3_knn

January 22, 2024

1 k-NN's: k-Nearest Neighbours

variationalform https://variationalform.github.io/

Just Enough: progress at pace https://variationalform.github.io/

https://github.com/variationalform

https://www.brunel.ac.uk/people/simon-shaw.

This document uses python

and also makes use of LaTeX

in Markdown

1.1 What this is about:

You will be introduced to …

- The penguins data set, data frames, data selection
- Data engineering: *mean imputation*, and dropping unknowns
- Data bifurcation and trifurcation; calibration; tuning and **hyperparameters**
- k-Nearest Neighbours - classifying by *nearness*
- using the `KNeighborsClassifier` from `sklearn.neighbors`
- Confusion Matrices

The idea is that by using vectors to represent our data set, we can classify a new data point by finding the nearest data point to it for which the class is known. We then assign the new point with the same class.

As usual our emphasis will be on *doing* rather than *proving*: *just enough: progress at pace*.

1.2 Assigned Reading

For this worksheet you should read pages 19 - 25 of

- MLFCES: Machine Learning: A First Course for Engineers and Scientists, by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön. Cambridge University Press. http://smlbook.org.

The pages leading up to Page 19 are also highly recommended as an overview of concepts, purpose and uses of Machine Learning.

2 Penguins: An Example Data Set

We bring in our standard imports and then recall the data sets that are available in seaborn. We'll be using the *penguins* data.

[ ]:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
import pandas as pd
import seaborn as sns
```

[ ]:

```
# See, for example,
#   https://github.com/mwaskom/seaborn-data
#   https://blog.enterprisedna.co/how-to-load-sample-datasets-in-python/
sns.get_dataset_names()
```

2.1 Some Data-Engineering

As we have seen, there are a lot of data sets here that can be used to demonstrate various aspects of, and techniques in, Machine Learning and Data Science, and we'll look at a few of them - and others - as we progress.

To start with though we'll be working with the penguins data set. Before we do any machine learning we are going to have to do some **data cleaning**, see e.g. https://en.wikipedia.org/wiki/Data_cleansing, to remove some undefined values.

This shouldn't be confused with https://en.wikipedia.org/wiki/Feature_engineering.

Let's grab the penguins data and see what is in it. We load it into a data frame called `dfp`, as in *data frame for penguins*, and then look at the head of the table - the first few rows.

[ ]:

```
dfp = sns.load_dataset('penguins')
dfp.head()
```

Let's look at the shape of the data set - how many rows and columns does it have?

[ ]:

```
num_rows, num_columns = dfp.shape
print('number of data points (or observations) = ', num_rows)
print('number of features (or measurement) = ', num_columns)
```

So, the data set contains 344 rows and seven columns. Each row corresponds to a single penguin, and for each row each column corresponds to a feature of that penguin. We can see its species, the island it was found on, its bill length, bill depth, and flipper length - all in millimetres, its body mass in grams, and its gender.

We can also see `NaN` values in row 3. That's the fourth row - be careful of this, indexing starts at zero. This stands for *Not a Number* and means that we can't use those values as they stand. We don't know why they are there - paerhaps the data got corrupted. It's a fact of life though that data sets are often a bit messy with wrong, missing or corrupted values. We'll see a couple of ways to deal with these instances below.

We haven't listed every row - just the `head` of the data table. Another way to visualize these data is to use a scatter plot.

See e.g. https://seaborn.pydata.org/generated/seaborn.scatterplot.html

[ ]:

```
sns.scatterplot(data=dfp, x="body_mass_g", y="bill_depth_mm", hue="species",␣
 ↪style="sex")
```

If that looks a little cramped you can control the size like this:

[ ]:

```
plt.figure(figsize=(8, 8))
sns.scatterplot(data=dfp, x="body_mass_g", y="bill_depth_mm", hue="island")
```

When we issued the command `dfp.head()` above we got to see the top of the table. We can also see the bottom like this:

[ ]:

```
dfp.tail()
```

This has given us two species, *Adelie* and *Gentoo*, but from the plots above we know there is also a third: *Chinstrap*.

We can also see from the *head* and *tail* functions that there are two islands, *Torgersen* and *Biscoe*, and that - from the plots - there is a third, *Dream*.

How could we find these without having to plot the data? Well, we could look at the whole table with this command:

`print(dfp.to_string())`

Try it in the cell below - uncomment it and execute the cell. It's a bit messy (and what if we had millions of rows?).

Now re-comment it and execute the cell again to clear that very long output.

[ ]:

```
# print(dfp.to_string())
```

A simpler way is to ask for all the unique entries in the *species* column, and
in the *island* column:

[ ]:

```
dfp.species.unique()
```

[ ]:

```
dfp.island.unique()
```

2.2 Summary

We have seen that three species are documented on three Antarctic islands.

We have also seen that some values are undefined: `NaN` stands for *Not a Number*. This may indicate that the data was not captured reliably.

We can see how many rows contain undefined values with this command:

[ ]:
```
dfp.isna().sum()
```

There are at least eleven - and there could be 11+2+2+2+2. Let's find them.

In the following `axis=1` tells python that we want to find **rows** with `NaN` in, as opposed to **columns**.

[ ]:
```
dfp[dfp.isna().any(axis=1)]
```

We can get a list of the row index numbers like this:

[ ]:
```
NaN_rows = dfp[dfp.isna().any(axis=1)]
print(NaN_rows.index)
```

And we can use these as an alternative to the `axis=1` command above:

[ ]:
```
dfp.loc[NaN_rows.index]
```

2.2.1 Data Engineering - our first method

One way to deal with missing values like this is to simply fill them with 'reasonable' values. For example, we can replace the numerical values with the mean, or average, of that feature, and replace categorical values with just one of the possible categories.

For example, let's use the mean for numerical values and treat all missing genders as *Female*.

[ ]:
```
# from https://datagy.io/pandas-fillna/
dfp1 = dfp.fillna({'bill_length_mm'   : dfp['bill_length_mm'].mean(),
                   'bill_depth_mm'    : dfp['bill_depth_mm'].mean(),
                   'flipper_length_mm': dfp['flipper_length_mm'].mean(),
                   'body_mass_g'      : dfp['body_mass_g'].mean(),
                   'sex': 'Female'})
```

Mean Imputation Replacing a missing numerical feature value with the mean of the known feature values in this way is called **imputing the mean**. It is easy to implement - just one line above - but you should be aware that it corrupts the original data set.

- On the upside this process maintains the sample size
- On the downside it (probably) alters some statistical properties of the data (the unknown variance, for example).

As an analyst you would be responsible for taking a decision as to how to deal with missing values. You may not be the only one involved in that decision.

We can compare the old and new data frames just to check this worked as expected.

[ ]:
```
# Here is the new one with the NaN's replaced - or engineered out
dfp1.loc[NaN_rows.index]
```

[ ]:

```
# Here is the old one with the NaN's
dfp.loc[NaN_rows.index]
```

It is always good practice to check your work. This can be challenging when dealing with large data sets because you can't keep printing them out and checking every item to make sure that no errors have been introduced.

One way to make sure that these commands didn't do something unexpected behind the scenes is just to plot each data set and make sure they look the same.

For example:

[ ]:

```
sns.scatterplot(data=dfp, x="body_mass_g", y="bill_depth_mm")
```

[ ]:

```
sns.scatterplot(data=dfp1, x="body_mass_g", y="bill_depth_mm")
```

Alternatively, the `describe()` function prints summary statistics. These should be the same for each.

Below we see how this works. What do you think? Is everything broadly OK with our data set?

Can you explain the differences?

[ ]:

```
dfp.describe()
```

[ ]:

```
dfp1.describe()
```

2.2.2 Data Engineering - our second method

In the method above we just replaced missing values with (hopefully) *nearby* ones.

On the other hand, if we have a lot of data and are able to live with a little less of it then we can just drop the data items (rows) that contain one or more undefined values.

> **THINK ABOUT**: what could go wrong?

For example: let's recall the rows with `NaN` entries and then total up how many there are in each column, and in total:

[ ]:

```
dfp.loc[NaN_rows.index]
```

[ ]:

