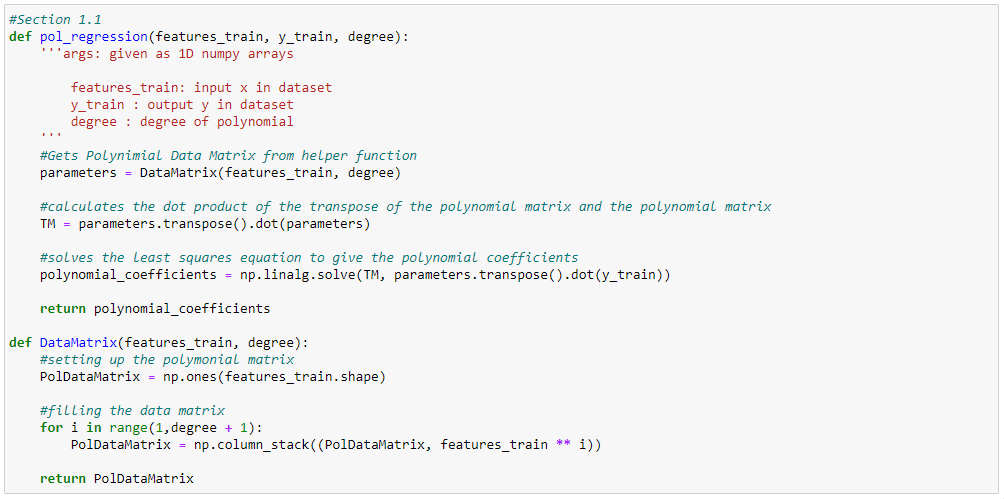
# Task 1

## Section 1.1

The polynomial regression algorithm was implemented using two functions of the following names: “pol\_regression” and “DataMatrix”, Figure 1 illustrates this. The former is the main function required by the brief performing the regression while the latter is a helper function. It builds the polynomial matrix and was done so to allow a polynomial matrix to be made separately from regressing it. This will be useful in the next section.

***Figure 1:*** *Shown here is the two functions which perform the polynomial feature expansions and the least squares method to find the polynomial weights*

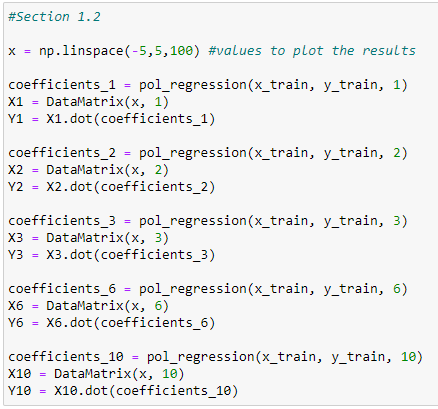
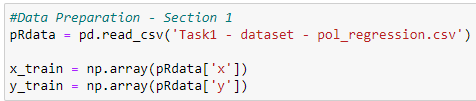
As can be seen in figure 1, The “Data\_Matrix” function does the feature expansion of the input values up to the given degree. Firstly, it creates a new matrix with the same shape as the matrix containing the input values. This new matrix is filled with the features train raised to the exponent “i” for the number of degrees that have been passed. For a 2-degree feature expansion, this would mean taking the exponent of each input values with 1 and 2 and stacking the results in the new array giving a 2-d array with the expanded features and an extra column containing ones. The reason for doing it like so is because the degree of the polynomial is used to control how many features are added during the feature expansion. A degree of 3 will give 3 features while a degree of 4 will give 4 and so on.

The “pol\_regression” is used to calculate the polynomial parameters or weights using the least squares method. The equations for this method can be seen below:

The equation is interpreted as the dot product of the pseudo inverse of the matrix X () and y giving us the weights of the polynomial (w). This equation has been implemented using numpy matrix operations.

The first step of the function is to call its helper function to get the polynomial matrix for the correct degree of polymonial. This polynomial matrix is referred to as X here. Next, we calculate using the “np.transpose()” and “np.dot()” commands from “numpy”. After we have calculated this we can solve the rest of the equation using “np.linalg.solve” and giving it and ) as arguments. Perhaps a more intuitive way to go about this would be to use “np.linalg.inv” to find the inverse of and using “.dot()” to find the dot product of the result and ) rather than using “np.linalg.solve” as it goes more along the lines of the equation, although ultimately the result and the calculations done would be the exact same either way.

