Week 6

Data Science Laboratory 1

Dr John Evans
j.evans8@herts.ac.uk

University of Hertfordshire UH

Plan for today

Data Manipulation

Identifying and Handling 'Bad' and Missing Data

- Outlier Detection
- Interpolation Techniques

Data Manipulation

- Now that we have understood ways to import data into Python, as well as visualise and perform statistical operations on this data, we can finally start exploring the data.
- We start by looking at time series.
- Lots of data is represented by a time series: stocks and shares, weather, the number of people logged on to Twitter, etc.
- We are going to look at a few ways we can manipulate time series using pandas.

Creating a time series

```
import pandas as pd

minutely = pd.date_range('2020-11-25', periods=3, freq='T')
print(minutely)
#DatetimeIndex(['2020-11-25 00:00:00',
#                '2020-11-25 00:01:00',
#                '2020-11-25 00:02:00'],
#                dtype='datetime64[ns]', freq='T')

hourly = pd.date_range('2020-11-25', periods=3, freq='H')
print(hourly)
#DatetimeIndex(['2020-11-25 00:00:00',
#               '2020-11-25 01:00:00',
#               '2020-11-25 02:00:00'],
#               dtype='datetime64[ns]', freq='H')
```

goes along in minutes

goes along in hours

Creating a time series

```
daily = pd.date_range('2020-11-25', periods=3, freq='D')
print(daily)
#DatetimeIndex(['2020-11-25', '2020-11-26', '2020-11-27'],
               dtype='datetime64[ns]', freq='D')

monthly = pd.date_range('2020-11-25', periods=3, freq='M')
print(monthly)
#DatetimeIndex(['2020-11-30', '2020-12-31', '2021-01-31'],
               dtype='datetime64[ns]', freq='M')
```

goes along in days

goes along in months

What are we doing?

- In these examples we set a start date and define a number of repeat ‘periods’ of a given frequency (given by a letter code).
- Note in the monthly example, the day is ignored, and the return start date is the end of the month.
- The `date_range()` method returns a `DatetimeIndex` object, which is a sequence of dates and times.
- Note the ‘`datetime64`’ datatype; this allows pandas to represent time down to nanosecond resolution.

What are we doing?

- In these examples we set a start date and define a number of repeat ‘periods’ of a given frequency (given by a letter code).
- Note in the monthly example, the day is ignored, and the return start date is the end of the month.
- The `date_range()` method returns a `DatetimeIndex` object, which is a sequence of dates and times.
- Note the ‘`datetime64`’ datatype; this allows pandas to represent time down to nanosecond resolution.
- Rather than defining periods, we could give a start and end date/time and set the frequency. For example, every minute between 2020-11-25 12:39 and 2020-11-26 01:42:

Another example

```
import pandas as pd

rng = pd.date_range('2020-11-25 12:39:00',
                    '2020-11-26 01:42:00', freq='T')
print(rng)
#DatetimeIndex(['2020-11-25 12:39:00',
#               '2020-11-25 12:40:00',
#               '2020-11-25 12:41:00',
#               '2020-11-25 12:42:00',
#               '2020-11-25 12:47:00',
#               '2020-11-25 12:48:00',
#               ...
#               '2020-11-26 01:39:00',
#               '2020-11-26 01:40:00',
#               '2020-11-26 01:41:00',
#               '2020-11-26 01:42:00'],
#              dtype='datetime64[ns]',
#              length=784, freq='T')
```

prints all the minutes in the given range

Plotting a time series using Matplotlib

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.dates import DateFormatter

rng = pd.date_range('2020-11-25 12:39:00',
                    '2020-11-25 12:52:00', freq='T')

