Joseph Doody 16603020@students.lincoln.ac.uk 16603020

Machine Learning CMP3751M Assignment 1

Task 1

Section 1.1 Description of polynomial Regression

Regression models are the “work horse” in statistics, supervised learning and data mining. We use regression for estimating the relationships between a dependent variable and one or more independent variables. In a polynomial regression we have our dependent variable and independent variable we must get a *d*th order polynomial (where d is denoted as degree). Error within regression is the distance a value is from the regression line and we can calculate it with function $e_n = y_n - \hat{y}_n$. We can use this function to calculate all the errors within the data set and with the errors we can calculate the Sum of Squares Error (SSE) which measures how far the data is from the model’s predicted values. An expression of SSE:

$$\sum_{n}^{N} e_n^2$$

So, let’s say we got a simple linear regression model, we can express it as $y := f(x) = w_0 + w_1 x + e$ where w_0 is the intercept and w_1 is the slop. From the expression of SSE earlier, where $e_n = y_n - \hat{y}_n = y_n - w_0 - w_1 x_n$. Then:

$$E(w_0, w_1) = \sum_{n=1}^{N} (y_n - w_0 - w_1 x_n)^2$$

We can consider SSE is a function of w_0 and w_1 and we can see SSE is a quadratic function of w_0 w_1.

If we go back to linear models, we can talk about least squares solution with linear regression given in matrix form. Given a vector of inputs $X^T = (X_1, X_2, \ldots, X_n)$ we can predict the output Y via the model:

$$\hat{y} = w_0 + \sum_{n=1}^{N} X_n w_n$$

Now we can fit the linear model to a set of training data which involves the least squares approach where we pick the coefficients w to minimise the residual sum of squares.

$$RSS(w) = \sum_{i=1}^{N} (y_i - x_i^T w)^2$$

As you can see RSS is a quadratic function, we can write it in another way in matrix notation.

$$RSS(w) = (y - Xw)^T (y - Xw)$$

Where X is an N x p matrix with each row an input vector, and y is an N-vector of the outputs in the training set. So now we can derive the function for 0 to work out for w:

- Cancel constant factors: $(y - Xw)^T X = 0$
- Transpose on both sides (using $(AB)^T = B^T A^T$: $X^T (y - Xw) = 0$

- Multiply out brackets: $X^T y - X^T X w = 0$
- Bring $X^T y$ on other side: $X^T X w = X^T y$
- Multiply both sides by the inverse of $X^T X$ (multiply on the left): $w = (X^T X)^{-1} X^T y$

Thus, giving us the least squares solution by:

$$w = (X^T X)^{-1} X^T y$$

Within polynomial regression our regression line will look different to out least squares regression line. Polynomial regression works on a degrees order so lets say with our random set of data we go to the second degree polynomial regression we will have a quadratic regression curve to match the dataset, if we want to go to the third degree the regression will look different again with a cubic regression curve matching to the data set. We can keep increasing the degree of the curve in polynomial regression, this is what's known as polynomial feature expansion because you are increasing the size of the feature to a certain degree.

We can use the equation within polynomial regression where d is degree.

$$\hat{y} = w_0 + \sum_{i=1}^{d} w_i x^i$$

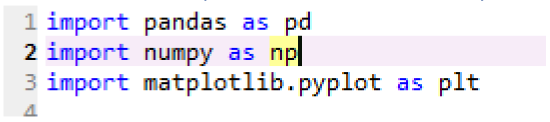
Noe that, while $\hat{y}$ is now non-linear in x, it is still linear in the parameters w_i. Hence, we can still apply linear regression. We can still describe $\hat{y}$ as a scalar product and set it as a data matrix which we can use to perform polynomial regression.

$$\hat{y} = w_0 + \sum_{i=1}^{n} w_i x^i = x^T w,$$

With $x = [1, x^1, x^2, \ldots, x^d]$ and $w = [w_0, \ldots, w_{d+1}]$.

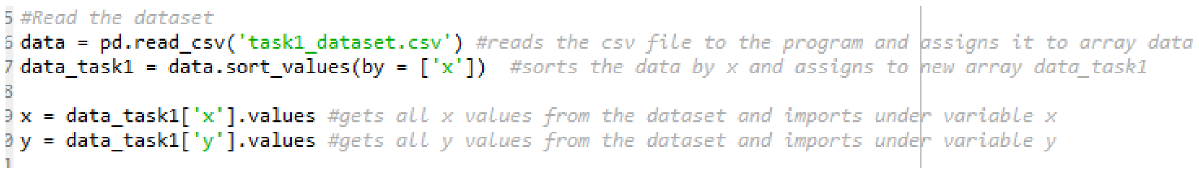
For the datasets to test on polynomial regression I will have a training set of data and a test set of data. Ideally I will randomise the data and have 70% of it train set and 30% test set. From here I will use the equations from above to create a matrix of the dataset which I can then use to create polynomial regression at different degrees for the whole data set.

Section 1.2 Implementation of Polynomial Regression



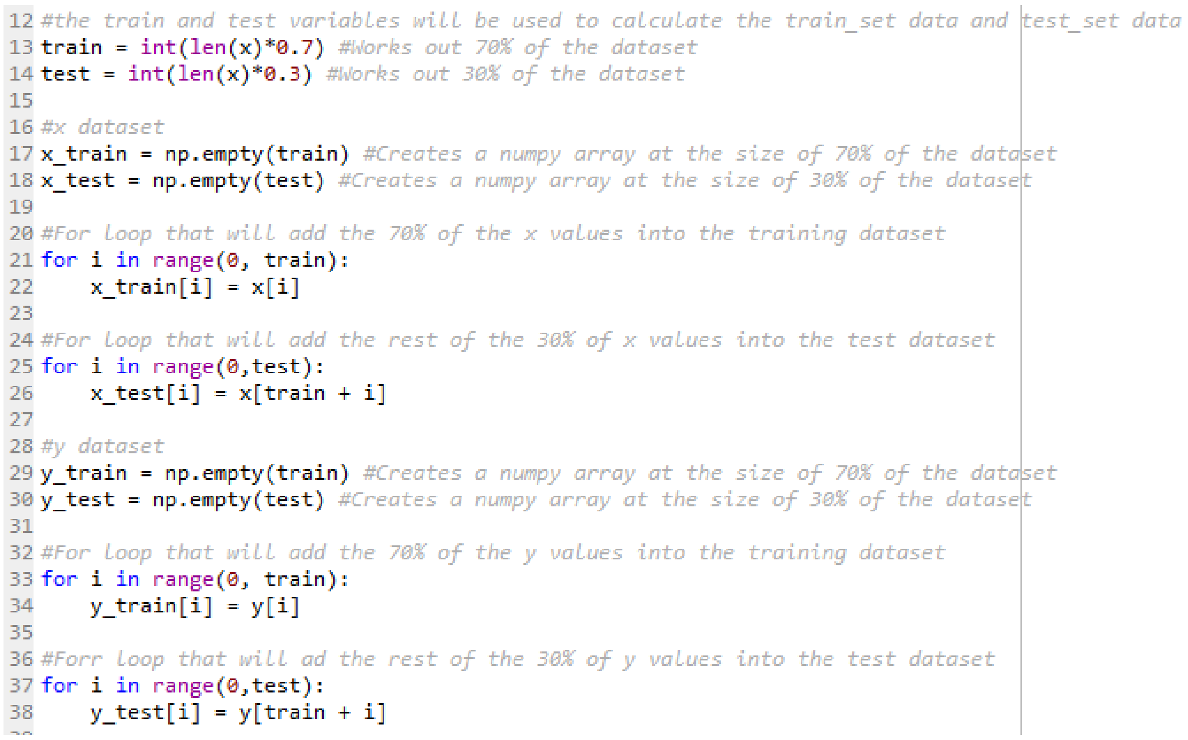
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