## Section 1.2



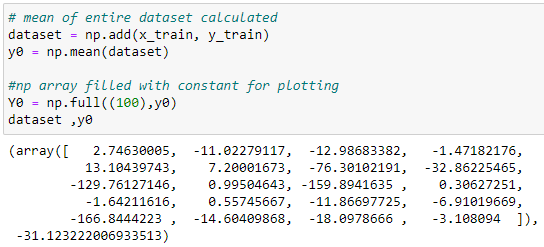
***Figure 2 (Opposite):*** *Shown here is the process used to get the polynomial regressions to a plottable stage for each degree. (1,2,3,6,10)*

***Figure 3 (Below)****: Data preparation before Figure 2 can be compiled*

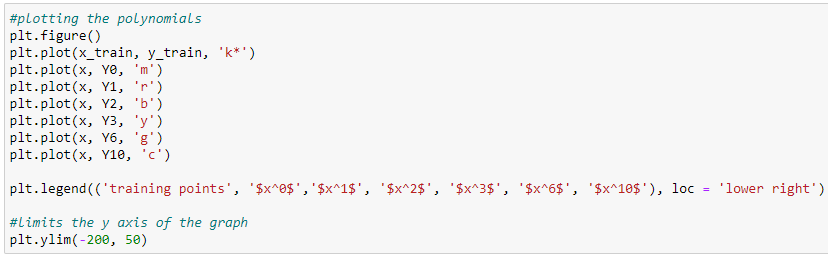
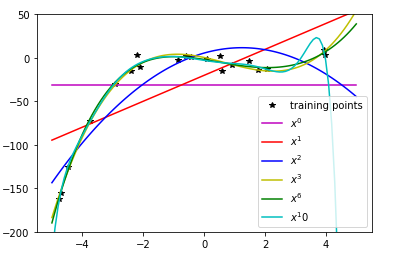
As shown in figure 2, to be able to plot the fitted polynomial functions, we would need to compute this using a sorted test set array. However, we used the whole dataset as a training set. We must therefore make our own test set here to be able to achieve this. This is the purpose of the first line in figure 2: it computes 100 random sorted numbers between the range of -5 and 5. This is the range that we require to compute the fitted polynomials and to plot them as required by the brief. Additionally, we must unpack the data from the excel spreadsheet and split it into two arrays: one for the x dataset and the other for the y dataset. This is done in Figure 3

Now we can calculate the fitted polynomial functions. To do this, the functions written in the previous section must be used. Firstly, we use “pol\_regression” to find the polynomial weights by supplying it with the dataset previously unpacked along with the degree of the polynomial. We must also perform the polynomial feature expansion to the relevant degree on the test set we have made so we can compute the fitted polynomials. This is the reason for writing the feature expansion function “DataMatrix” as a separate function from “pol\_regression” as discussed in section 1.1. The last step consists of calculating the dot product of the feature expanded test set and the coefficients calculated to give us the fitted polynomial function ready to be plotted. Figure 2 shows this being done for the polynomials of degree 1, 2, 3, 6 and 10. 0 is discussed separately on the next page as it was done differently in a simpler manner.

To compute a polynomial of 0 degree, it is not necessary to go through the same process as done with the other degrees. This is the reason why: “A polynomial of degree 0 is just a constant because ” (Que, 2021). It is also mentioned that “polynomial regression with a degree of 0” would return a “single constant value” which would be the same as “the mean of the dataset” (Que, 2021). This means that rather than having to go through the same process as the other degrees, we can just calculate the mean of the entire dataset and plot the constant given. Figure 4 shows this. The entire dataset is added into a single array. The mean of that array is then calculated giving us the constant that we require to plot the polynomial of degree 0 of our dataset. A 1D numpy array of size 100 is then filled with this constant for the purposes of plotting as otherwise an error would be thrown.



***Figure 4:*** *The process for finding the 0-degree polynomial is shown here along with the numerical results of the array and the constant calculated*

Lastly, we must plot all of the fitted polynomial functions as well as the training points so that we can examine the degree of polynomial that best fits the data. The code to achieve this can be seen in Figure 5. It is also worth noting that we have limited the y axis to between -200 and 50 due to the odd shape of the 10th degree polynomial. This so that we can see the area of the graph that we require.

