground nodes that will be randomly located in the graph. Foreground nodes cannot be placed on the matrix borders;
- `COLOR_BG` (#a8dadc), the color of the background nodes and edges;
- `COLOR_FG` (#457b9d), the color of the foreground nodes and edges;
- `SIZE_BG_NODES` (800), the size of the nodes in the background;
- `SIZE_FG_NODES` (2400), the size of the nodes in the foreground;
- `WEIGHT_BG_EDGES` (4), the weight of the edges in the background;
- `WEIGHT_FG_EDGES` (24), the weight of the edges in the foreground;
- `FIG_SIZE` ((12, 12)), the size of the figure to be passed as an argument to the `plt.figure()` function for plotting.

```
# All the import goes here

COLOR_BG = "#a8dadc"
COLOR_FG = "#457b9d"

SIZE_BG_NODES = 800
SIZE_FG_NODES = 2400

EDGE_BG_WEIGHT = 4
EDGE_FG_WEIGHT = 24

MATRIX_SIDE = 10
NO_FG_NODES = 20

FIG_SIZE = (12, 12)

# The rest of the activity code goes here
```

Listing 2: Constants to create the graph.

2.1 Sub-activity: Graph creation

The first part of this activity requires you to create a graph through three tasks:

Task 1: Create the background matrix of nodes as shown in Fig. 4. Use two nested loops for that.

Task 2: Choose randomly a list of `NO_FG_NODES` background nodes from the matrix created in Task 1 and convert them into foreground nodes. Remember in this step to avoid nodes located on the borders, meaning the first and last matrix row and the first and last matrix column. An example of a graph is in Figure 5; yours will be different, as the foreground nodes are picked randomly each time one runs the code.

Task 3: Add the edges. Remember that edges involve only nodes of the same type that are next to each other. In particular both *background* and *foreground* nodes must be connected

Figure 4: Matrix of background notes to be used as a frame.

with their vertical and horizontal neighbours, but only *background* nodes can connect with their diagonal neighbours.

2.2 Sub-activity: Graph manipulation and output

Now compute some standard metrics on the graph and export it as a *JSON* file.

Task 4: The *NetworkX* package offers a series of methods to analyse graphs:

- Print out the graph info using the `nx.info()` function;
- Print out the density of the graph;
- Print out the degree centrality of its nodes.

If needed, refer to the *NetworkX* documentation.

Task 5: Finally, the method `json_graph` from `networkx.readwrite` allows to export a *NetworkX* graph in *json*. Use it to save your masterpiece into a *JSON* file. Listing 3 shows an example of how the exported JSON file should look like.

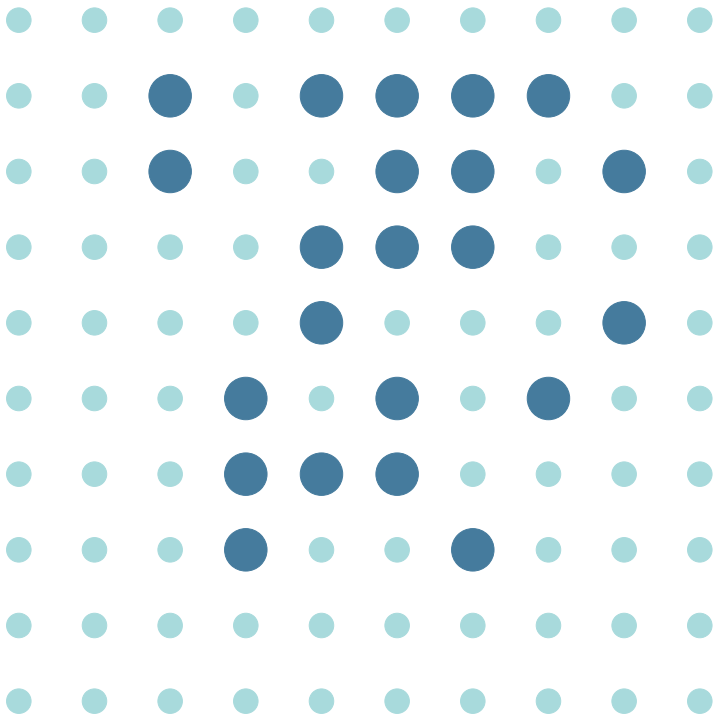

Figure 5: The graph after `NO_FG_NODES` background nodes, randomly chosen, into foregrounds nodes.

