Doctoral Academy Python **Worksheet 2**
12th June 2018

Overview

This lab will cover more-advanced ways of filtering and processing data, and demonstrate a variety of different plots.

1 Digital breathalyser test data

Throughout this worksheet we will be using one dataset. Specifically, we'll be using digital alcohol breathalyser test data collected by the UK Department for Transport. This is part of the suite of datasets on Road Safety available at *data.gov.uk*. Many different Road Safety datasets are provided (see here for the full list), but we will be focusing on "*Road Safety - Digital Breath Test Data 2011*". You can download this directly from *data.gov.uk* (see here).

The dataset consists of the results of breath tests carried out by police authorities on road users. Police are permitted to carry out breath tests to identify the amount of alcohol recently consumed by a driver. The driver is tested using a digital breathalyser device, which gives the fraction of alcohol in the driver's breath, measured in microgrammes of alcohol per 100 millilitres of breath (microgrammes/mL). This measure is commonly referred to as the *Breath Alcohol Content* or *BrAC*. In the police dataset it is referred to as *Breath Alcohol Level*.

After downloading the dataset and extracting the *csv* file from it, we start in the usual way by importing some relevant modules and loading the dataset:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

breath_data = pd.read_csv('DigitalBreathTestData2011.csv')
```

In addition to this, we're also going to have *pandas* automatically filter out any rows that contain missing information:

```
breath_data = breath_data.dropna()
```

Missing information can occur due to reporting errors or typos when the data was collected.

We can inspect the column names as follows:

```
print(breath_data.columns)
```

Note: `breath_data.columns` *is equivalent to* `breath_data.keys()`*, as we saw in the last lab.*

And we can list all rows with:

```
print(breath_data)
```

2 Selecting and filtering data

We'll now see some more advanced data filtering and manipulation using *pandas*.

We've previously seen how we can look at a single column; for example, to extract the first few rows for the AgeBand column:

```
print(breath_data['AgeBand'][:5])
```

If we want to filter down to more than one column, we need to use double square brackets syntax; for example, to extract the first few rows for only the AgeBand and Gender columns:

```
print(breath_data[['AgeBand','Gender']][:5])
```

If we want to look at a subset of rows for **all** columns, the `iloc` attribute allows us to select a range of rows within the dataframe. For example, to display the first few rows:

```
print(breath_data.iloc[:5])
```

Or to display the first row alone:

```
print(breath_data.iloc[0])
```

Note that *pandas* indexes from zero, so the first row is at index 0.

To display the 20th row and the three rows that follow it:

```
print(breath_data.iloc[19:21])
```

Note that *pandas* ranges are **inclusive** at the end point. This is contrary to the Python convention. So, in *pandas*, `19:21` refers to the rows at indexes 19, 20, and 21.

Exercise 2.1. Display the **last** five rows (with all columns included) of the dataset.

pandas provides us with ways to filter data using comparison operators such as equality, greater/less than, and so on.

For example, to display only the breath tests carried out in November:

```
print(breath_data[ breath_data['Month'] == 'Nov' ])
```

pandas also allows us to use logical operators *and* and *or*, but these are denoted with `&` and `|`.

To display breath tests carried out during either November **or** December we would use `|`:

```
print(breath_data[ (breath_data['Month'] == 'Nov') | (breath_data['Month']
    == 'Dec') ])
```

This gets a bit clumsy if we want to check against more months. A more-elegant approach is to use the `isin` method:

```
print(breath_data[ breath_data['Month'].isin(['Nov', 'Dec']) ])
```

Finally, the & operator corresponds to logical *and*. With this we can filter data down to rows that match multiple requirements; for example, we might want to only display the breath tests carried out on a weekend in the time band *4am-8am*. This corresponds to rows where the *WeekType* is equal to *Weekend* **and** the *TimeBand* is equal to *4am-8am*:

```
print(breath_data[ (breath_data['WeekType'] == 'Weekend') & (breath_data['
    TimeBand'] == '4am-8am') ])
```

To demonstrate the use of *more than/less than* comparisons let's try to answer the following question: How many drivers in the dataset were over the UK's legal drink-drive limit?

UK law states that the legal breath-alcohol limit for a driver is 35 microgrammes/100mL[1]. Let's start by displaying the rows over the limit by using the `>` (greater than) comparison operator:

```
print(breath_data[ breath_data['BreathAlcoholLevel(microg/100ml)'] > 35 ])
```

Since we're going to be using this subset of rows for a few things, let's store them in a variable named `overlimit` so we can refer to them later:

```
overlimit = breath_data[ breath_data['BreathAlcoholLevel(microg/100ml)'] >
    35 ]
```

We can count the number of rows using `len`:

```
print(len(overlimit))
```

[1] http://www.drinkaware.co.uk/check-the-facts/alcohol-and-the-law/drink-driving

So we have 59,912 drivers who were found as over the legal limit in 2011.

> We should be careful with our conclusion that 59,912 drivers were tested as over the limit in 2011. For example, this conclusion assumes that the same driver doesn't appear in the dataset twice. There may be individuals who were caught on multiple occasions. What other caveats should we note with this dataset?

Exercise 2.2. Calculate the number of drivers over the limit as a percentage of all drivers breathalysed. (You should of course use Python to do this, rather than doing it by hand or with a calculator.)

Exercise 2.3. How many breath tests came up with zero alcohol content? In other words, how many tests gave a result of 0 microgrammes/100mL? You'll need to refer to the original data (i.e., `breath_data`) to answer this question.

Recall that `overkill` contains the rows for over-the-limit breath tests. This is a useful starting point for further analysis. For example, if we want to find out how many breath tests were over-the-limit on weekdays vs weekends:

```
print("Weekday: %d" % len(overlimit[overlimit['WeekType'] == 'Weekday']))
print("Weekend: %d" % len(overlimit[overlimit['WeekType'] == 'Weekend']))
```

Exercise 2.4. How many drivers were found to be over the limit in winter (i.e., in December/January/February)?

Exercise 2.5. How does this compare to the other three seasons?

3 Visualisation

In the rest of the worksheet we'll build on the filtering and processing tools we've covered so far to produce a variety of plots to visually present information in the breath test dataset.

3.1 Plotting distributions

We saw in the previous lab how to use a box plot to visualise the distribution of a set of values. For example, we might want find out how the BrAC (Breath Alcohol Content) among over-the-limit drivers varies. Are they usually very close to the limit? Or are they often extremely over the limit? Or are there a minority of people who are extremely over the limit?

To build the box plot of BrAC values for over-the-limit drivers:

```
breath_alcohol = overlimit['BreathAlcoholLevel(microg/100ml)']

fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

ax.boxplot([breath_alcohol])
ax.set_xlabel('')
ax.set_ylabel('Breath Alcohol Content\n(microgrammes/100mL)')
ax.set_title('Breath Alcohol Content of Over-the-Limit Drivers')
ax.set_xticks([1])
ax.set_xticklabels( [''] )
plt.show()
```

In the plot we can see that the box and whiskers are very tightly concentrated around 100. This means that the overwhelming majority of the values are concentrated around here. However, because there is a single extreme outlier between 6000 and 7000 it's difficult to get much detail from the box plot.

To investigate the extreme outlier further, we can get its exact value by finding the highest value in the column:

```
print(breath_alcohol.max())
```

This value is actually an anomaly in the data, as a BrAC above 1,000 is almost certainly fatal. This may have occurred due to a typo during data entry.

We're therefore going to sanitise the data so it only consist of reasonable BrAC values. We'll only keep rows in the data that have a BrAC below 1000:

```
breath_data = breath_data[breath_data['BreathAlcoholLevel(microg/100ml)'] <
    1000]
overlimit = breath_data[breath_data['BreathAlcoholLevel(microg/100ml)'] >
   35]
```

Notice that we've also updated the `overlimit` after carrying out the filtering on `breath_data`. The rest of the worksheet will assume that this sanitisation has been carried out.

We can now re-plot the box plot, repeating the same code we used previously:

```
breath_alcohol = overlimit['BreathAlcoholLevel(microg/100ml)']

fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

ax.boxplot([breath_alcohol])
ax.set_xlabel('')
ax.set_ylabel('Breath Alcohol Content\n(microgrammes/100mL)')
ax.set_title('Breath Alcohol Content of Over-the-Limit Drivers')
ax.set_xticks([1])
ax.set_xticklabels( [''] )
plt.show()
```

The resulting box plot more-effectively conveys the distribution of the BrAC values since it's no longer being skewed by the invalid outlier. We can see that the median BrAC for over-the-limit drivers is 65.

A box plot doesn't tell us everything about the distribution of values, however. By definition, it's a visual depiction of the median and inter-quartile range. Other questions we might ask are: How common are high breath alcohol levels? Do we have many drivers with very low breath alcohol, and then fewer and fewer at higher breath alcohol levels? A **histogram** is better suited to answering these questions:

```
breath_alcohol = overlimit['BreathAlcoholLevel(microg/100ml)']

fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

ax.hist([breath_alcohol], bins=40, normed=False, facecolor='green')

ax.set_xlabel('Breath Alcohol Content\n(microgrammes/100mL)')
ax.set_title('Breath Alcohol Content of Over-the-Limit Drivers')
```

In this histogram, the BrAC values have been split into intervals along the x-axis. The height of the bar above a particular interval tells us how many drivers had a BrAC value in that interval. We can see that there are no bars below 35 microgrammes/100mL. This is because we're only looking at drivers that were over the legal limit (note how we refer to `overlimit` on the first line of the cell).

By finding the highest bar in the plot we can see that most breath tests were around 35. The height of this bar is roughly 14,000, indicating that around 14,000 breath tests were near 35 microgrammes/mL.

Exercise 3.1. So far we've only focused on drivers that were over-the-limit. Let's now look at **all** drivers in the dataset, including those that were within the limit:

(a) Produce a box plot of BrAC values for the breath data, *including* the drivers that were

within the limit. *Hint: use* `breath_data` *rather than* `overlimit`. How is the plot different to its over-the-limit counterpart?

(b) Produce a histogram of BrAC values for the breath data, *including* the drivers that were within the limit. How is the plot different to its over-the-limit counterpart?

Which plot, the box plot or the histogram, is more useful now that we're considering **all** drivers?

3.2 Pie charts

A pie chart allows us to visually depict the relative proportions of a number of quantities. As a basic example, we can use a pie chart to visualise the number of men vs women in the dataset. For now we'll ignore those that were recorded as 'Unknown'.

First, we'll get the number of men and women using *pandas* filtering:

```
num_male = len(breath_data[ breath_data['Gender'] == 'Male' ])
num_female = len(breath_data[ breath_data['Gender'] == 'Female' ])
```

Then, to create the plot:

```
fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

counts = [num_male, num_female]
labels = ['Male', 'Female']
ax.pie( counts, labels=labels )
plt.show()
```

If we wish, we can further customise the plot with a title, custom colours, and shifted ('exploding') segments:

```
fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

counts = [num_male, num_female]
labels = ['Male', 'Female']
colors = ['yellowgreen', 'gold']
explode = [0, 0.1]

ax.pie( counts, labels=labels, explode=explode, colors=colors )

ax.set_title('Gender Comparison')
plt.show()
```

Pie charts are a very poor choice if you are only visualising two proportions. There's rarely any need to use a visualisation for only two proportions. It would have been more effective to simply state the two proportions in text, since it is easy to understand and compare two values without the need of a visual aid.

A more reasonable use of a pie chart is to compare three or more proportions. Let's plot the proportion of breath tests by the time they occurred. This data is given in the TimeBand column. Time bands used in the dataset are '8pm-12pm', '12am-4am', '4pm-8pm', '8am-12pm', '12pm-4pm', '4am-8am', and 'unknown'. As with the gender example, before we can build the pie chart we need to obtain the count for each of the time bands; i.e., the number of breath tests in '8pm-12pm', '12am-4am', and so on.

We could use the *pandas* filtering as we did before; however, repeating this code for each of the time bands would be laborious and inelegant. Instead, *pandas* provides us with a `value\_counts` method that does the work for us:

```
times = breath_data['TimeBand']
counts = times.value_counts()
```

Try the following to see what the function has done:

```
print(counts)  # the count of each time band
print(counts.keys())  # the name of each time band, in order that it
    appears in `counts'
```

Using this result, we can plot the pie chart as follows:

```
times = breath_data['TimeBand']
counts = times.value_counts()

labels = counts.keys()

fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])

ax.pie( counts, labels=labels )
ax.set_title('Number of Breath Tests by Time Band')
plt.show()
```

By looking at the plot, when do most breath tests occur?

Exercise 3.2. Pie charts can be annotated with the numeric amount that each segment corresponds to by giving an `autopct=...` argument to the `pie` method. Update the TimeBand pie chart to include these annotations by using the following line:
`ax.pie(counts, labels=labels, autopct='%1.1f%%')`

Exercise 3.3. Update the TimeBand pie chart so that the segments are ordered by size. *Hint: you'll need to sort the* `counts` *before the chart is plotted.*

Exercise 3.4. Use a pie chart to visualise the ages of drivers in the dataset.

Exercise 3.5. In practice, bar charts are more commonly used than pie charts. With a bar chart it is readily clear which quantities are greatest and least by simply referring to the bar height. Produce a **bar chart** to visualise the ages of drivers in the dataset. Compare this to the pie chart you produced in **Exercise 3.4.** – which chart is more effective?
Hint: Bar chart plotting was covered in the antibiotics section of the previous lab.

4 Extra exercises

Exercise 4.1. Visit the Road Safety Data page on *data.gov.uk* and download the 2012 breath test dataset. Choose one of the plots we covered in this worksheet and re-produce it for the 2012 dataset. What differences, if any, do you notice between the two years?