***Figure 6:*** *The plot resulting from the code in* ***figure 5*** *being compiled. This shows the training points as well as each of the degree of polynomials computed. The training set is plotted on the x axis with the fitted polynomial functions being plotted on the y axis.*

***Figure 5:*** *The code used to plot the polynomials as well as labelling the graph and limiting the y axis*

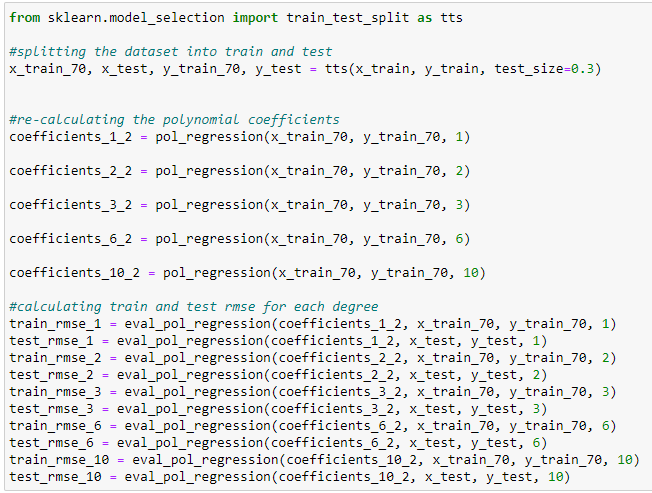
Upon seeing the results in Figure 6, there are a few things that can be discussed. Firstly, we can see from the graph that the straight lines in magenta and red (0 degree and 1 degree respectively) do not fit the data well. While the blue line (2 degrees) is curved, its curve doesn’t represent the data well either. These three lines have a model complexity that is too low and cannot accurately represent the data. Likewise, the cyan line (10 degrees) represents the data too well and takes into account the noise leading it to have an overly curvy shape. This leaves the yellow and green lines (3 degrees and 6 degrees respectively). These both fit the data well without taking too much of the noise into account. Because they are so similar in the way that they fit the data, in this case I would choose either to represent the data.

## Section 1.3

***Figure 7:*** *This shows the implementation of the eval\_pol\_regression function along with the calculated rmse values for the parameters from section 1.2*

As shown in Figure 7, the eval\_pol\_regression function was implemented firstly by performing the polynomial feature expansion on the parameter x in the manner implemented in section 1.1 as well as finding the fitted polynomial as done in section 1.2. This will serve as the predicted y dataset which is used along with the actual y output dataset to compute the rmse. The equation for rmse can be seen below:

This equation is interpreted as the square root of the mean of the predicted values minus the actual values squared. As our values are in matrices, we can just use numpy matrix operations to calculate this. In Figure 7, this is done by subtracting y from Y, squaring it, calculating the mean and computing the square root of the mean. This gives us the rmse. The results are shown below the code snippet. As expected, the values for 0 and 1 are high while the rest are lower as they fit better.



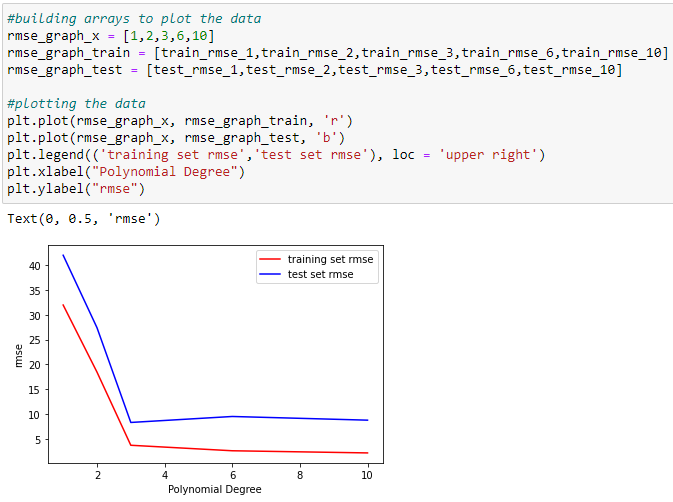
***Figure 8:*** *This shows the code for splitting the dataset into a 70-30 train-test set, calculating new coefficients and train rmse and test rmse for each degree of polynomial*