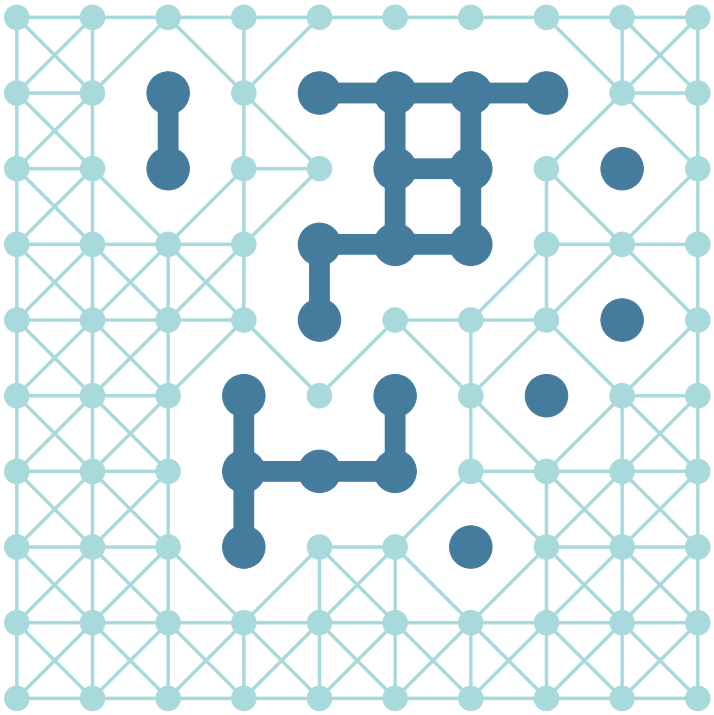

Figure 6: After adding the edges your masterpiece is complete.

```
{
    "directed": false,
    "multigraph": false,
    "graph": {},
    "nodes": [
        {
            "color": "#a8dadc",
            "node_type": "bg",
            "id": "0-0"
        },
        {
            "color": "#a8dadc",
            "node_type": "bg",
            "id": "0-1"
        },

...
...
...
  "links": [
        {
            "color": "#a8dadc",
            "weight": 4,
            "source": "0-0",
            "target": "1-0"
        },
        {
            "color": "#a8dadc",
            "weight": 4,
            "source": "0-0",
            "target": "0-1"
        },
...
...
...
   ]
}
```

Listing 3: Example of a exported *JSON* file representing a graph.

3 Activity 3 (25 points)

In this activity you will be required to do the following:

1. Create a text corpus using data available on the web;
2. Process the text data and apply NLP operations;
3. Visualise the results using *matplotlib*.

This activity's solutions should be provided in a single IPython Notebook file, named `CW2_A3.ipynb`.

We will use the WikiData API and the Wikipedia API to obtain information about the ACM Turing Award winners. Use the API documentations to get familiar with them. The ACM Turing award is a prize given annually by the Association for Computing Machinery (ACM) for contributions of lasting and major importance to computer science.

3.1 Sub-activity: Loading and pre-processing of text data

Task 1: Create a function `get_turing_award_recipients(...)` that returns a list of WikiData entities of humans which won the ACM Turing Award. Use the WikiData Api for this task. Hint: use the property `P166` to find humans which received the ACM Turing Award `(Q185667)`.

Task 2: Using the retrieved list from the previous task, get the Wikipedia page content for all recipients if a page exists. Write a function `get_wikipedia_content(...)` that takes as input a Wikidata ID and returns the content of the English Wikipedia page for this entity.

Task 3: Create a dictionary `award_winners` with the following entries: `name`, `intro`, `gender`, `birth_date`, `birth_place`, `employer`, `educated_at`. Fill the property `intro` with the introduction text of the Wikipedia page, enter for all other properties information you retrieved from WikiData. If no information is available enter `None` for the property, if multiple entries are available (e.g. for "employer") save all of them as a list and assign it to the property in the dictionary.

Task 4: Print the name of all award winners in alphabetical order.

Task 5: Provide statistics of the text retrieved from Wikipedia (i.e. `award_winners["intro"]`) in a `pandas.DataFrame` named `award_winners_intro`:

(a) Add five columns to the dataframe with the following names: `winner_name`, `count_words`, `count_sentences`, `count_paragraphs`, and `common_words`.

(b) Fill this dataframe with one award recipient (winner_name) per row and the following data: number of words, number of sentences and number of paragraphs, and the ten most frequently occurring words in the `intro` text saved for this recipient in the dictionary `award_winners`.