```
dfp.isna().sum()
```

We could have written `dfp.isna().sum(axis=0)` to insist that we are counting down columns here, but that's the default so the `axis=0` isn't needed.

We can see that there are no more that two `NaN` values in the third to sixth columns, but eleven in the last, the seventh, column.

NOTE: the digit in the left most column is just the index of the column - it is not considered part of the data set.

So, given that we have 344 data points (penguins), it looks like we can afford to drop these bad data rows from the set. We can do it like this:

[ ]:
```
dfp2 = dfp.dropna()
```

Let's compare...

[ ]:
```
dfp
```

[ ]:
```
dfp2
```

It looks fine - the `NaN` values have disappeared from the newly engineered dataset. We can check, as above, by counting how many `NaN`'s are found in the new data set:

[ ]:
```
dfp2.isna().sum()
```

On the other hand, the index values in the left most column are off. There is no **3** for example. We can reset them with the `reset_index()` function but we have to make sure we drop the original indices otherwise they will persist.

[ ]:
```
# don't do this - you'll just a column of old and useless index labels.
# dfp2 = dfp2.reset_index()
# instead reset the index and drop the original index column
dfp2 = dfp2.reset_index(drop=True)
```

[ ]:
```
dfp2
```

Now we have a clean data set with no false values introduced, with no undefined entries, and with consecutive labelling down the left.

Visualization Data sets are often much too large to be able to effectively work with them in tabular form. Visualization is then more useful.

Let's pause to explore a few visuals of our cleaned-up data set.

[ ]:
```
sns.scatterplot(data=dfp2, x="bill_length_mm", y="bill_depth_mm", hue="species")
```

[ ]:
```
sns.scatterplot(data=dfp2, x="body_mass_g", y="flipper_length_mm",␣
 ↪hue="species")
```

[ ]:
```
sns.pairplot(dfp2, hue='species')
```

[ ]:
```
# lots of options for the above. See
# https://seaborn.pydata.org/generated/seaborn.pairplot.html
sns.pairplot(dfp2, corner=True, hue='species', height=1.5)
```

[ ]:
```
g = sns.pairplot(dfp2, diag_kind="kde", hue='species')
g.map_lower(sns.kdeplot, levels=4, color=".2")
```

Further Exploration of the Data Set So far we have loaded the data, and operated on it row by row as well as plotted various views of the data.

Let's look now at how to manipulate the data set at a lower level, and see how we might separate out clusters of data - data items that each share a common feature.

Recall, this is what our set contains...

[ ]:

```
dfp2.head()
```

We can see how the species form almost distinct clusters with the following plot.

[ ]:

```
sns.scatterplot(data=dfp2, x="bill_length_mm", y="bill_depth_mm", hue="species")
```

We can access the column of `species` data using square brackets like this

`dfp2['species']`

This refers to every row - with lots of repeated values. In fact they wont all get printed out.

[ ]:

```
dfp2['species']
```

We can squeeze out the repeats into just one uniquely occuring feature value like this...

[ ]:

```
dfp2['species'].unique()
```

This tells us that there are three unique species. We knew this from the plots - but that was a human taking a look. This method allows the code to determine the same information without human intervention.

Creating Data Subsets It is sometimes useful to be able to separate out the data subsets, by a given feature value. If we choose to separate by 'species' then this command

`dfp2.loc[ dfp2['species'] == 'Adelie' ]`

will give us back a new data frame that just contains the Adelie penguin data. It does this by using square brackets and double equals so that this statement,

`dfp2['species'] == 'Adelie'`

evaluates to **true** if, for a given row, the species feature is *Adelie*. Then

`dfp2.loc[ ? ]`

keeps only those rows for which the question mark is *true*. We can assign these rows to a new data frame.

This means that we can create three data subsets - one for each species - as follows...

[ ]:

```
dfA = dfp2.loc[dfp2['species'] == 'Adelie']
dfC = dfp2.loc[dfp2['species'] == 'Chinstrap']
dfG = dfp2.loc[dfp2['species'] == 'Gentoo']
```

Using `matplotlib` to plot the clusters separately We can use `plt.scatter` to plot scatter plots directly in `matplotlib` as below. First we create arrays (vectors if you like) of values, and then we plot them in 2D.

[ ]:

```
blA=np.array(dfA['bill_length_mm'].tolist())
bdA=np.array(dfA['bill_depth_mm'].tolist())
plt.scatter(blA,bdA,color='blue')

blC=np.array(dfC['bill_length_mm'].tolist())
bdC=np.array(dfC['bill_depth_mm'].tolist())
plt.scatter(blC,bdC,color='orange')

blG=np.array(dfG['bill_length_mm'].tolist())
bdG=np.array(dfG['bill_depth_mm'].tolist())
plt.scatter(blG,bdG,color='green')
plt.xlabel('bill_length_mm')
plt.ylabel('bill_depth_mm')
plt.legend(['Adelie', 'Chinstrap', 'Gentoo'],loc='lower right')
```

Interpreting the plot. We can get some statistics by using `describe` - as we have seen before. By comparing the means, below, with the plot above we can check that all is as it should be.

Finding short cut ways to sanity check your working like this is useful.

Here `dfA` is plotted in blue, and we can check that the means look reasonable given the axis labelling.

[ ]:

```
dfA.describe()
```

If you are interested in the arrays that we created in order to do these plots you can take a look at them like this.

[ ]:

```
blA
```

These are `numpy` arrays. There are a number of ways that you can select out just a subset of an array by using square brackets with `slicing`.

For example, we can look at the third to fifth entries like this:

`blA[2:5]`

Indexing starts at zero, hence the `2`. The `5` denotes the first index that is *not used*. This is confusing so watch out for it.

[ ]:

```
blA[2:5]
```

And we can look at all entries except the last five like this:

[ ]:

```
blA[:-5]
```

Let's look now at how we can interrogate our three smaller data subsets.

Here are two ways to determine the number of rows in each.

First, using `shape[0]`...

[ ]:

```
print('number of rows in dfA = ', dfA.shape[0], '; in dfC = ', dfC.shape[0], '␣
↪and in dfG = ', dfG.shape[0])
```

And second using the fact that shape provides a list of two values, and we can ignore the second with `_`...

[ ]:

```
rA, _ = dfA.shape; rC, _ = dfC.shape; rG, _ = dfG.shape
print('number of rows in dfA = ', rA, '; in dfC = ', rC, ' and in dfG = ', rG)
```

Each of these can be used to determine how many of each species there are, because there is one row for each penguin in each data subset.

3 k-NN's - developing intuition

We can now look at the k Nearest Neighbours, or k-NN, method for classification of data.

The setting we assume at the outset is that we have a 'training set' of data such that each row of the data set corresponds to one observation.

Moreover, in each row there are numerical features which can be organized into a vector, $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$, and a label, y, which is categorical.

There may be other numerical and categorical data that we choose not to use.

We imagine plotting these data points in n-dimensional space (hard to imagine when $n > 3$, which is why the abstraction of mathematics is so useful), and we imagine them being coloured according to the value of the label y.

In the example above we had

$$\boldsymbol{x} = (\texttt{bill\_length\_mm}, \texttt{bill\_depth\_mm})^T \tag{1}$$

$$y = (\texttt{Adelie}, \texttt{Chinstrap}, \texttt{Gentoo})^T \tag{2}$$

and we coloured the labels as blue, orange or green.

Now imagine that a field researcher reports in some new measurements for a penguin, and that we want to classify its species based only on those measurements.

The idea is to plot the new measurements and see which cluster of like colour they are closest to. This closest cluster (colour) is then used to assign the species to that new measurement.

Let's see a dummy run of this in a picture.

In the diagram below we pretend that we only have the first twenty rows of each of the data subsets. We plot them as coloured dots, just as above.

Then we pretend that we get three new observations. For illustration purposes we take the entries from the fourth from last position in each data set.

But in the ***REAL WORLD*** we would be expecting new data to be arriving ***UNSEEN*** from the field.

We plot these 'new observations' with a cross.

[ ]:

```
# plot first twenty rows of each as coloured dots.
plt.scatter(blA[0:20],bdA[0:20],color='blue')
plt.scatter(blC[0:20],bdC[0:20],color='orange')
plt.scatter(blG[0:20],bdG[0:20],color='green')
plt.legend(['Adelie', 'Chinstrap', 'Gentoo'],loc='lower right')
plt.xlabel('bill_length_mm')
plt.ylabel('bill_depth_mm')