# Some random data
data = np.random.normal(0,1,size=len(rng))
```

e.g. data set could be stock prices

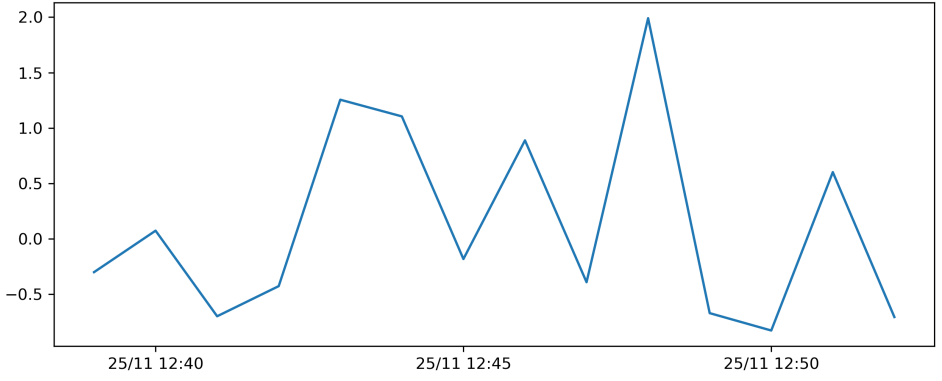
```
# Make it into a series using the time range as index
series = pd.Series(data,index=rng)

fig,ax = plt.subplots(1, 1, figsize=(10,4))

ax.plot(series)
```

creates plot

```
date_form = DateFormatter("%d/%m %H:%M")
ax.xaxis.set_major_formatter(date_form)
```

how the date and time is going to be displayed

The result



What have we done?

- We used the `DateFormatter` function in Matplotlib to customize how we want the *x*-axis date/time labels to be displayed (in this case day/month hour:min).
- One of the most powerful functionalities of handling series that are indexed by `DateTimeIndex` objects is resampling.
- For example, here we use the `resample()` method of the `Series` object and ask to resample to a 5 minute resolution, and then call the `mean()` method.
- This is basically the rolling average on a timescale of 5 minutes:

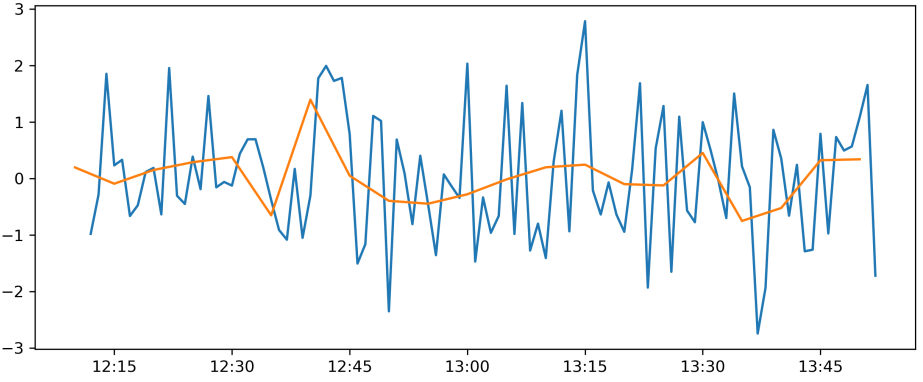
The code

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.dates import DateFormatter

rng = pd.date_range('2020-11-25 12:12:00',
                    '2020-11-25 13:52:00', freq='T')

data = np.random.normal(0,1,size=len(rng))
series = pd.Series(data,index=rng)

fig,ax = plt.subplots(1,1,figsize=(10,4))

ax.plot(series)
ax.plot(series.resample('5Min').mean())

date_form = DateFormatter("%H:%M")

ax.xaxis.set_major_formatter(date_form)
```

taking data from the series, and resamples them (in blocks of 5) and takes the mean