These are the libraries that I will be using within the implementation of my polynomial regression algorithm. The Pandas library allows me to handle data and I will mainly be using it to import a file from outside, the file will store the necessary data to be analysed for the polynomial regression. The NumPy library allows me to work with NumPy specific arrays which is helpful for certain functions I will be using in the means of creating a data matrix for the regression. The Matplot library allows me to display my data in the forms of a chart so I will be able to show the plots from the dataset and show the regression.



```python
#Read the dataset
data = pd.read_csv('task1_dataset.csv') #reads the csv file to the program and assigns it to array data
data_task1 = data.sort_values(by = ['x'])  #sorts the data by x and assigns to new array data_task1

x = data_task1['x'].values #gets all x values from the dataset and imports under variable x
y = data_task1['y'].values #gets all y values from the dataset and imports under variable y
```

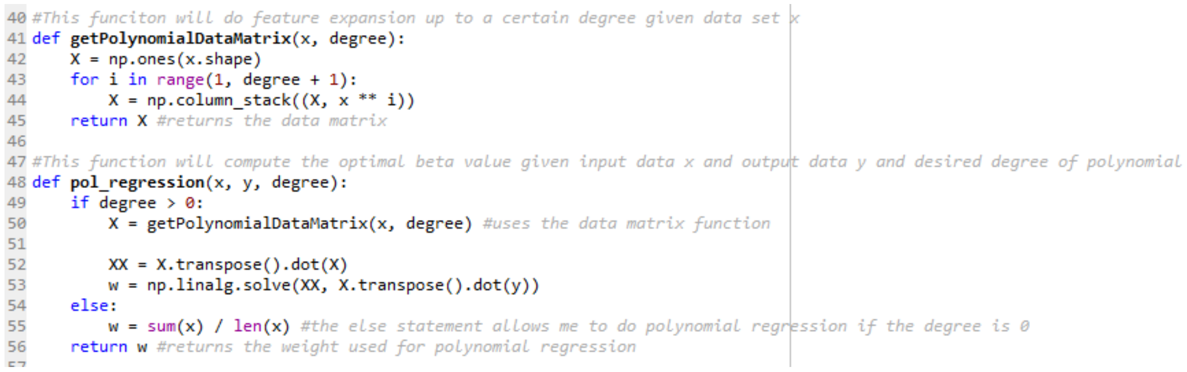
Here I have read the file where the dataset is stored and gathered the data from both x and y columns within the dataset, I have then sorted the dataset by column x so it would be easier for me to perform the polynomial regression and assigned them to variables within the program to x and y respectably. Both x and y arrays holding the datasets will be later used to create training data and test data.



```python
#the train and test variables will be used to calculate the train_set data and test_set data
train = int(len(x)*0.7) #Works out 70% of the dataset
test = int(len(x)*0.3) #Works out 30% of the dataset

#x dataset
x_train = np.empty(train) #Creates a numpy array at the size of 70% of the dataset
x_test = np.empty(test) #Creates a numpy array at the size of 30% of the dataset

#For loop that will add the 70% of the x values into the training dataset
for i in range(0, train):
    x_train[i] = x[i]

#For loop that will add the rest of the 30% of x values into the test dataset
for i in range(0,test):
    x_test[i] = x[train + i]

#y dataset
y_train = np.empty(train) #Creates a numpy array at the size of 70% of the dataset
y_test = np.empty(test) #Creates a numpy array at the size of 30% of the dataset

#For loop that will add the 70% of the y values into the training dataset
for i in range(0, train):
    y_train[i] = y[i]

#Forr loop that will ad the rest of the 30% of y values into the test dataset
for i in range(0,test):
    y_test[i] = y[train + i]
```

Here I have the snippet of code which allows me to split the dataset into training data and test data. What I have done is split the code 70% to 30% for both x and y. I then assign the 70% dataset (training dataset) to both x_train and y_train NumPy arrays and assign the rest of the 30% of the dataset (test dataset) to both x_test and y_test NumPy arrays. Now I have the data to start performing my polynomial regression. The way I have figured out to split my data is by finding the length of the dataset using the function len(). From here I would multiply the length by 0.7 which would give me 70% amount of values from the dataset, this number I will then put to the variable train so I would know what amount of data is needed for the training set. I would do a similar operation for the test set where I would use the len() function again but instead multiply it by 0.3 which would give me 30% of the data and assign that number to test.

