Data Engineering Report

When we are given a dataset, it is full of raw, unfiltered, data with no structure or 'rhyme or reason' to it. Prior to use, we need to explore and pre-process it (which may include standardisation), to convert it into a usable format. Discussed in Week 3, Lecture 2.

The first step was choosing what data to use. Within the Github folder, there were many (sub)folders, so I needed to decide which was relevant. It made most sense to plot cases over time, so I used the Time Series folder. The global files covered confirmed cases, deaths, *and* recovered cases (US omitted), so I chose those 3 files.

I considered linking my code to the dataset so that it would always have the newest data (as below), but then I could not be sure that the dataset had not changed, so I had to download it to be able to 'certify' it as sufficiently prepared for use.

```
confirmed_link = 'https://raw.githubusercontent.com/CSSEGISandData/COVID-19/
master/csse_covid_19_data/csse_covid_19_time_series/
time_series_covid19_confirmed_global.csv'
confirmed_df = pd.read_csv(confirmed_link)
```

Data Exploration (Slides 15-20)

Data exploration is defined here as: "*users [exploring] a large data set in an unstructured way to uncover initial patterns, characteristics, and points of interest*". We need to 'look at' our dataset; figure out how the data is structured, see if there are any patterns within it, and try and find the important parts.

Implementation of Data Exploration

As I had the Excel files downloaded, I just looked at them in Excel (quicker and easier, this is what Excel was created to do), however I include code chunks of how I *could* have done this through Python. I began by looking at my Excel files:

```
# Read in Data
confirmed_data = "time_series_covid19_confirmed_global.xlsx"
confirmed_df = pd.read_excel(confirmed_data)
print(confirmed_df)
```

I noted that there were:

- 712 columns.

```
print(confirmed_df.shape[1])
```

- 4 'information' columns, and otherwise columns corresponding to cases per date.

```
print(confirmed_df.columns)
```

- 196 different Countries/Regions in each table.

```
print(len(confirmed_df['Country/Region'].unique()))
```

- Larger countries divided into multiple Provinces/States, resulting in 280 rows overall.

```
print(confirmed_df.shape[0])
```

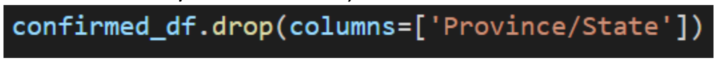
Data Pre-processing (Slides 21-25)

Now that data exploration is complete, we know what data we have, and what we want to do with is, however we need to convert it into a usable format. As defined here, data pre-processing is “*the process of transforming raw data into an understandable format*”.

Implementation of Data Pre-processing

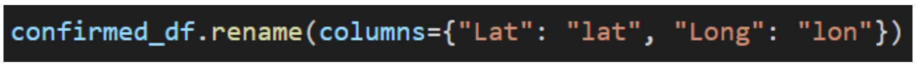
After my data exploration, I noticed a few issues to be addressed:

- **Too many columns**: to comply with the concept of data reduction (in large volumes of data, getting rid of some may make analysis quicker or easier), I removed some columns:
    - I considered only taking data from one day each month so I would have one data point per month, however, I decided that this was not a good choice, as I was plotting cases over time and so the more points the better (one day per month may not fairly/accurately reflect that month’s trend), so I kept them all.
    - The recovered dataset had no data after August 5th 2021, so I removed columns after that date as it did not contribute to analysis.
    - I was not interested in the Province/State column, so I removed it.



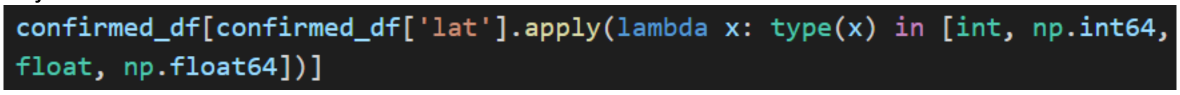
```
confirmed_df.drop(columns=['Province/State'])
```

- **Latitude/Longitude Columns:**
    - I found the *st.map()* function, which can be used to plot data points on a map, but the dataset “must have columns called 'lat', 'lon', 'latitude', or 'longitude'”, while mine were ‘Lat’/’Long’, so I renamed them.