** aka a rolling average

The result



Further additions

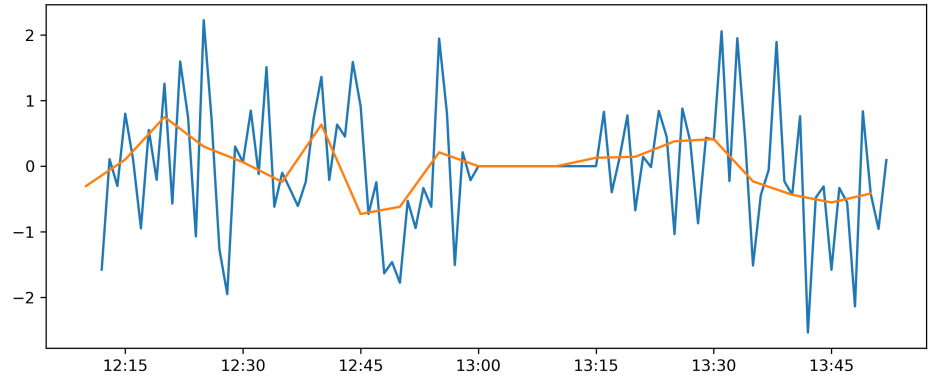
- We can also use datetime strings in indexing and slicing operations, provided the DataFrame or series index is date-based.
- For example, let us set a range of values in the series to zero for a 15-minute interval 13:00 to 13:15:

The code

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.dates import DateFormatter

rng = pd.date_range('2020-11-25 12:12:00',
                    '2020-11-25 13:52:00',
                    freq='T')

data = np.random.normal(0,1,size=len(rng))
series = pd.Series(data,index=rng)

series['2020-11-25 13:00:00':'2020-11-25 13:15:00'] = 0

fig,ax = plt.subplots(1,1,figsize=(10,4))

ax.plot(series)
ax.plot(series.resample('5Min').median())

date_form = DateFormatter("%H:%M")
```

taking a slice between 13:00 and 13:15 and replacing the data values with 0

The result



Exercise: Explore the functionality of pandas time series handling and manipulation functionality referring to the documentation

`https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html`.

Note that pandas can handle things like different timezones, business days, holidays, and daylight savings times!

Example continued

Question: Is there such a thing as bad data?

Example continued

Question: Is there such a thing as bad data?

- This is a question which raises a lot of discussion. As such, for now we will gloss over this point and instead note that, in some cases, we may need to curate our data for one reason or another.
- The two key things we might have to deal with are cases of ‘outliers,’ and the estimation of the values of missing data (or data that has been rejected for one reason or another).

Outlier detection

An outlier is something different from the normal.

- The definition of an ‘outlier’ is vague, but in general if we can say that a given sample (a data set) is described by some probability distribution, then an outlier is a data point that has a very low probability of being selected at random.

Outlier detection

- The definition of an ‘outlier’ is vague, but in general if we can say that a given sample (a data set) is described by some probability distribution, then an outlier is a data point that has a very low probability of being selected at random.
- Outliers could arise in a measurement or observation due to faulty equipment producing an anomalous value, or errors in data entry (e.g. missing or adding an extra zero or putting the decimal in the wrong place), or for any number of other reasons.
- Outliers could mislead us if we do not handle them correctly.
- For example, an anomalously high value in a set of values will cause the mean value to be biased high.
- Generally we want to identify outliers and either remove them from our analysis, or exclude them from the sample entirely.

Methods to identify outliers

- We will examine two methods to identify outliers.
- The first assumes that the probability distribution from which a given sample is drawn can be described by a Normal distribution.
- Earlier, we saw how the probability of drawing a given value from a Normal (Gaussian) distribution scales with σ – the number of standard deviations the value is from the mean of the distribution.
- Refer back to the notes for the maths, but let us see how we could use this idea to identify outliers in Python.

The code

```
import numpy as np

# Draw 100 samples from a Normal distribution
# with mean mu and std. dev. sigma

mu = 1.3
sigma = 0.5

sample = np.random.normal(0,1,100)