(c) Add a sixth column to the dataframe named `common_words_after_preprocessing`, which contains the ten most common words per winner's intro text, excluding stopwords and punctuation. For each intro text, remove English stopwords and punctuation tokens before re-calculating the top-10 frequent words. Hint: consider using the package `nltk`.

(d) **Print** the first ten rows of the final dataframe `award_winners_intro`.

3.2 Sub-activity: Applying NLP operations on the corpus

Next, we apply some basic NLP operations on **all intro texts** of the dictionary `award_winners`.

3.2.1 Stemming

Task 3: Stem all words with the `PorterStemmer`.

(a) First, remove stopwords and punctuation from all intros and save the remaining texts in a list named `intro_words`.

(b) Calculate and **print** the number of unique words in `intro_words`.

(c) Afterwards apply the `PorterStemmer` available in the `nltk.stem`[1] package to stem the words in `intro_words`. Print how many unique words you have left after applying the stemmer.

Task 4: Repeat the previous step with the `SnowballStemmer`. Again print the number of unique words before and after applying the stemmer.

3.2.2 Lemmatization

Task 5: Repeat the process from sub-activity 3.2.1 but this time you will lemmatize the words. Use the `WordNetLemmatizer`, calculate the number of unique words before and after lemmatization, and print them.

3.2.3 Finding synonyms and antonyms

Task 6: For all words of column `common_words_after_preprocessing` of your dataframe, find synonyms and antonyms. For this, use *WordNet*, which is a lexical database for the English language. Import the WordNet reader from NLTK with `from nltk.corpus import wordnet`. Hint: to find synonyms and antonyms, the function `wordnet.synsets()` will be useful.

(a) First, add two new columns to the `award_winners_intro` dataframe: `synonyms`, `antonyms`.

(b) For all words of column `common_words_after_preprocessing` calculate the list of synonyms. Therefore, write a function `get_synonyms()` which takes as its input a list of words and returns a list of all their synonyms. Apply this function to the column `common_words_after_preprocessing` and save the list of synonym words in column `synonyms`.

(c) Repeat the previous step, write a function for antonyms, and extract all antonyms for words in `common_words_after_preprocessing` save the resulting list of words in column `antonyms`.

(d) Print the first ten rows of the dataframe.

[1] https://www.nltk.org/api/nltk.stem.html

3.2.4 Bigrams and trigrams

Next, we will use the intro text to extract bigrams.

Task 7: First tokenize the intro texts you find in `award_winners`.

Task 8: Next, extract all bigrams occurring in the intro text of all winners. Therefore, write a function named `get_bigrams_frequency()` which takes as input a list of intros and returns a dictionary. The keys of the dictionary are the bigrams and the values are the corresponding frequencies (i.e. how often this bigram occurs in the list of intros). Moreover, the bigrams in the dictionary should not contain any punction tokens and stopwords from the NLTK English stopword list.

Task 9: Save the dictionary in a variable named `winners_bigrams`.

Task 10: Print the 15 most frequently occurring bigrams of this dictionary.

3.3 Sub-section: Visualisation

In this final part of the activity you will use the Python package *matplotlib* to create visualisations of your work so far.

3.3.1 Barplots

Task 11: First, create three barplots to compare: the *number of words* (1st plot), *number of sentences* (2nd plot), and *number of paragraphs* (3rd plot) **per winner** in your dataframe `award_winners_intro`. The winner names should be displayed on the x-axis of each chart.

Task 12: All three barplots should be displayed in a single figure, use therefore the `subplots()` function and **display** the resulting figure in your notebook.

Task 13: Next apply the following changes to the generated barplots:

- Change the color of the bars for the 1st barplot.
- Adjust x-axes labels for all barplots by rotating them by 90 degrees clockwise.
- Include the exact count of words, sentences and paragraphs at the top of each bar in all three barplots.
- Order the bars for each plot in ascending order, e.g. the smallest bar is the first one from the left.
- Replace the 3rd barplot by a horizontal barplot with the names displayed on the y-axis. Sort the bars in ascending order from top to bottom. You don't need to rotate any labels for the horizontal barplot.
- Print all three barplots again after applying these changes.

3.3.2 Heatmap

For the next visualisation use the 15 most frequent bigrams of the dictionary `winners_bigrams` and calculate their frequencies per winner.

Task 14: Create a heatmap that fulfills the following conditions:

(a) The winner names are displayed on the x-axis and the 15 different bigrams on the y-axis of the heatmap.

(b) Depending on how often a bigram occurs in the intro text of a winner, the heatmap color ranges from light blue (lowest number of occurrences) to dark blue (highest number of occurrences).

(c) Print the resulting heatmap in your notebook.