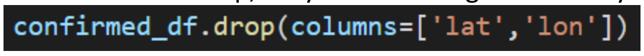
```
confirmed_df.rename(columns={"Lat": "lat", "Long": "lon"})
```

    - When using this function, I received errors about non-numeric data, and then I realised that some latitude/longitude values were blank. If they were for countries of interest, I could have (for example) used the average latitude/longitude for that country. However, this was not the case, and so I just removed those rows.



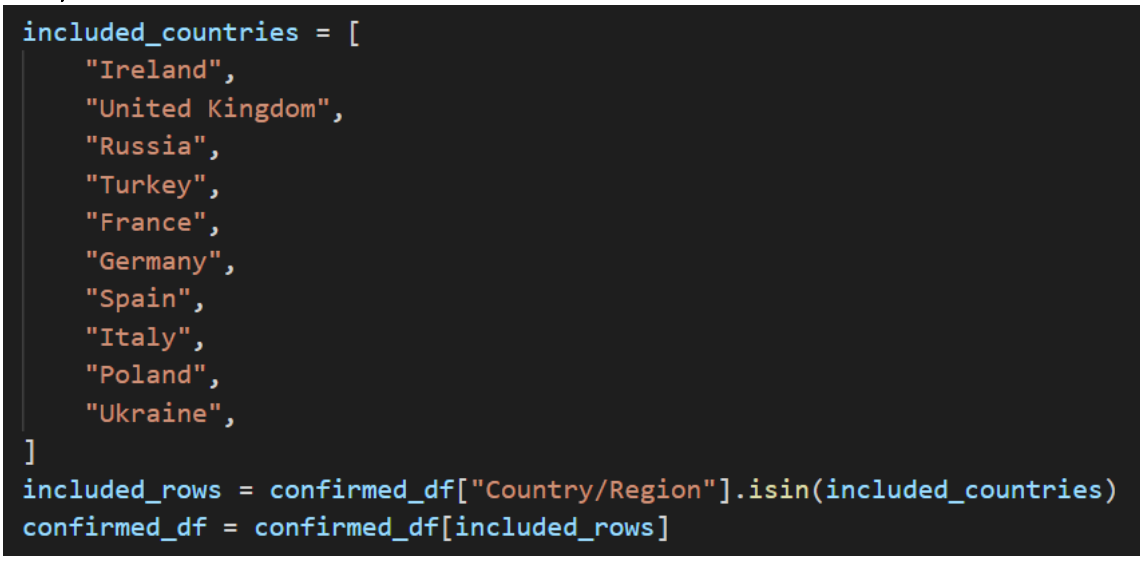
```
confirmed_df[confirmed_df['lat'].apply(lambda x: type(x) in [int, np.int64,
float, np.float64])]
```

    - However, once I had used them for this map, they were no longer necessary and so I removed them.



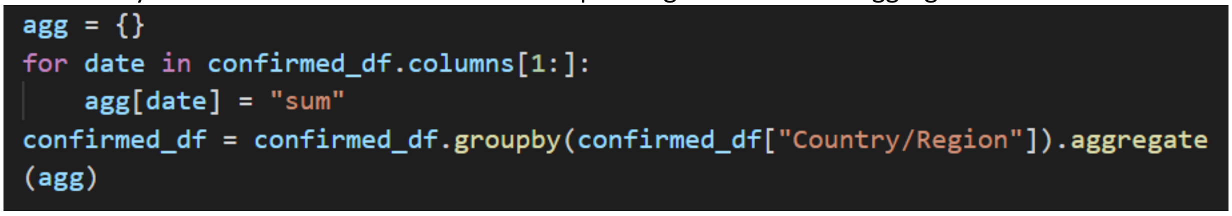
```
confirmed_df.drop(columns=['lat','lon'])
```

- **Too many rows**: similarly, once I had created my map, I had way too many countries to be able to reasonably plot them, and so I decided to use only the 9 worst European Countries (as of 22/12) plus Ireland for my analysis.