# Define a threshold in units of sigma
thresh = 3

# Obtain a Boolean array where outliers are identified with True
outliers = np.abs(sample-mu)>=(thresh*sigma)
```

mean

standard deviation

be between +/- 3 sigma

| sample - mean | >= threshold * standard deviation

What are we doing here?

- We subtract the mean from the sample and also calculate the absolute values, as we are interested in outliers in either tail.
- The threshold we set to identify outliers is somewhat arbitrary and depends on the situation at hand. A higher threshold value will be less conservative (that is, we will keep more of the sample). A lower threshold will be sure to remove outliers, but might also flag good data with genuinely large deviations from the mean.
- This is why it is always a good idea to plot your data and explore it before blindly applying such manipulation.

The Z-score

- Related to this idea is the so-called *Z*-score (or standard score), which is essentially just a measure of how many standard deviations a particular sample x is away from the mean:

$$Z = \frac{x - \mu}{\sigma}$$

Z = (sample - mean) / standard deviation

Tukey's Fences

- Another example is more general to any given distribution, and is called Tukey's[1] Fences.
- It uses the interquartile ranges, such that outliers can be identified as follows:

$$x_i < Q_1 - k(Q_3 - Q_1)$$
$$x_i > Q_3 + k(Q_3 - Q_1),$$

  where k is a positive value.
- Tukey suggested $k = 1.5$ identifies outliers, and $k = 3$ identifies data that is really 'far out'.
- In Python we could write a simple function to trim an array based on Tukey's Fences:

[1] Named after the mathematician John Wilder Tukey (1915–2000).

The code

```
import numpy as np
```

using functions

```
def fences(x, k=1.5):
    # Calculate quartiles
    Q1,Q3 = np.percentile(x,[25,75])
    # Return a compressed array of the values within the fences
    return np.compress(np.greater(x, Q1-k*(Q3-Q1)) &
                       np.less(x, Q1+k*(Q3-Q1)), x)
```

Interpolation techniques

- Interpolation is simply an estimation method to be used where we are missing an observation, or set of observations.
- For example, perhaps we are monitoring some parameter over time, such as rainfall or temperature. Our instrument works fine for a few weeks, but then suffers a malfunction for a day and fails to record any data before being fixed and record as normal.
- Interpolation allows us to estimate that missing data.

Interpolation techniques

- Interpolation is simply an estimation method to be used where we are missing an observation, or set of observations.
- For example, perhaps we are monitoring some parameter over time, such as rainfall or temperature. Our instrument works fine for a few weeks, but then suffers a malfunction for a day and fails to record any data before being fixed and record as normal.
- Interpolation allows us to estimate that missing data.
- Another use of interpolation is to attempt to sample a measurement onto a finer grid.
- Using the weather example again, we might want to use the daily measurements to predict what is happening on an hourly basis.

Nearest neighbour interpolation

- The simplest form of interpolation is called **nearest neighbour interpolation**.
- Here we simply set the missing value to the value that is closest to it in the parameter space.

Nearest neighbour interpolation

- The simplest form of interpolation is called **nearest neighbour interpolation**.
- Here we simply set the missing value to the value that is closest to it in the parameter space.
- So, suppose we had a measurement of the room temperature taken on the hour, every hour, and we wanted to estimate what the room temperature was at – say – 16:29, then we would use the nearest measurement, taken at 16:00. Obviously this is a pretty coarse approach because if we estimate the point just a few minutes later at 16:31, the nearest neighbour is at 17:00!

Nearest neighbour interpolation

- The simplest form of interpolation is called **nearest neighbour interpolation**.
- Here we simply set the missing value to the value that is closest to it in the parameter space.
- So, suppose we had a measurement of the room temperature taken on the hour, every hour, and we wanted to estimate what the room temperature was at – say – 16:29, then we would use the nearest measurement, taken at 16:00. Obviously this is a pretty coarse approach because if we estimate the point just a few minutes later at 16:31, the nearest neighbour is at 17:00!
- A more sophisticated approach is to take into account any trends in the data by performing **linear interpolation**. Consider two points (x_1, y_1) and (x_2, y_2). The x values could represent time (e.g. number of days since January 1st) and the y values could represent some observation (maybe rainfall). We can interpolate between the points by considering the straight line ($y = mx + c$) that connects the points:$m = (y_2 - y_1)/(x_2 - x_1)$, $c = y_1 - mx_1$

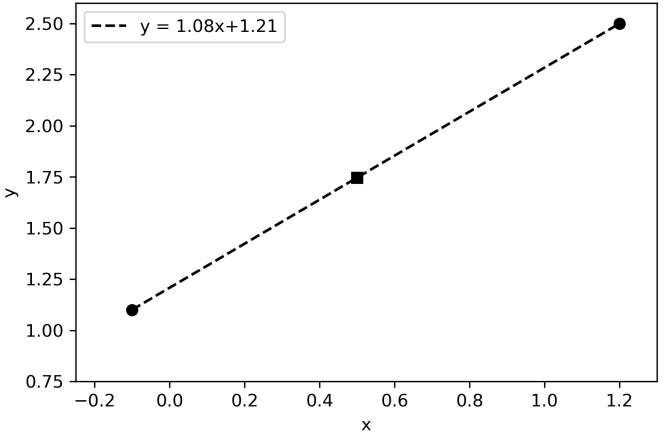
In Python...

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import interpolate

# Some points
x1 = -0.1
x2 = 1.2
y1 = 1.1
y2 = 2.5

plt.plot(x1,y1,linestyle='none',marker='o', markerfacecolor='black',
         markeredgecolor='black')
plt.plot(x2,y2,linestyle='none',marker='o', markerfacecolor='black',
         markeredgecolor='black')

# The equation of a straight line
m = (y2-y1)/(x2-x1)
c = y1 - m*x1
```

