Practical 06: Data Visualisation

Dr. Todd Jones

t.r.jones@reading.ac.uk

Department of Computer Science

Follow the instructions to complete each of these tasks.

This set of exercises focuses on practicing implementation of data visualisation tools from `pandas`, `matplotlib`, and `seaborn`.

This work is not assessed but will help you gain practical experience for the coursework.

You will have accessed this practical notebook by downloading a `.zip` file containing the relevant datasets. The **relative path** to the data from this notebook should be:
`./data/<dataset_file_name>`

Please refer back to the **Data Visualisation lecture notebook** for internet references to various routines and packages.

Where **additional materials**, not covered in the lecture, are required, they are implemented for you or referenced below.

In [ ]:
```
# You will need these tools
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Histograms

Look at the `./data/faithful.csv` data, which gives a list of eruptions of a geyser in Yellowstone National Park in the USA. The `eruptions` variable gives the temporal length of the eruption, while the `waiting` variable gives the waiting time until the next eruption. One "source" on this very commonly explored dataset: https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/faithful.html

- Explore the data by looking at the **distribution** of the two variables.
    - Which kinds of plots look at distributions?
    - Try out `plt.hist()`, `sns.histplot()`, and `sns.jointplot()`.
- How do they **vary together**?

- What do you notice about the distribution of the individual variables?
- Are they related?

In [ ]:
```
# Simple read of the data
```

In [ ]:
```
# Simple histogram with matplotlib
```

In [ ]:
```
# Use sns.histplot
```

Hong Kong Marine Water Quality

Hong Kong government open data: historical marine water quality

- Measurements of various **water quality** indicators in

different monitoring stations.

- Data is split up into different zones.
- Data is measured at different times in different zones.

Source: https://data.gov.hk/en-data/dataset/hk-epd-marineteam-marine-water-quality-historical-data-en

Data: `./data/marine-historical-2021-en.csv`

Description: `./data/historical_marine_data_dictionary_en.pdf`

Read Data

Read the data and explore its contents.

In [ ]:
```
file = './data/marine-historical-2021-en.csv'
data = pd.read_csv(file)
```

In [ ]:
```
data
```

In [ ]:
```
data.columns
```

In [ ]:
```
data.info()
```

In [ ]:
```
data.describe()
```

Re-Read Data

Read in the data again, this time specifying that the second column be used as the `DataFrame` index and ask `read_csv` to `parse_dates`.

- Use the `index_col` parameter to set which input column should be used to represent the `index` of the `DataFrame`.

In [ ]:
```
data = pd.read_csv(file, index_col=2, parse_dates=True)
```

In [ ]:
```
data
```

Clean Data

Call the `clean_data` function on your `DataFrame` to replace the small value strings with zeros.

In [ ]:
```
# Convert each column to a float
def clean_data(indata):
    '''
    This function will modify the indata DataFrame inplace.

    The HK data has several features that record small numerical values as
    strings that start with "<".

    This code converts data from column 4 onward to floats, replacing
    the small values with zero.
    '''
    for col in indata.columns[4:]:
        if indata[col].dtype == 'object':
            try:
                indata[col] = indata[col].astype(float)
            except Exception as e:
                print(col)
                print('\t', e)