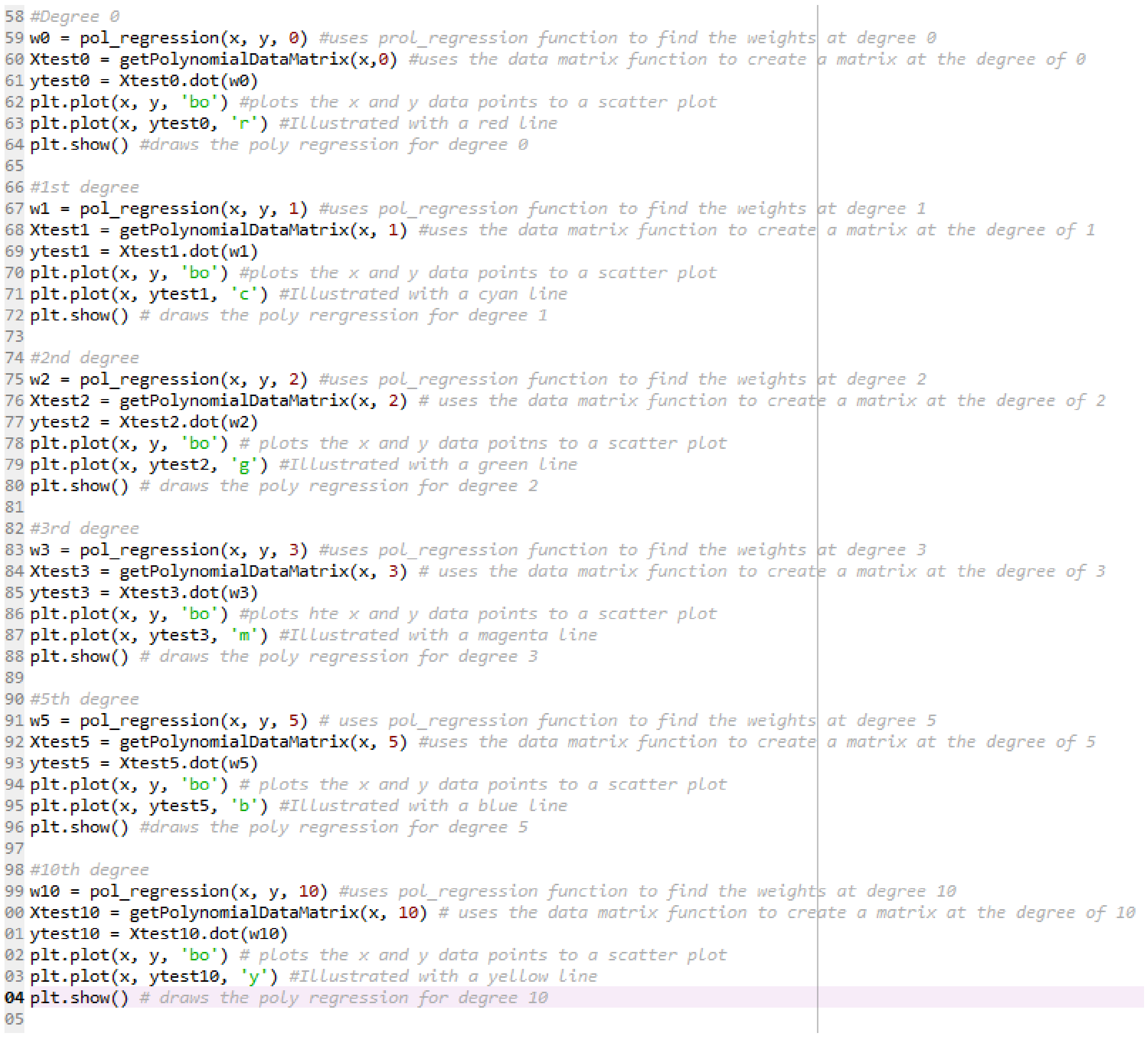
From here I can get the data within the dataset and get the first 70% of the dataset and store it as training set, to work out the test set I would start 1 element after the training set within the dataset so that would give me the rest of the data set as test data. I would do that for both x and y values.



```
#This funciton will do feature expansion up to a certain degree given data set x
def getPolynomialDataMatrix(x, degree):
    X = np.ones(x.shape)
    for i in range(1, degree + 1):
        X = np.column_stack((X, x ** i))
    return X #returns the data matrix

#This function will compute the optimal beta value given input data x and output data y and desired degree of polynomial
def pol_regression(x, y, degree):
    if degree > 0:
        X = getPolynomialDataMatrix(x, degree) #uses the data matrix function

        XX = X.transpose().dot(X)
        w = np.linalg.solve(XX, X.transpose().dot(y))
    else:
        w = sum(x) / len(x) #the else statement allows me to do polynomial regression if the degree is 0
    return w #returns the weight used for polynomial regression
```

I have two different functions used for my polynomial regression model. The first function I have is called getPolynomialDataMatrix, this allows me to do the feature expansion I will need to do to perform polynomial expansion. What I will do is have an x value, which will be my feature and create the matrix based on the degree it is given.

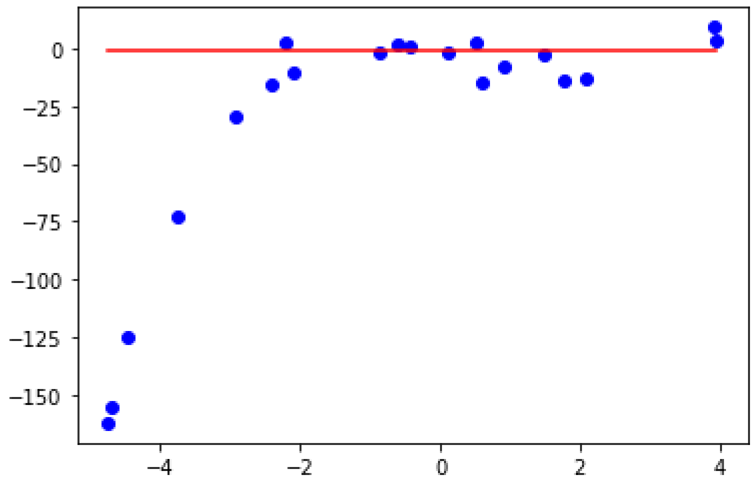
The second function I have is called pol_regression, with this function it allows me to compute the optimal beta values given the input data x and output data y and desired degree of polynomial. The function will return the weights depending on the degree it is given and will also be given both datasets for values x and y. However I did have to create an if statement if I wanted to find the polynomial regression with degree of 0 because if I was to give it a degree of zero it would give a 0-dimensional array and the Array must be of at least two-dimensions.