# pick out the data item fourth from the end in each
indx = -4
# and plot each as a cross
plt.scatter(blA[indx],bdA[indx],color='blue', marker='x', s=500)
plt.scatter(blC[indx],bdC[indx],color='orange', marker='x', s=500)
plt.scatter(blG[indx],bdG[indx],color='green', marker='x', s=500)
```

We carry out the classification as follows:

1. The green cross is quite central in the green, Gentoo, cluster and so we can classify this new observation as a Gentoo penguin.
2. The blue cross isn't that central in the blue cluster, but on the other hand it is far away from the yellow and green clusters and so we can safely classify this observation as an Adelie penguin.
3. The yellow cross presents us with more of a dilemma though. A careful look suggests that it is slightly closer to the yellow cluster than the blue and so, on that basis, we would probably choose to classify that penguin as a Chinstrap.

Any comments, thoughts, questions? The first two steps seem safe, and justifiable. They are *explainable*. The third less so. We can see that the yellow cross corresponds to a fairly typical bill depth for an Adelie.

- So is it a Chinstrap?
- We can also see that Adelie penguins have bill lengths that straddle the value indicated by the yellow cross.
- So should the yellow cross observation be classified as a Chinstrap?
- We see here that the issue of **explainability** can be vexed.
- If we had more data the yellow cross might become obviously a Chinstrap,
- Or it might be obvious that it is an Adelie.

Explainability may or may not matter. But it is increasingly becoming a hot topic in data science.

Suppose your pension fund invested everything in a new tech venture that was going to design batteries with infinite life. It will fail of course.

If this venture was suggested by an Artificially Intelligent agent powered by machine learning algorithms then the pension company directors wont be able to explain their reasoning if the underlying data science was not explainable.

This is hardly realistic, but explainability is a big and important deal in areas like finance and investing, and in medical diagnosis, to name but two. The reasons for its importance are obvious.

4 k-NN's - the mathematical details

We index each data point in the training set with a subscript. So we have the feature vectors $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{x}_3$, ... Each of these has a label, y_1, y_2, y_3, ...

These are the coloured dots above. The positions are the features. The colours are the labels.

We now get a new observation, $\boldsymbol{x}^*$ and we want to classify it - we want to apply a label to it using the data from the training set.

The mathematical version of the process we followed above was to determine the distance between $\boldsymbol{x}^*$ and each $\boldsymbol{x}_i$ using

$$\|\boldsymbol{x}^* - \boldsymbol{x}_i\|_2 \qquad \text{(recall: the Euclidean, Pythagorean or } \ell_2 \text{ norm).}$$

We then to choose the value i such that this distance is a minimum. The label, y_i, corresponding to that particular i is then assigned to the new observation $\boldsymbol{x}^*$.

4.1 Cross-Reference to the Assigned Reading

You were recommended to read pages 19 - 25 of

- MLFCES: Machine Learning: A First Course for Engineers and Scientists, by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, Thomas B. Schön. Cambridge University Press. http://smlbook.org.

More details on this are given there, in paticular:

- the use of k-NN for regression as well as classification.
- the use of more than one 'nearest' neighbour - see which cluster 'wins' a vote.
- notes on how to choose the number of neighbours, and 'overfitting'.
- the importance of normalizing the inputs

Also of importance, but not mentioned in the book, is the choice of norm. We referred to the Euclidean or Pythagorean norm above, but we could just as easily have chosen any of the other p norms that we discussed when we reviewed the material on vectors.

4.1.1 Hyperparameters

In the discussion above we just touched upon the important issue of picking *hyperparameters.* These are values and choices that need to be specified to the algorithm, the code, prior to the machine learning phase.

In the above we mentioned that we need to choose:

- k - the number of nearest neighbours to search for.

- *p* - the choice of norm to use to measure distance, *nearness*.

These are *human* choices: the *hyperparameters* are not learned from the data, but need to be chosen upfront.

4.1.2 Data Set Bifurcation and Trifurcation

However, we don't necessarily need to worry about making a wrong choice of hyperparameters that cannot subsequently be changed. In practice we would be prepared to *calibrate* the model by *tuning* its performance by turning the dials on the hyperparameter values.

Usually the dataset that we are working with will be either *bifurcated* into a *training* and a *test* set. Or will be *trifurcated* into a *training*, *validation* and a *test* set.

We'll return to this as we go through, but briefly…

- The *training set*: used to initialise the machine learning model.
- The *validation set*: used to tune the hyperparameters.
- The *test set*: used as **unseen data** to derive final performance quality measurements after training and validation has been completed.

It is important to realise that the test set output should never be used to further tune and calibrate the model. It is a ***hold out*** set that simulates how the model will perform in the ***real world*** on unseen data.

The data set is treated in all of these cases as *ground truth* - it is believed to be true, although in practice some data points might contain errors, or be missing. And there is almost certainly going to be some noise on any numerical values recorded in the data.

There no hard and fast rules on the proportions to use to bifurcate or trifurcate the data set. We might bifurcate using 75%/25% for example, or trifurcate with 50%/25%/25%.

Introducing scikit-learn, our first visit

Let's now see now how to use `scikit-learn` to do *k*-NN classification with the penguins data that we cleaned and prepared.

The following code was adapted in its early stages from *Machine Learning with Python, tutorialspoint* as found here https://www.tutorialspoint.com/machine_learning_with_python/index.htm or here https://www.tutorialspoint.com/machine_learning_with_python/machine_learning_with_python_tutorial.pdf

You'll have seen a number of instances by now in these notebooks where external sources are liberally referenced. Feel free to do this - but make sure that you **always acknowledge your sources**.

We are going to work with the entire cleaned-up penguins data set that we originally stored in `dfp2`.

Let's remember what it loked like…