                try:
                    indata[col] = indata[col].where(~indata[col].astype(str).str.startswith('<'
                    indata[col] = indata[col].astype(float)
                except Exception as e:
                    print('ERROR: Our modification did not work.')
                    print(e)
    print('')
    indata.info()
```

In [ ]:
```
# Calling function to clean the data
clean_data(data)
```

`pandas` Line Plots

Use `.plot()` from `pandas` to make a simple line plot showing a time series of `'Dissolved Oxygen (mg/L)'` for the **entire dataset** (all locations).

Some of this plot will look as though there are very linear periods (instead of highly variable). What are these linear features?

- Were any plot labels automatically generated?
- How can you add any missing information?

In [ ]:
```

```

Create a similar plot for `Temperature (°C)`.

Does the result make logical sense? That is, do warmer and colder temperatures occur at appropriate times of the year?

In [ ]:

Customising Plot Elements

Plot a time series of `'Dissolved Oxygen (mg/L)'` for the `DM1` **Station** using `pandas`. Ensure that the y-axis is labelled appropriately.

- You will need to rely on `plt.ylabel()`.

- See if you can also make each individual data point on the line to display as a downward-pointing triangle.
  - https://matplotlib.org/stable/api/markers_api.html

- Can you make the line **green**?

- Can you make the triangles **magenta** with **blue** edges?

- Can you control the **size** of the triangles?

In [ ]:

`seaborn` Violin Plots

Use the **seaborn** `catplot` utility to create a set of **violin plots** describing the distribution of `pH` in each `Water Control Zone`.

Place the **categorical** variable along the x-axis and the **numerical** variable on the y-axis.

Investigate the `rotation` parameter associated with `set_xticklabels` to ensure that each label can easily be read. You might also find the `ha` parameter to provide more aesthetically pleasing results.

To modify the ticks, save the resulting plot object and use `set_` methods on that object.

See: https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.set_xticklabels.html

In [ ]:

`seaborn` Bar Plots

Use the **seaborn** `catplot` utility to create a group of **bar plots**. You will create an array of subplots (`catplot` can do this automagically) where each subplot (i.e., the `col` parameter), corresponding to individual `Water Control Zone` locations, displays the measurement of `'E. coli (cfu/100mL)'` in each month. The plotting routine will consider all entries with the same month name in the same water control zone. The bars will represent the mean measurement for each of these entries, and variability will be represented with a line through the end of the bar.

To organise the data by month, we will need to create a new `'Month'` column in the `DataFrame`. To do this, we will convert the time values in the `index` to **string abbreviations** for calendar months by using the `strftime` function with a suitable **format code**. See:
https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

Appropriate representations will look like:

```
['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov',
'Dec']
```

Try to display these neatly by using the `col_wrap` parameter and with the months displayed in time order from Jan-Dec by using the `order` parameter.

In [ ]:

Comparing Zones

Exploring the Data

We would like to **compare** how certain measurements change in time in different zones. Let's investigate how this might look for a specific measurement in a specific zone.

Create a **timeseries plot** of `'Salinity (psu)'` for `'Victoria Harbour'`. In this plot, **do not display the line** between the data points (e.g., `linestyle=''`). Instead, only show the data points as **circular markers**.

- Ensure your plot is reasonably labeled.

In [ ]:

In the previous plot, it appears that there are many days without measurements and several days for which there are multiple measurements.

Confirm this suspicion by inspecting the `index` entries of the **entire** data set or a subset, such as specifically Victoria Harbour, or specific stations in Victoria Harbour.

- How many entries are present in the dataset?
- How many **unique** entries are present in the `index`?
- How many **duplicate** entries are present in the `index`?

In [ ]:
```
# For the entire dataset
```

In [ ]:
```
# For Victoria Harbour
```

In [ ]:
```
# Report the number of duplicated dates for each station in Victoria Harbour
```

Reducing the Data

Now that we see that some days and stations have multiple measurements, let's **reduce** the total number of data points by averaging out the representation of multiple measurements and multiple stations.

Group the data by `'Water Control Zone'` and `'Dates'`. Then, **average** the grouped data to obtain average measurements for each zone on each date.

Store the result in a new object.

Display the resulting `DataFrame`.

In [ ]:
```

```

Reformatting the Data

In the previous result, you will see that the `index` now has two components: `'Water Control Zone'` and `'Dates'`. If you inspect the index, you will see that it is a `MultiIndex` object with a **hierarchical structure**.

You can use the `.unstack()` method on the `DataFrame` to **pivot** a level of the `MultiIndex` so that it will become the **inner** component of the column structure. - https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.unstack.html

Let's focus on the `'Suspended Solids (mg/L)'` variable. Select this variable from the grouped and averaged data and **unstack** the Water Control Zone. By default, `unstack()` will operate on the *last* level, so you will need to specify the correct level by name or number.

Display the unstacked result. You should see zone names as columns and dates as the indices.

In [ ]:
```

```

Plot the Data

Inspecting the unstacked result, we see that there are many **missing values**. Most of these are the result of the fact that different Water Control Zones were **sampled on different dates**. The unstacked data now shows a row for every date in the suspended solids dataset, and a value is recorded for each zone and date, regardless of whether data was collected on that date.

We are interested in plotting the suspended solids time series for each zone on the same figure (a line for each zone).

Assuming your suspended solids data is now called `ss`, attempt to plot the data with `pandas` as:

```
ss.plot()
```

In [ ]:

Adjust the Plot Legend

One obvious problem is that the **legend** is not in a very good location. We can relocate this by storing the resulting `AxesSubplot` object and modifying its associated legend information.

- `bbox_to_anchor` creates a new "bounding box" according to a tuple of spatial parameters.
  - The first two elements correspond to position (x, y) in relative axis units (0 to 1).
- `loc` specifies the legend location.

Legend info: https://matplotlib.org/stable/api/legend_api.html

Try:

```
ss.plot().legend(bbox_to_anchor=(1.04, 0.5), loc='center left')
```

Note: If you later save this figure, it is likely that the legend will not be fully inside the plot, as it likely extends outside of the original figure plotting region. One way to account for this is with an additional parameter in `savefig()`:

```
plt.savefig('figure.png', bbox_inches="tight")
```

In [ ]:

Investigate the Plot Representation

The other major problem with this plot is that very little of the data is visible. Lines are only drawn between adjacent valid numbers. Everywhere that numbers are separated by NaNs, no line is drawn.

Re-plot the data with out drawn lines, only markers.

In the resulting figure, you should see many more data points than are depicted above.

In [ ]:

Solution #1: `seaborn`

`seaborn`'s `lineplot()` works a bit differently: it will connect data points across the NaN values. Specify the `data` parameter to test this.

- The legend can be repositioned in the same way as for `pandas` plotting.

Solution #2: `matplotlib`

As an alternative to relying on `seaborn`, we could modify the data to make similar figures with `pandas` or `matplotlib`.

One way to do this is to **mask the missing values**.

Let's try this with `matplotlib`. Create a `for` loop over the column names (zones).

In the loop:

- Isolate the values in the column.
- Isolate the index values.
- Create a mask (`True` / `False`) of finite values (non-NaNs).
- Plot a line: masked indices, versus masked values.

After the loop:

- Adjust the legend.
- Add a y-label.

Solution #3: Interpolation

Another option is to **interpolate** between the known data points to **fill in the missing values**.

Let's try this with `seaborn`.

- To **linearly** interpolate the data in the columns, use the `pandas` method `df.interpolate()`.
  - See: https://pandas.pydata.org/pandas-docs/version/2.2.2/reference/api/pandas.DataFrame.interpolate.html
  - Test this without any arguments. **Can you determine why the lines look different?**
  - If you cannot determine what has gone wrong, check the description of the `method` parameter for `interpolate()`.
    - What is happening at the latest times for some of the zones? Why do the lines become horizontal?
  - Once you have sorted the issue, be sure to inspect the interpolated `DataFrame` to clearly understand how it has modified the data.
  - You might explore additional interpolation options. For this data, few others produce reasonable results.

In [ ]:

Correlations and Heatmaps

The `./data/longley.csv` data set contains US macroeconomic variables measured over the middle of the last century. The variables are:

Column	Description
GNP.deflator	GNP implicit price deflator (a measure of inflation)
GNP	Gross National Product
Unemployed	number of unemployed
Armed.Forces	number of people in the armed forces

Column	Description
Population	population with age older than 14
Year	the year (1947 - 1962)
Employed	number of people employed

Explore the data and find which variables appear to be related.

Notes:

- Take a look to the **correlation** function in the following link to review the relationship between variables.
  - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.corr.html
  - Computes a pairwise correlation, comparing all of the columns in a `DataFrame` against one another..
- A good way to visualise correlation comparisons is with a **heatmap**. Please take a look to the following link to learn how to present the correlation between variables with this kind of graph.
  - https://seaborn.pydata.org/generated/seaborn.heatmap.html
  - Plots any rectangular dataset (like that produced by pairwise correlation).
  - Customise the display:
    - `cmap` : https://matplotlib.org/stable/tutorials/colors/colormaps.html (possibly something **diverging**)
    - `square` : Is the data this shape?
    - `annot` : write the data values on the plot
    - `vmin` , `vmax` : What should the range be?
    - `annot_kws={"fontsize":????}` : set the font size in the annotations (and other potential keyword arguments)
    - `linecolor` , `linewidths` : Do you want to add lines of some design?

In [ ]:
```
# Simple read of the data
longley = pd.read_csv('data/longley.csv')
longley
```

In [ ]:
```
```