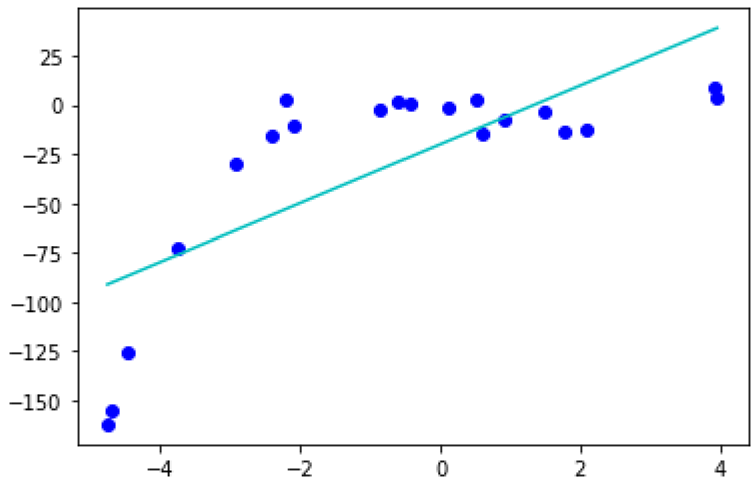
```
#Degree 0
w0 = pol_regression(x, y, 0) #uses prol_regression function to find the weights at degree 0
Xtest0 = getPolynomialDataMatrix(x,0) #uses the data matrix function to create a matrix at the degree of 0
ytest0 = Xtest0.dot(w0)
plt.plot(x, y, 'bo') #plots the x and y data points to a scatter plot
plt.plot(x, ytest0, 'r') #Illustrated with a red line
plt.show() #draws the poly regression for degree 0

#1st degree
w1 = pol_regression(x, y, 1) #uses pol_regression function to find the weights at degree 1
Xtest1 = getPolynomialDataMatrix(x, 1) #uses the data matrix function to create a matrix at the degree of 1
ytest1 = Xtest1.dot(w1)
plt.plot(x, y, 'bo') #plots the x and y data points to a scatter plot
plt.plot(x, ytest1, 'c') #Illustrated with a cyan line
plt.show() # draws the poly rergression for degree 1

#2nd degree
w2 = pol_regression(x, y, 2) #uses pol_regression function to find the weights at degree 2
Xtest2 = getPolynomialDataMatrix(x, 2) # uses the data matrix function to create a matrix at the degree of 2
ytest2 = Xtest2.dot(w2)
plt.plot(x, y, 'bo') # plots the x and y data poitns to a scatter plot
plt.plot(x, ytest2, 'g') #Illustrated with a green line
plt.show() # draws the poly regression for degree 2

#3rd degree
w3 = pol_regression(x, y, 3) #uses pol_regression function to find the weights at degree 3
Xtest3 = getPolynomialDataMatrix(x, 3) # uses the data matrix function to create a matrix at the degree of 3
ytest3 = Xtest3.dot(w3)
plt.plot(x, y, 'bo') #plots hte x and y data points to a scatter plot
plt.plot(x, ytest3, 'm') #Illustrated with a magenta line
plt.show() # draws the poly regression for degree 3

#5th degree
w5 = pol_regression(x, y, 5) # uses pol_regression function to find the weights at degree 5
Xtest5 = getPolynomialDataMatrix(x, 5) #uses the data matrix function to create a matrix at the degree of 5
ytest5 = Xtest5.dot(w5)
plt.plot(x, y, 'bo') # plots the x and y data points to a scatter plot
plt.plot(x, ytest5, 'b') #Illustrated with a blue line
plt.show() #draws the poly regression for degree 5

#10th degree
w10 = pol_regression(x, y, 10) #uses pol_regression function to find the weights at degree 10
Xtest10 = getPolynomialDataMatrix(x, 10) # uses the data matrix function to create a matrix at the degree of 10
ytest10 = Xtest10.dot(w10)
plt.plot(x, y, 'bo') # plots the x and y data points to a scatter plot
plt.plot(x, ytest10, 'y') #Illustrated with a yellow line
plt.show() # draws the poly regression for degree 10
```

This is where I put both the functions together and display the polynomial regression in visual form under the aid of a scatter plot. What I do here is find weight for the desired degree using the pol_regression function, create a test result from the data matrix function on the same degree, dot product the test result of x to test result y and then plot the x and y data points onto the graph and then the test results y based on x. I would do this for degrees: 0, 1, 2, 3, 5 and 10 and would give me the following results.