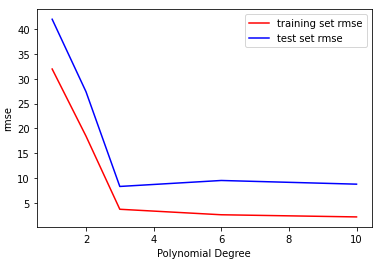
The dataset was split into a training set and a test set in a 70% - 30% split using the “train\_test\_split” function from sklearn. Next, we could re-calculate the polynomial coefficients using the “pol\_regression” function already implemented in Section 1.1. This was done for degrees 1, 2, 3, 6 and 10. 0 degree was not included here as it was done through different means in the previous sections and therefore did not use this method of calculation. The rmse on the training set and the test set was calculated next using the “eval\_pol\_regression” function discussed previously. This was again done for every degree but 0. All the steps mentioned can be seen in Figure 8 above.

Once all the required values had been computed, they were put into three list so that they could be plotted into a graph to illustrate the results. This process is shown in Figure 9 and the resulting graph in Figure 10.

***Fig 9(Opposite) & Fig 10 (Below):***

*This shows the code used to make arrays for the data computed and plotting it. The resulting graph is also shown.*



The graph in Figure 10 shows both rmse values, train rmse and test rmse plotted against their respective degree of polynomial. As expected, the graph has a downward trend as the bigger the degree, the best it will fit the data. This is true up to a point however which is generally when the polynomial degree is more than 4. At that point the model complexity becomes too high and instead of fitting just the data, the noise is also taken into account meaning the data isn’t really represented well enough and the fitted polynomial can start to take on some interesting shapes. This is known as over-fitting. This actually can be seen on Figure 10 as the line becomes notably flatter after 3 degrees and actually loses a bit of accuracy when it reaches 6 degrees. Before the line reaches 3 however, the rmse and therefore accuracy of the prediction is much lower. This is due to the model complexity being too low which is usually for degrees less than 3. This means that the data cannot be represented well enough as the degree of polynomial is simply too low to do so. This is known as under-fitting. We can conclude that the best degree to represent our data is therefore a polynomial of degree 3. This does not suffer from over or under fitting the data and will give the most accurate result on the dataset. Something also worth noting is that the training set rmse is generally lower. This is for the simple reason that this is the data that the model as learnt from meaning that it can represent it well as it has learnt it by the book. The test set will therefore always have a lower rmse because it is actually predicting the data rather than just applying something it has simply learnt by the book to itself.

# Task 2

## Section 2.1

The k-means algorithm implemented is shown on the next page in Figure 11. It consists of the main function “k-means” as well as two helper function, these are “compute\_eucledian\_distance” and “initialise\_centroids”. Firstly, the helper functions will be discussed following by the main function.

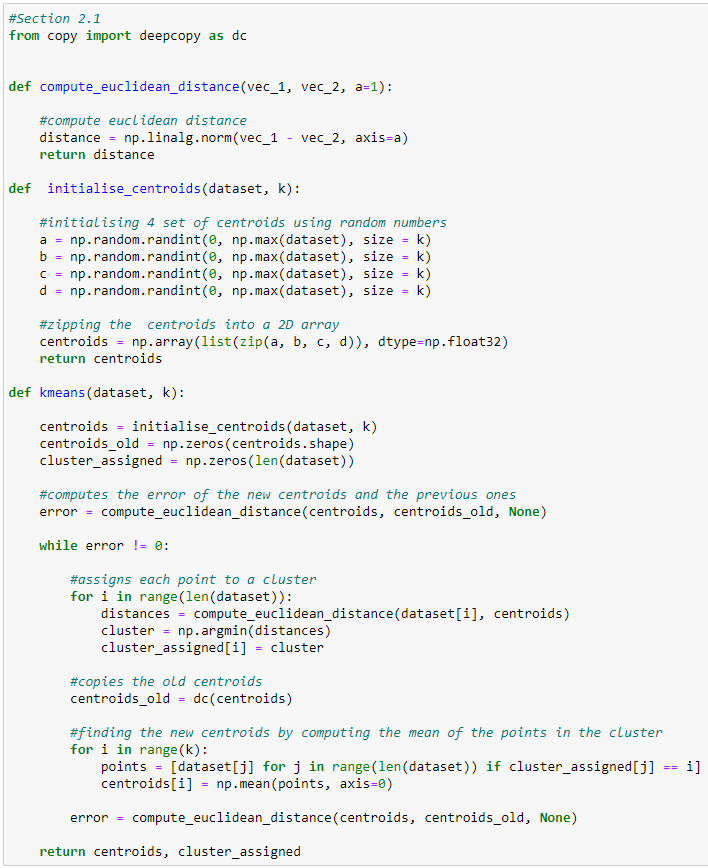
The “compute\_euclidean\_distance” computes the Euclidean distance between the points in two vectors as its names suggests. The equation for calculating the Euclidean distance can be seen below:

As shown by the equation, the Euclidean distance can be calculated from the cartesian coordinates of the points using the theorem of Pythagoras giving the distance between the points. In the implementation in Figure 11, it does so using the linalg.norm function from numpy. This computes the same thing as the equation essentially. One argument was added however to specify the axis along which to return the matrix norm. In some instances, this is used but is not in other instances, it is reverted to None when it is not needed such as when calculating the distance between the new and old centroids in the “k-means” function.

“initialise\_centroids”, as its name suggests, is used to initialise the centroids. This is done by picking random numbers within the range of the dataset. This was deemed to be the best approach as picking a random point within the data. This ensures that the centroids can be spread on a bigger range of places within the data ensuring that there is less chance that the centroids are next to each other. At least that was the idea, in practice however, it frequently means that one of the centroids is not assigned any data meaning the algorithm must be run 3 or 4 times to achieve satisfactory results. This is especially true when k is 3. The other approach mentioned would have been better as then the centroids are always close to data points and would always have data assigned to them making the algorithm much more effective.

It is also worth nothing that we need as many centroids as we have features. In this case, we have 4 features, so we need 4 set of k centroids as shown in the code in Figure 11.

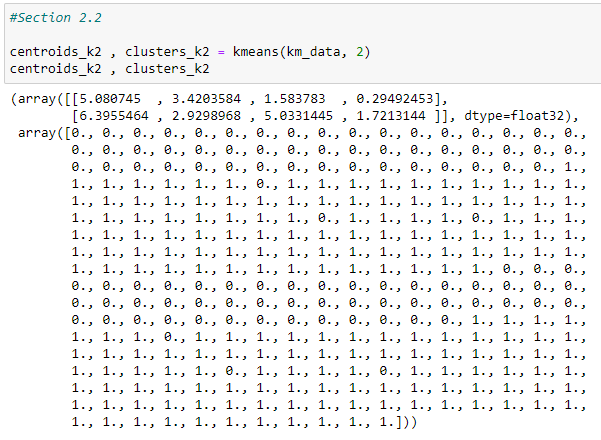
***Figure 11:*** *This shows the whole k-means algorithm implemented including the functions to initialize the centroids and compute the Euclidean distance*

The main function in the algorithm, “k-means” works in the following way. Firstly, the centroids are initialised using the function “initialise\_centroids” previously discussed. An empty copy of the centroids array, “centroids\_old“, is also made to store the old centroids for comparison purposes. The cluster\_assigned array is also initialised to store the cluster to which the points are assigned to. The error variable is also initialised here to prevent an error being thrown when the compiler hits the while loop. Inside this while loop is where the magic happens. Each point is assigned to a cluster by computing the Euclidean distance to each centroid and picking the smallest value. The centroids are then deep copied to “centroids\_old” before the new centroids are calculated using the “deepcopy” function from the copy library. The new centroids are then calculated by gathering all of the points in a cluster and calculating the mean of those points. The error is then calculated which is the Euclidean distance between the new and the old centroids. The algorithm is repeated until the error becomes 0. This is when the centroids have reached a state where they are not moving anymore, and the best clusters have been formed.

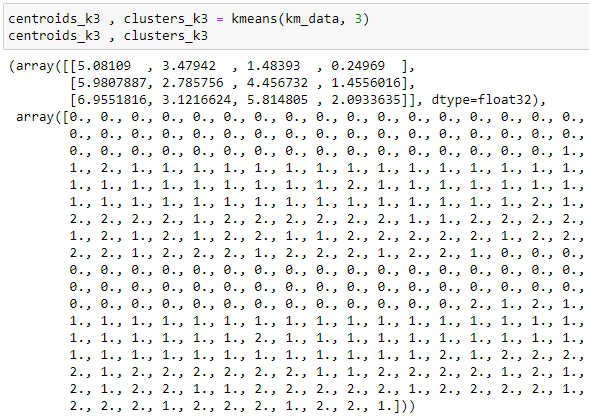
## Section 2.2

With the k-means algorithm described within the context of the implementation in the previous section. We will move on to the results of the implementation in this section.