In Python...

```
# Some test point between x1 and x2
x = 0.5
y = m*x + c

plt.plot(x,y,linestyle='none',marker='s',
         markerfacecolor='black',
         markeredgecolor='black')


xs = np.linspace(x1,x2,100)
plt.plot(xs,m*xs+c,linestyle='dashed',
         color='black',label='y = %.2fx+%.2f'%(m,c))

plt.axis((-0.25,1.3,0.75,2.6))
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

The result


y = 1.08x+1.21
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
y
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2
x

Ordinal attribute

- As we will soon see, we do not actually have to do this by hand!
- Linear interpolation is the simplest of the polynomial interpolations. In general, we can define a polynomial function of order n and so linear interpolation is the case where $n = 1$.

Ordinal attribute

- As we will soon see, we do not actually have to do this by hand!
- Linear interpolation is the simplest of the polynomial interpolations. In general, we can define a polynomial function of order n and so linear interpolation is the case where $n = 1$.
- More sophisticated still is **cubic spline** interpolation.
- A spline is a function that is made up 'piecewise' from a set of polynomials, and provides finer control of the interpolation in cases where a linear interpolation fails to capture the variation of the data on the scales of interest.
- We can use the `scipy.interpolate` package to implement interpolation:

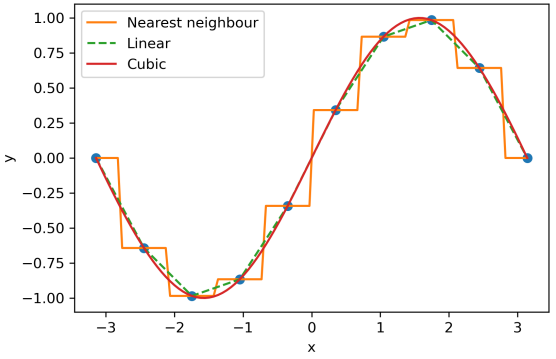
The code

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import interpolate

# Some poorly sampled data
x = np.linspace(-np.pi,np.pi,10)
y = np.sin(x)

# Plot it
plt.plot(x,y,linestyle='none',marker='o')
plt.xlabel('x')
plt.ylabel('y')
```

The code

```
# Set up NN, linear and cubic interpolation
nn_interp = interpolate.interp1d(x,y,kind='nearest')
lin_interp = interpolate.interp1d(x,y,kind='linear')
cub_interp = interpolate.interp1d(x,y,kind='cubic')

# Define a finer grid
x_i = np.linspace(-np.pi,np.pi,100)

# Use the intepolation functions and plot
plt.plot(x_i,nn_interp(x_i),
         label='Nearest neighbour')
plt.plot(x_i,lin_interp(x_i),
         linestyle='dashed',
         label='Linear')
plt.plot(x_i,cub_interp(x_i),
         linestyle='solid',
         label='Cubic')

plt.legend()
```

The result



Exercise: Have a go at replicating the above code, but adding some random Gaussian noise to the points. Then have a look at the effect.

Summary

- We have seen how to create a time series.
- We have some ways to detect outliers.
- We have seen some interpolation techniques.