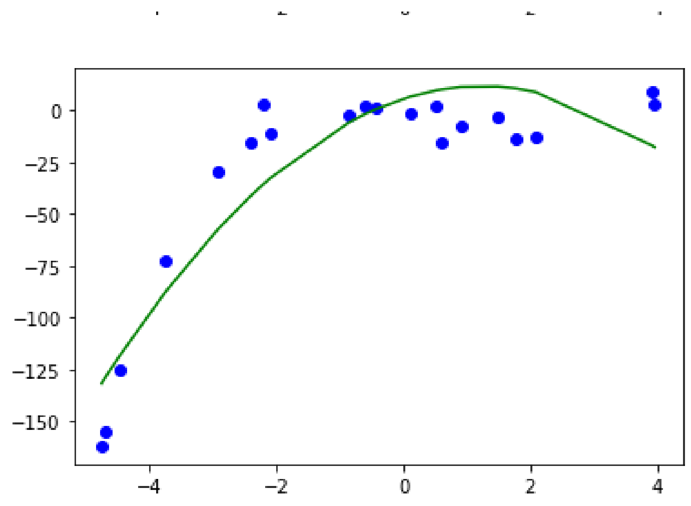
Degree of 0



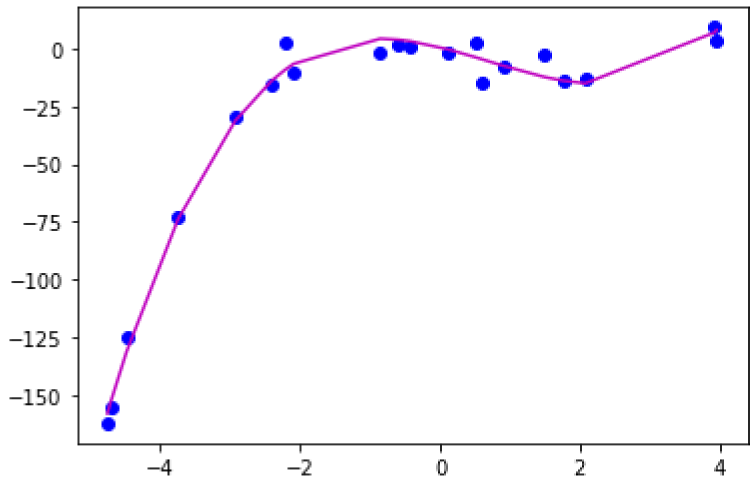
Degree of 1



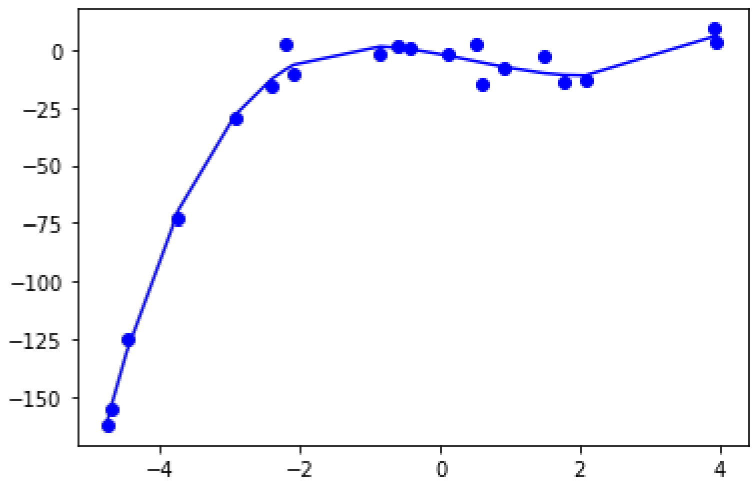
Degree of 2



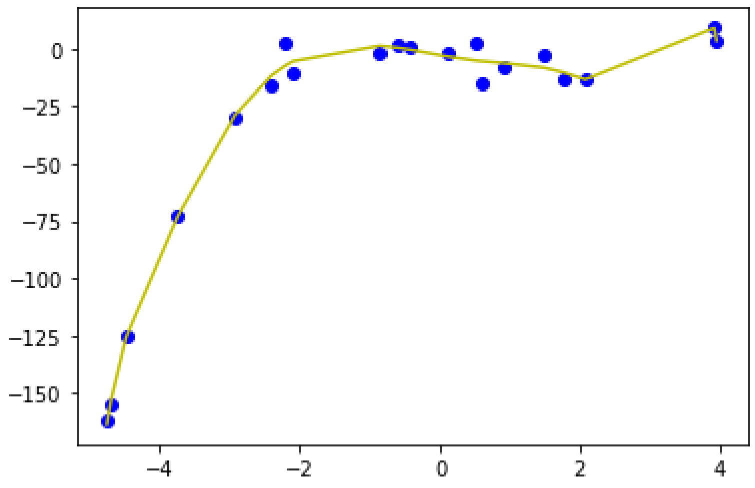
Degree of 3



Degree of 5



<u>Degree of 10</u>

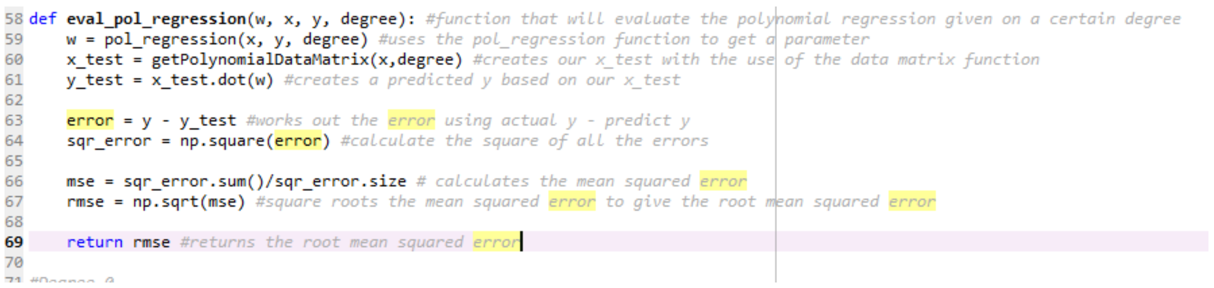


From what I can deduct from my polynomial regression the higher the degree the better the regression line matches to the ground truth data points. From the degree values I used there was not much difference between degree of 5 and degree of 10 so there would be no need to go up to that level of degree since of the little changes it shows.

It is nice to see how the when the degree does increase you can see the gradual change where the regression line turns to a curve to better match the data points. Looking at degree 0 through to 3 this shows the difference in how the degree of the polynomial can really affect the reliability of the polynomial regression.

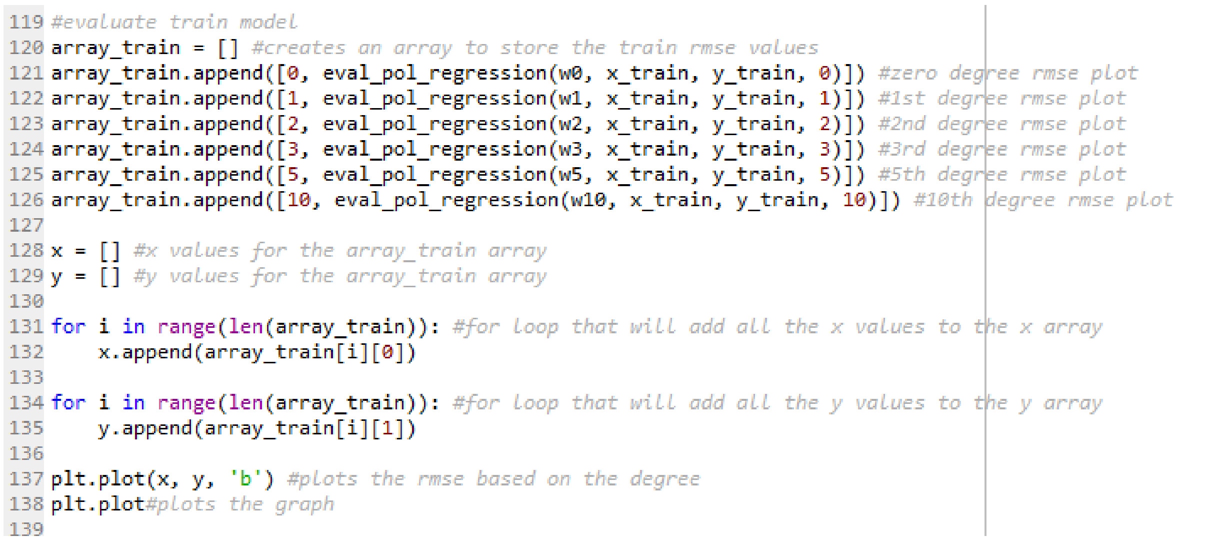
Section 1.3 Evaluation

I will now evaluate both RMSE on training set and test set.



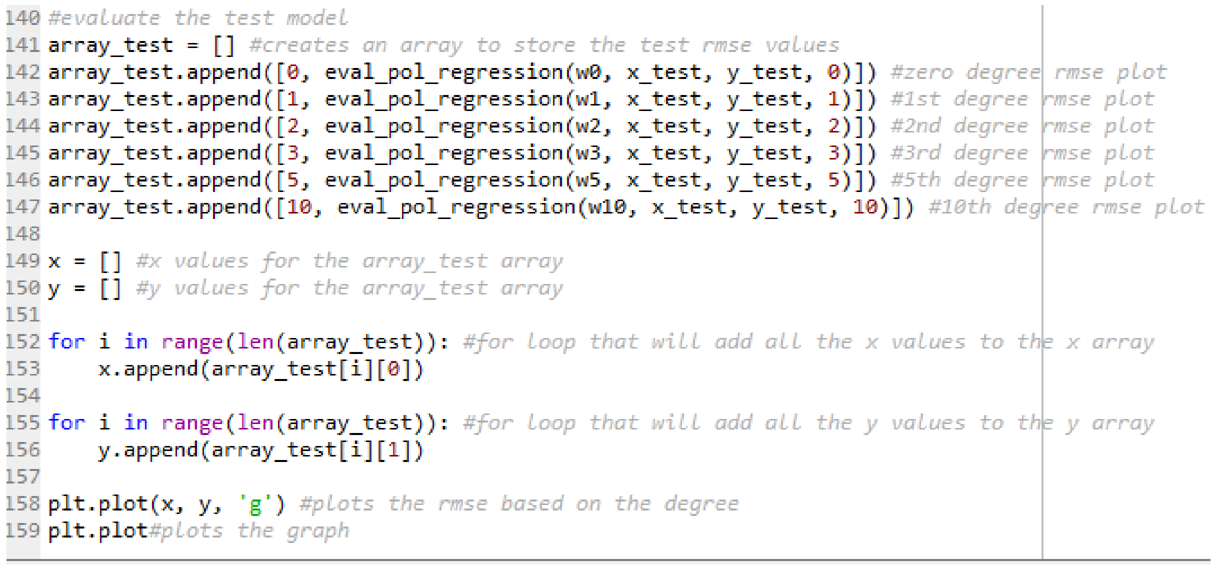
```python
def eval_pol_regression(w, x, y, degree): #function that will evaluate the polynomial regression given on a certain degree
    w = pol_regression(x, y, degree) #uses the pol_regression function to get a parameter
    x_test = getPolynomialDataMatrix(x,degree) #creates our x_test with the use of the data matrix function
    y_test = x_test.dot(w) #creates a predicted y based on our x_test

    error = y - y_test #works out the error using actual y - predict y
    sqr_error = np.square(error) #calculate the square of all the errors

    mse = sqr_error.sum()/sqr_error.size # calculates the mean squared error
    rmse = np.sqrt(mse) #square roots the mean squared error to give the root mean squared error

    return rmse #returns the root mean squared error