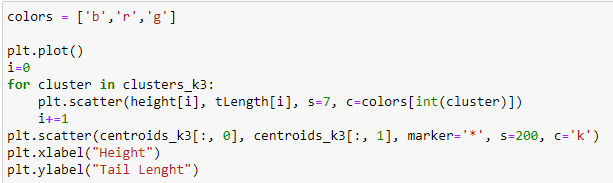
The algorithm in Figure 11 (previous page) was run twice with the whole dataset and the k values of 2 and 3 respectively. The output shown in Figure 12 shows the centroids as well as the cluster assigned for both iterations of the algorithm.



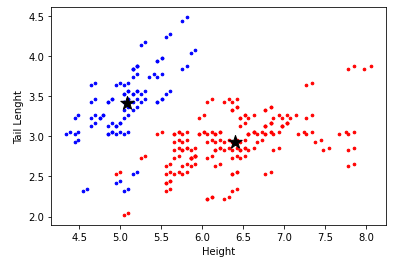
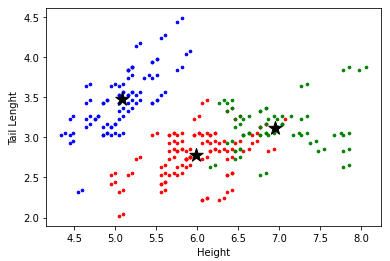
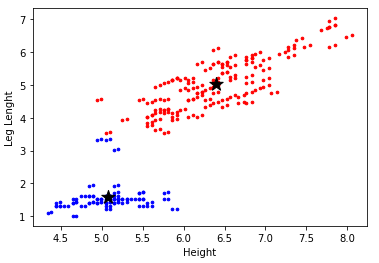
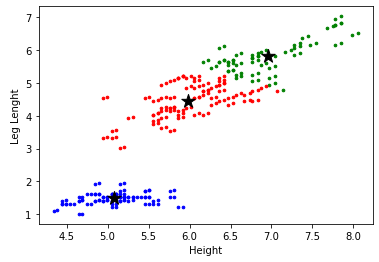
***Figure 12:*** *This shows the k-means implementation being run* ***twice*** *with the respective k values of 2 and 3. Underneath each code snippet is the data returned from the function which includes the centroids and the cluster to which the data was assigned*



After the algorithms was compiled. The required plots were plotted using the code in Figure 13, this simply iterated through the data and the cluster assigned to plot the data with the correct colour to mark the cluster that it is assigned to. The resulting plots are shown in Figure 14.



***Figure 13:*** *Shown here is an example of the code used to plot the graphs displaying the results. All plots were done in this manner with the labels changing depending on the plot*

As can be seen in Figure 14, the data can be clustered quite well. All graphs have height as their x-axis whilst the upper graphs have tail length as their y-axis and the lower having leg length as their y-axis. The graphs have a tendency to work better with a k value of 2 because of the way that the data separates into two different groupings generally meaning that it is easier to cluster that way. Having said that, it still works well with a k value of 3. Albeit it is perhaps less obvious where the third cluster would be especially on the upper right graph.

***Figure 14:*** *Shown here are the plots displaying the clustered data from the k-means algorithm that was executed twice as discussed in figure 13. The left plots are for k = 2 whilst the left plots are for k = 3*

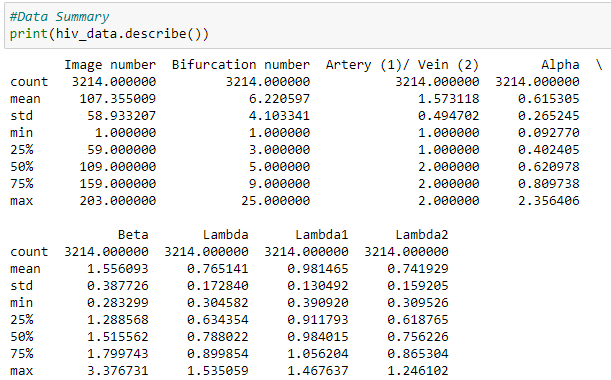
Going back to the question asked in the brief: “Can these features be used to cluster the dog breeds into 3 groups?” Some features do this better than other as shown in Figure 14. The Height and Leg Length pair of graphs (lower pair in the figure) do this very well especially with k = 3 as there is a distinct correlation between the two features and the three cluster reflect that. You can see three distinct groups of data which undoubtedly could be grouped into three different dog breeds. However, it is another story with the Height and Tail Length pair (upper pair) as the data is all over and is much less grouped giving almost no correlation at all between the two features. To answer, I would say that it depends on the features that you choose whether you could group the data into different groups for different dog breeds, but it can be done if you choose the right features.