```
[ ]: dfp2.head()
```

We want to use the numerical features (values) in each row to predict species.

Before we start using the `sklearn` python library we need to see how we can pick these data items out using **array slicing**.

First, we can pick out the value of the species with this command (the colon part is important - it refers to column zero)

`dfp2.iloc[2, 0:1].values`

This refers to the entry in the third, the '2', row and first, the '0:1', column.

To refer to all rows we replace the `2` with a colon `:` - as we'll see below.

Let's see it in action…

[ ]:
```
dfp2.iloc[2, 0:1].values
```

Second, we can refer to the four numerical features with this command

`dfp2.iloc[1, 2:6].values`

which refers to second row, and columns three to six inclusive. Once again we will use a colon to refer to all rows.

Again, let's see this in action…

[ ]:
```
dfp2.iloc[1, 2:6].values
```

4.2 Using sklearn

We will now fit the k-NN model using the Manhattan, or taxicab, norm, which we also call the $p = 1$ norm:

$$\|\boldsymbol{x}^* - \boldsymbol{x}_i\|_1.$$

In addition, we will use two ($k = 2$) nearest neighbours, and we will also obtain something called the confusion matrix, and will print some performance data.

The last two of these will be re-visited because they exhibit two very important means in which we can assess the performance of our model.

Typically we assign the data set features to a variable called `X`, and the data set labels to a variable called `y`. Using the array slicing that we saw above this is straightforward…

[ ]:
```
# We assign the numerical features to X
X = dfp2.iloc[:, 2:6].values
# And we assign the species label to y
y = dfp2.iloc[:, 0].values
```

We could bifurcate the data into a training and test set ourselves, but `sklearn` provides a helper function for this. It is called `train_test_split`.

First we have to import it. Then we give it `X` and `y` and specify the proportion of the data that we use for the *hold out*, or *test* set. we'll specify that 40% of the data should be reserved for testing.

[ ]:

```python
# from the scikit-learn library we use 40% of the data to test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.40)
```

The function returns four subsets of data:

```
X_train - 60% of the data set features to be used to configure the model
X_test  - 40% of the data set features to used to test the configured model
y_train - 60% of the data set features matching the X_train features
y_test  - 40% of the data set features matching the X_test features
```

We can look at the sizes of each of these by using `shape` as follows...

[ ]:

```python
print('shape of X_train = ', X_train.shape,' and of X_test = ', X_test.shape)
print('shape of y_train = ', y_train.shape,' and of y_test = ', y_test.shape)
```

Normalization of Data The next step is to normalize the feature data - the importance and role of this step is discussed in the recommended reading of pages 19 - 25 [MLFCES]. Again, `sklearn` provides a helper function for this called `StandardScaler`. This will remove the mean from the data and scale to unit variance. You can read more about this here: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[ ]:

```python
# import the helper and give it a name
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# initialise the scaler by feeding it the training data
scaler.fit(X_train)
# now carry out the transformation of all of the feauture data
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

REMARK: note that `X_train` is used to provide the scaling data, and not `X_test`. This is because `X_test` is *hold out data*. We must treat it as **unseen**. We can freely transform it though, because that can be done without actually looking at it.

Fitting: Learning from Data We can now bring in the k-NN classifier method from `sklearn` and obtain a `classifier` object that uses $k = 2$ nearest neighbours and the $p = 1$ norm.

[ ]:

```python
# import the k-NN classifier
from sklearn.neighbors import KNeighborsClassifier
# assign it with k=2 and p=1
classifier = KNeighborsClassifier(n_neighbors=2, p=1)
# give the training data to the classifier
classifier.fit(X_train, y_train)
```

The last step above is just like the coloured cluster plots above **before** we plotted the larger crosses. The model now has *knowledge* of these clusters, this is an example of *machine learning*.