```

This is the function used to evaluate the polynomial regression based on the degree. The parameters the function takes are the parameters from the pol_regression function alongside an x and y which will be the train and test datasets, alongside the degree that we are finding the error on. To work out my weight I must re use the pol_regression function where I can then use the data matrix formula to help me work out predictions for $\hat{y}$. I dot product the x data matrix with the weights found to give me $\hat{y}$. From here I can use the formula $y_i - \hat{y}$ to calculate the error for the points of y. First I will need to figure out the square of all the errors, I do this by storing it in a NumPy array which allows me to square every value within it. I then want to find the Mean squared error where I sum all values in the squared errors array and divide by the size of the array. By square rooting the mean squared error I can then work out the root mean squared error.



```python
#evaluate train model
array_train = [] #creates an array to store the train rmse values
array_train.append([0, eval_pol_regression(w0, x_train, y_train, 0)]) #zero degree rmse plot
array_train.append([1, eval_pol_regression(w1, x_train, y_train, 1)]) #1st degree rmse plot
array_train.append([2, eval_pol_regression(w2, x_train, y_train, 2)]) #2nd degree rmse plot
array_train.append([3, eval_pol_regression(w3, x_train, y_train, 3)]) #3rd degree rmse plot
array_train.append([5, eval_pol_regression(w5, x_train, y_train, 5)]) #5th degree rmse plot
array_train.append([10, eval_pol_regression(w10, x_train, y_train, 10)]) #10th degree rmse plot

x = [] #x values for the array_train array
y = [] #y values for the array_train array

for i in range(len(array_train)): #for loop that will add all the x values to the x array
    x.append(array_train[i][0])

for i in range(len(array_train)): #for loop that will add all the y values to the y array
    y.append(array_train[i][1])

plt.plot(x, y, 'b') #plots the rmse based on the degree
plt.plot#plots the graph
```

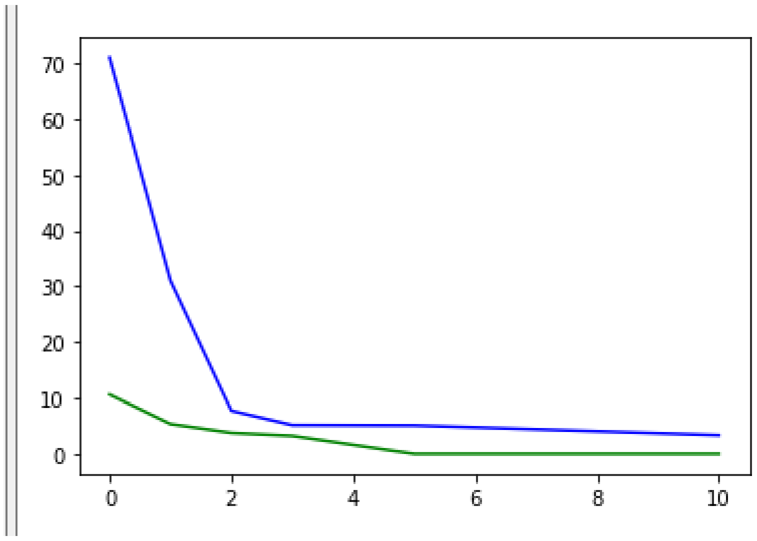
I now want to plot each of the Root Mean Squared Errors to the degree it was calculated on to a graph. I want to do this with train and test data which have been split up into 70% and 30% respectively. I will first start with the training data, so I create a train array to store the Root Mean squared Errors. For each degree I am recording data on I perform the function to evaluate the polynomial regression. Once all have been added to the new array, I then split the array to two separate arrays x and y, I will run a for loop for both x and y to run through the train array to get the x values out and append it to the respected new array. From here I can then create a plot that will display the curve of how the Root Mean Squared Error changes as the degree changes.



```python
#evaluate the test model
array_test = [] #creates an array to store the test rmse values
array_test.append([0, eval_pol_regression(w0, x_test, y_test, 0)]) #zero degree rmse plot
array_test.append([1, eval_pol_regression(w1, x_test, y_test, 1)]) #1st degree rmse plot
array_test.append([2, eval_pol_regression(w2, x_test, y_test, 2)]) #2nd degree rmse plot
array_test.append([3, eval_pol_regression(w3, x_test, y_test, 3)]) #3rd degree rmse plot
array_test.append([5, eval_pol_regression(w5, x_test, y_test, 5)]) #5th degree rmse plot
array_test.append([10, eval_pol_regression(w10, x_test, y_test, 10)]) #10th degree rmse plot

x = [] #x values for the array_test array
y = [] #y values for the array_test array

for i in range(len(array_test)): #for loop that will add all the x values to the x array
    x.append(array_test[i][0])

for i in range(len(array_test)): #for loop that will add all the y values to the y array
    y.append(array_test[i][1])

plt.plot(x, y, 'g') #plots the rmse based on the degree
plt.plot#plots the graph
```

Similar action was taken to find the test data as it was to find the Root Mean Squared Error for the training data.

Below are the results where the blue line is the training data and the green line is the testing data:



On this graph we have the degree alongside the x-axis and Root Mean Squared Error along the y-axis. The lower the Root Mean Squared Error is the better the regression is so as you can see on this graph the Root Mean Squared Error decreases as the degree gets bigger however it tends to level out once it gets to degree 5 and above. Before I would have said using degree five is good because looking at the graph there wasn't much change but when you look at the evaluation of the two datasets you can see there is a small change in how it is lower as the degree is higher but to me it isn't too much of a big deal and you can probably still get away with just using degree level of 5.

Task 2

Section 2.1 Description of the K-means Clustering

K-means clustering makes use of the K-means algorithm, an iterative algorithm that tries to partition a dataset into K pre-defined distinct non-overlapping subgroups known as clusters where each data point belongs to only one group. The way the algorithm works is that you determine the amount of clusters you want which you would assume as K. You would then initialise k amount of random points from the dataset, these are called the centroid points. These centroid points will be the key factors to starting the K-means clustering because you will need to find the lowest distance from each data point, and the data point that has the lowest distance to a centroid point because a member of that cluster within the centroid point, this is part of the cluster assignment step. The way distance is found out is by using the Euclidian distance formula:

$$D(X_i, c_j)^2 = \sum_{k=1}^{p}(X_{ik} - c_{jk})^2$$

Where $D(X_i, c_j)^2$ is the instance between X_i and cluster centrec_j.