# Task 3

## Section 3.1

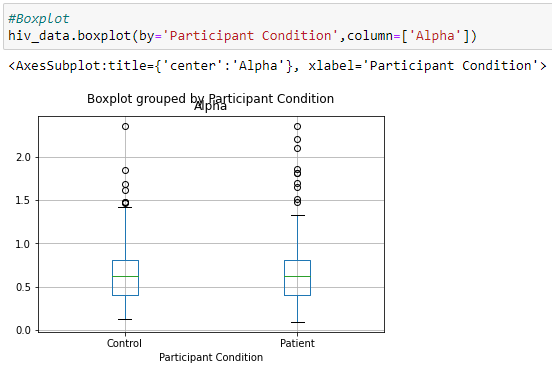
This section will discuss the data supplied for section 3 and will establish and discuss whether that data requires normalisation before doing anything else.

Firstly, the dataset has been loaded in jupyter notebook as a pandas dataframe and a statistical summary of the dataset has been provided using the df.describe() functions from pandas dataframe library. This is shown in Figure 15.

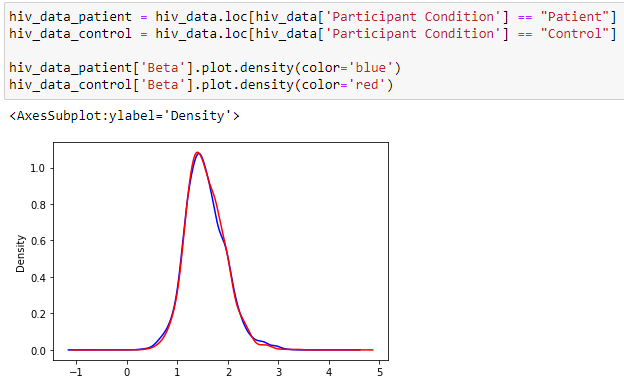


***Figure 15:*** *Shown here is the summary of the HIV dataset. It includes the mean, standard deviation, min value and max values as well as additional summary values*

The fields we are interested in is the mean and the standard deviation of the dataset. The standard deviation tells us how close the data is to the mean. These alone will give us a good indication of the spread of the data and whether there are many outliers within the dataset.

Looking at these statistics, we can see that the standard deviation is low across the board with most of the data having a standard deviation of less that 0.5. This is the case in all that data that we are likely to use to classify later such as the alpha, beta and lambda. This means that the data does not differ a lot from the mean telling us there are not a lot of outliers within it. Also relevant is to compare the min and max values with the mean. This gives an indication of where the bulk of the data would sit within the entire set of data for each feature. Again, I am pleased to report that the means tend to sit in roughly the middle of the min and max values which is good news especially given the low standard deviation values. Next, we will have a look at a few plots to summarise the data further. These are in Figure 16 and 17.

***Figure 16:*** *Shown here is a boxplot of the dataset. This shows the spread of the data of the alpha column grouped by the participant conditions*



***Figure 17:*** *Shown here is a density plot of the beta column, again grouped by patient condition with the red line being control and the blue line patients*

Upon seeing the two plots in Figure 16 and 17, the data between the two participant conditions is extremely similar with both box plots being almost the same as well as the lines in the density plot being the same. This could suggest that the data would be hard to classify as it is so similar. While this may be true, there is another observation that can be made. It is that the data isn’t very spread and does not have many outliers especially in the beta feature as shown in Figure 17. Figure 16, however, does show some outliers towards the top of the graph. These are very few and should not affect the results much.

To conclude, the data does not need normalisation as there are few outliers as demonstrated by the summary statistics and plots shown. We can therefore just use the data as it is as it is unlikely to skew or change the results. We can now move onto classifying the data.

# References

Que, A. (2021). *Polynomial Regression.* Retrieved from polynomialregression.drque.net: http://polynomialregression.drque.net/math.html