By giving the model the unseen test data we are in effect telling it where the large crosses are. The model then finds the two nearest neighbours, using the manhattan norm, to classify the species of those crosses. This produces predictions of the species in `y_test`, and we call these predicted species values `y_pred`.

So, with the `crosses` as the features in the test set, we feed this in to the classifier and obtain the predicted values as follows…

[ ]:
```python
y_pred = classifier.predict(X_test)
```

Evaluation of Performance Now we come to the real crux of the matter. We know what `X_test` should produce as species values - they are in `y_test`. What we actually get though are `y_pred`. If `y_pred = y_test` then we should be very happy because it indicates that the model works very well on unseen data.

In practice though, it is unlikely that each of the 134 elements in `y_pred` will match every one of the corresponding values in `y_test`.

We have several tools available to assess the quality of the model. We'll take a quick look at a couple of these now, with a brief explanation, and we'll return many times to them later and understand them in more detail.

First we import the helper functions. Then we obtain and print the **confusion matrix**, next some statistics in a **classification report**, and then an **accuracy score**.

[ ]:
```python
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)

clsrep = classification_report(y_test, y_pred)
print("Classification Report:",)
print(clsrep)

accsc = accuracy_score(y_test,y_pred)
print("Accuracy:", accsc)
```

We'll come back to the classification report later, and for now just note that the accuracy score tells us the proportion of the test set for which the species was correctly predicted.

What we want to spend some time on here is the confusion matrix.

The Confusion Matrix

[ ]:
```python
print(cm)
```

The confusion matrix is square with the same number of rows/columns as there are values for the label. In our case there are three possible label values: *Adelie*, *Chinstrap*, and *Gentoo*. We can refer to these as group 1, 2 and 3.

The entry in row i and column j of the confusion matrix tells us how many data points in `X_test` that were in group i were predicted by the model to be in group j.

Now, the representation of the confusion matrix above is a numpy array and although it is useful for coding, it isn't very user friendly. The following code gives us something much nicer, and it is much easier to understand.

[ ]:
```python
from sklearn.metrics import ConfusionMatrixDisplay
cmplot = ConfusionMatrixDisplay(cm, display_labels=classifier.classes_)
cmplot.plot()
plt.show()
```

We can now immediately get a feeling for *how good* the model is. The diagonal elements tell us how many species predictions match the true value. The off-diagonals tell us how many misses there are, and how they missed.

For example, the number in the middle of the top row tells us how many Adelie penguins were mistakenly predicted to be Chinstraps.

Also, the overall accuracy percentage can be determined by adding all the numbers in the matrix, calling the total B, and adding all the diagonal elements together, as A. The value of A/B then tells us the proportion of correct predictions - and that is the *Accuracy* score above.

We haven't yet properly reviewed the mathematical concept and notion of a matrix yet, although we will do soon. We will be coming back to confusion matrices over and over again though.

Before moving on to some exercises we close with a comment about using the k-NN model for *regression*.

4.3 k-NN for regression

Above we saw how we can use k-N for classification: here, given feature data from an observation, we predict the label as the category the observation should be assigned to.

Regression is where the features and the labels vary in continuous sets of values. For example, we might want to predict the amount of rainfall given the number of hours of cloud, sun, daylight, along with air tempertaure, humidity and pressure.

These are all continuous variables, not discrete categorical ones.

The k-NN technique can also be used for regression by, in effect, turning the continuous variables into discrete ones. To get an idea of this imagine learning a function $y = f(x)$ as follows.

- take a set of values (features) $x_1, x_2, x_3, \ldots$
- take the corresponding labels $f(x_1), f(x_2), f(x_3), \ldots$
- plot these points - treat each as a cluster, as above.

The predict the function value given a new point, x^*, we would:

1. determine i such that $|x^* - x_i|$ is minimal over all of $x_1, x_2, x_3, \ldots$.
2. say that x_i is the nearest neighbour to x^*
3. estimate $f(x^*)$ by $f(x_i)$.

This will work well if f is quite well-behaved: continuous and not very rapidly varying, for example.

We wont touch more on this - it isn't on our journey. If you are interested in seeing more about this though you can, for example, look here https://stackabuse.com/k-nearest-neighbors-algorithm-in-python-and-scikit-learn/ for a demonstration of this using the California house data set.