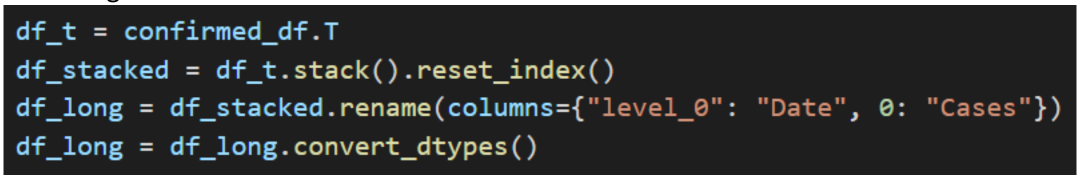
```
included_countries = [
    "Ireland",
    "United Kingdom",
    "Russia",
    "Turkey",
    "France",
    "Germany",
    "Spain",
    "Italy",
    "Poland",
    "Ukraine",
]
included_rows = confirmed_df["Country/Region"].isin(included_countries)
confirmed_df = confirmed_df[included_rows]
```

- After this, I combined Provinces/States per country, so that for each country I only had one row. agg is a dictionary of the dates as indexes and the sum function as their values. Combining *groupby()* and *aggregate(agg)* performs the aggregation of row data in summing each column's values in each row of a country and allocates total value into corresponding column of the aggregated row.



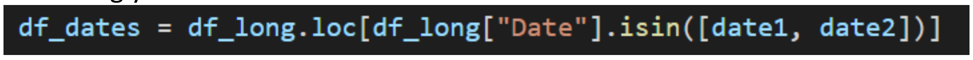
```
agg = {}
for date in confirmed_df.columns[1:]:
    agg[date] = "sum"
confirmed_df = confirmed_df.groupby(confirmed_df["Country/Region"]).aggregate
(agg)
```

- **Date issues**: the dates were in the format month/day/year, while my code interpreted it as day/month/year (e.g., 01/06 as June 1st, actually January 6th), so in Excel I changed the date formats to UK (couldn't get this to work in Python).
- Convert.dtypes: I discovered this function which "[converts] columns to best possible dtypes", and so used it on my final dataset.
- **Pivot longer**: when plotting by date, I needed all of the dates to be a column, and so I had to transform the data set to be 'long' rather than 'wide'.



```
df_t = confirmed_df.T
df_stacked = df_t.stack().reset_index()
df_long = df_stacked.rename(columns={"level_0": "Date", 0: "Cases"})
df_long = df_long.convert_dtypes()
```

- For the subsequent graph showing a before and after comparison, the required dates/columns were subsetted accordingly.



```
df_dates = df_long.loc[df_long["Date"].isin([date1, date2])]
```

Standardisation (Slides 26-27)

Data standardisation is cited here as "*the process of converting data to a common format to enable users to process and analyze it"*, and is a helpful tool to prevent features from being (potentially unfairly) weighted as more important. This is particularly useful when your data set is 1) comprised of data using different measurement units, or 2) a wide range of values.

Implementation of Standardisation

I didn't utilise standardisation as a part of my project.

I could have standardised case numbers, however, as they're all on the same scale (1), this did not seem justified or necessary. If, for example, in one column '5' represented 5 *million* cases, while in another it meant 5 cases, standardisation would have been necessary, however this was not the case.

I wanted it to be very apparent how much drastically higher cases are nowadays than initially, so I wanted the scale to stay accurate, and so the 'wide range of values' (2) criteria also did not seem appropriate.