The Euclidian formula works in a similar way of Pythagoras theorem, $c^2 = a^2 + b^2$, in theory what we are trying to find is the shortest distance to the point which if you would imagine on a 2-dimensional plane, would be the hypotenuse of a right angled triangle. In this case of the K-means cluster we are working in 2-dimensions so another way of writing the Euclidian formula to work out the distance between 2 plots in the dataset is:

$$D(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

Where we have the data points of (p,q).

Once each data point has been clustered to their centroid point you would then find the new centroid point, this part of the K-means algorithm is the update step. The way you would do this is by calculating the sum of the distance between each data point and its centroid and calculate the average of the distance, this would give you the new coordinates of the next centroid and you would go through the K-mean algorithm once again but this time using the new centroids and clustering the data points again, this time some data points may be sorted into new clusters. You will re iterate the process until the all the centroid points do not change. From here you can plot the centroid points done and data points that cluster to that centroid point and that would give you the data through K-means cluster.

There is also a function which can be used to minimise the aggregate cluster distance known as the objective function which is [1]:

$$J = \sum_{i=1}^{m} \sum_{k=1}^{K} w_{ik} \left\| x^i - \mu_k \right\|^2$$

Where you have $w_{ik} = 1$for data point x^i if it belongs to cluster k; otherwise, $w_{ik} = 0$. Also, $\mu_k$is the centroid of xi's cluster.

Following are the strengths and weaknesses to k-means clustering technique [2]:

Strengths of K-Mean

- Simple: - Easy to understand and to implement
- Efficient: Time complexity of O(tkn) which is where n is the number of data points, k is the number of clusters and t is the number of iterations
- Since both k and t are small. K-means is considered a linear algorithm.

Weaknesses of K-means

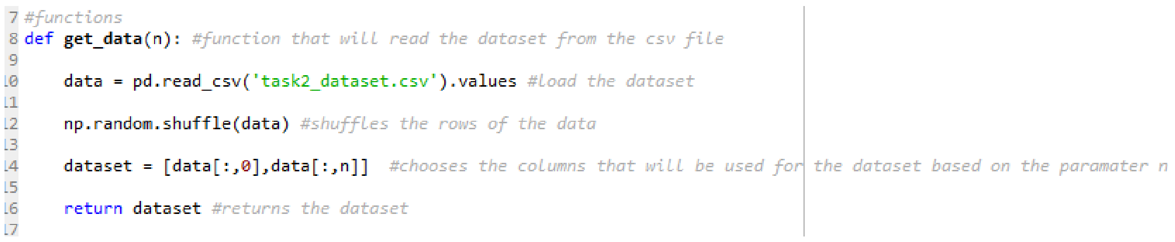
- The algorithm is only applicable if the mean is defined – For categorical data, k-mode – the centroid is represented by most frequent values.
- The user needs to specify k
- The algorithm is sensitive to outliers – Outliers are data points that are very far away from other data points.
  Outliers could be errors in the data recording or some special data points with very different values
- Weaknesses of k-means: To deal with outliers
- One method is to remove some data points in the clustering process that are much further away from the centroids than other data points

Section 2.2

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
import math

```

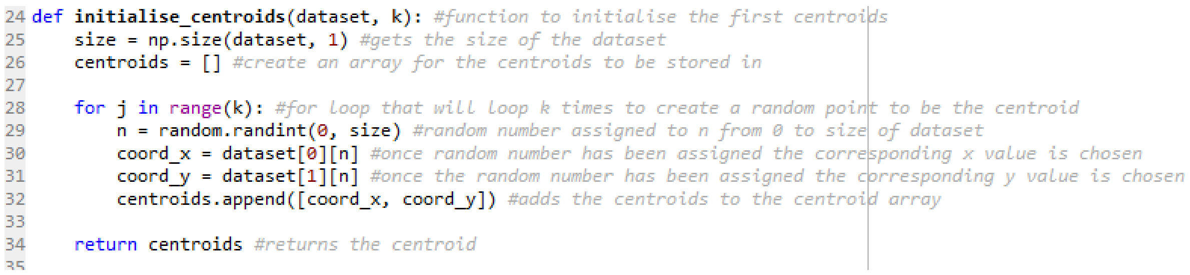
These are the libraries that I have used in the creation of my k-means algorithm. I use the NumPy library to use the various functions within NumPy such as the NumPy arrays which are needed for storing the various datasets and manipulating the datasets. The Pandas library will allow me to read the file that the dataset is stored in, into the program. I use the matplotlib library to allow me to create graphs which will be needed to create the scatter charts to show the clustering. The random library will allow me to generate a random number which is needed in the process of initialising the first centroids for K-means. And the last library is the math libraries which will allow me to use math functions such as sqrt() to find the square root of a number, this will be needed for the Euclidean formula.

I have four different functions that are used within my program. Once function is used to read the data into the program and assign two variables to a dataset. Another function will allow me to initialise the first centroid points based on K to start off the K-means algorithm. The third function will allow me to compute the Euclidian distance between two vectors which will be used to work out the shortest distance between a centroid and data point to determine the cluster. And the last function will be used to perform the K-means algorithm.



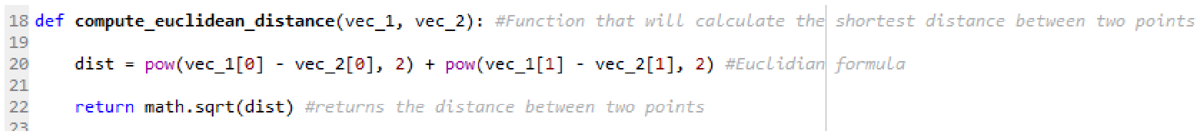
```
#functions
def get_data(n): #function that will read the dataset from the csv file

    data = pd.read_csv('task2_dataset.csv').values #load the dataset

    np.random.shuffle(data) #shuffles the rows of the data

    dataset = [data[:,0],data[:,n]]  #chooses the columns that will be used for the dataset based on the paramater n

    return dataset #returns the dataset

```

Our first function which will read the dataset stored in file ‘task2_dataset.csv’ and store all the values in the program under the variable of data. What I would do then is shuffle the data up and then create a new array named dataset based on what data would be needed such that it would hold the height data and n data whether it be the tail length, leg length or nose circumference. I will then return the dataset that is called within the get_data function depending on its y data that’s chosen.