Exercise Experiment with the k-NN classifier we just developed. For example,

- Change the 60%/40% bifurcation
- Change the value of k: decrease it to 1, or increase it to $3, 4, 5, \ldots$
- Change the norm from $p = 1$ to $p > 1$.
- Does $p < 1$ make any sense here?

Exercise Look at the following scatter plots. Suppose we wished to predict gender from two features.

- What two features would work best do you think?
- Which pairs of features are unlikely to work well?

```
[ ]: sns.scatterplot(data=dfp2, x="bill_length_mm", y="bill_depth_mm",␣
      ↪style="species", hue="sex")
```

```
[ ]: sns.scatterplot(data=dfp2, x="flipper_length_mm", y="body_mass_g",␣
      ↪style="species", hue="sex")
```

```
[ ]: sns.scatterplot(data=dfp2, x="body_mass_g", y="bill_depth_mm", style="species",␣
      ↪hue="sex")
```

```
[ ]: sns.scatterplot(data=dfp2, x="body_mass_g", y="bill_length_mm",␣
      ↪style="species", hue="sex")
```

```
[ ]: sns.scatterplot(data=dfp2, x="bill_length_mm", y="flipper_length_mm",␣
      ↪style="species", hue="sex")
```

```
[ ]: sns.scatterplot(data=dfp2, x="bill_depth_mm", y="flipper_length_mm",␣
      ↪style="species", hue="sex")
```

Exercise The confusion matrix we generated above is a **numpy array**. We will be looking in much more detail at these objects - both mathematically and in code - soon, but first here is a warm up. Let's recall the matrix:

```
[ ]: cm
```

We can use `cm[0,0]` to access the value in the first row and first column.

- What do you think `cm[1,1]` and `cm[2,2]` refer to?
- what do you think `cm[0,0]+cm[1,1]+cm[2,2]` produces?

Check your answers by using

- `print(cm[1,1],cm[2,2])`

- `print(cm[0,0]+cm[1,1]+cm[2,2])`

[ ]:

```python
print(cm[1,1],cm[2,2])
print(cm[0,0]+cm[1,1]+cm[2,2])
```

What do you think `cm.sum()` produces? Check, or discover, with

- `print(cm.sum())`

[ ]:

```python
print(cm.sum())
```

How do you think `cm[0,0]+cm[1,1]+cm[2,2]` and `cm.sum()` relate to the `Accuracy` score given above? Print out your answer and check.

[ ]:

```python
print((cm[0,0]+cm[1,1]+cm[2,2])/cm.sum())
```

Compare `np.trace(cm)` to `cm[0,0]+cm[1,1]+cm[2,2]` - use your findings to shorten the command above

[ ]:

```python
print(np.trace(cm)/cm.sum())
```

4.4 Technical Notes, Production and Archiving

Ignore the material below. What follows is not relevant to the material being taught.

Production Workflow

- Finalise the notebook material above
- Clear and fresh run of entire notebook
- Create html slide show:
  - `jupyter nbconvert --to slides 3_knn.ipynb`
- Set `OUTPUTTING=1` below
- Comment out the display of web-sourced diagrams
- Clear and fresh run of entire notebook
- Comment back in the display of web-sourced diagrams
- Clear all cell output
- Set `OUTPUTTING=0` below
- Save
- git add, commit and push to FML
- copy PDF, HTML etc to web site
  - git add, commit and push
- rebuild binder

Some of this originated from

https://stackoverflow.com/questions/38540326/save-html-of-a-jupyter-notebook-from-within-the-r

These lines create a back up of the notebook. They can be ignored.

At some point this is better as a bash script outside of the notebook

```
[ ]: %%bash
     NBROOTNAME='3_knn'
     OUTPUTTING=1

     if [ $OUTPUTTING -eq 1 ]; then
       jupyter nbconvert --to html $NBROOTNAME.ipynb
       cp $NBROOTNAME.html ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.html
       mv -f $NBROOTNAME.html ./formats/html/

       jupyter nbconvert --to pdf $NBROOTNAME.ipynb
       cp $NBROOTNAME.pdf ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.pdf
       mv -f $NBROOTNAME.pdf ./formats/pdf/

       jupyter nbconvert --to script $NBROOTNAME.ipynb
       cp $NBROOTNAME.py ../backups/$(date +"%m_%d_%Y-%H%M%S")_$NBROOTNAME.py
       mv -f $NBROOTNAME.py ./formats/py/
     else
       echo 'Not Generating html, pdf and py output versions'
     fi
```