```
def initialise_centroids(dataset, k): #function to initialise the first centroids
    size = np.size(dataset, 1) #gets the size of the dataset
    centroids = [] #create an array for the centroids to be stored in

    for j in range(k): #for loop that will loop k times to create a random point to be the centroid
        n = random.randint(0, size) #random number assigned to n from 0 to size of dataset
        coord_x = dataset[0][n] #once random number has been assigned the corresponding x value is chosen
        coord_y = dataset[1][n] #once the random number has been assigned the corresponding y value is chosen
        centroids.append([coord_x, coord_y]) #adds the centroids to the centroid array

    return centroids #returns the centroid

```

This next function will be used to initialise the first centroid points in the called dataset and will iterate it K amount of times. What the function will first do is calculate the size of the dataset. This allows me to figure out how much data is in the dataset and how far I can go into the dataset to call a random data point out from it to call it as a centroid point. The function will have a new array created called centroids which will store the centroids and then a for loop will be run which will run K times where it will create a random number based on the size of the dataset and this number will be the row of data used to create a centroid point. Once enough centroid points have been initialised it will return the centroid array.

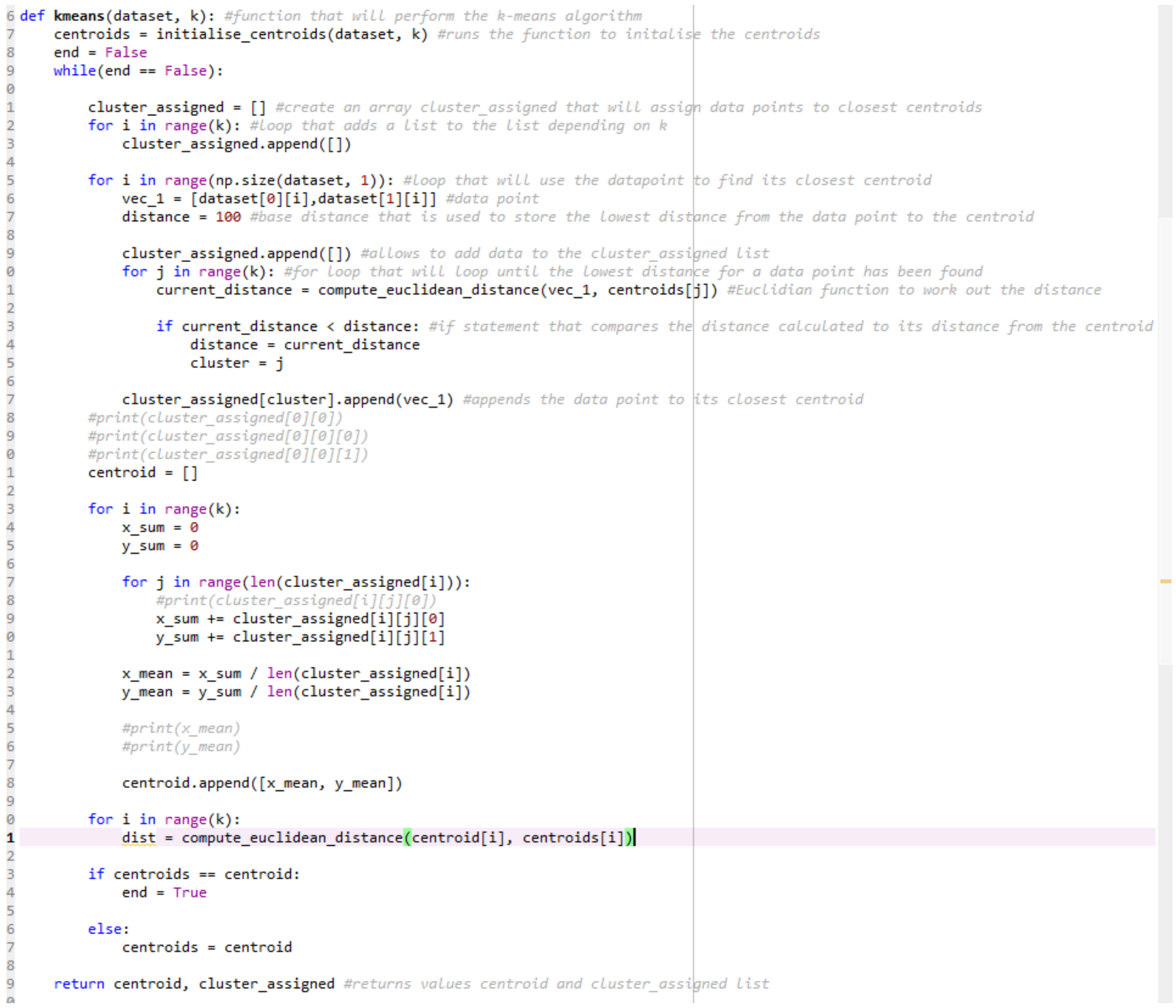


```
def compute_euclidean_distance(vec_1, vec_2): #Function that will calculate the shortest distance between two points

    dist = pow(vec_1[0] - vec_2[0], 2) + pow(vec_1[1] - vec_2[1], 2) #Euclidian formula

    return math.sqrt(dist) #returns the distance between two points

```

For the Euclidean_distance function we will take two parameters, both that will be data points. Since we are working on a two dimensional plane we will use the following formula to work out the distance between a point.

$$D(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

This function will be used in the next function K-Means to work out the shortest distance between a centroid and data point. The function will return distance between the two data points.



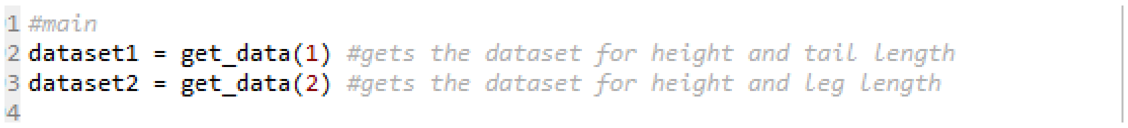
```python
def kmeans(dataset, k): #function that will perform the k-means algorithm
    centroids = initialise_centroids(dataset, k) #runs the function to initalise the centroids
    end = False
    while(end == False):

        cluster_assigned = [] #create an array cluster_assigned that will assign data points to closest centroids
        for i in range(k): #loop that adds a list to the list depending on k
            cluster_assigned.append([])

        for i in range(np.size(dataset, 1)): #loop that will use the datapoint to find its closest centroid
            vec_1 = [dataset[0][i],dataset[1][i]] #data point
            distance = 100 #base distance that is used to store the lowest distance from the data point to the centroid

            cluster_assigned.append([]) #allows to add data to the cluster_assigned list
            for j in range(k): #for loop that will loop until the lowest distance for a data point has been found
                current_distance = compute_euclidean_distance(vec_1, centroids[j]) #Euclidian function to work out the distance

                if current_distance < distance: #if statement that compares the distance calculated to its distance from the centroid
                    distance = current_distance
                    cluster = j

            cluster_assigned[cluster].append(vec_1) #appends the data point to its closest centroid
        #print(cluster_assigned[0][0])
        #print(cluster_assigned[0][0][0])
        #print(cluster_assigned[0][0][1])
        centroid = []

        for i in range(k):
            x_sum = 0
            y_sum = 0

            for j in range(len(cluster_assigned[i])):
                #print(cluster_assigned[i][j][0])
                x_sum += cluster_assigned[i][j][0]
                y_sum += cluster_assigned[i][j][1]

            x_mean = x_sum / len(cluster_assigned[i])
            y_mean = y_sum / len(cluster_assigned[i])

            #print(x_mean)
            #print(y_mean)

            centroid.append([x_mean, y_mean])

        for i in range(k):
            dist = compute_euclidean_distance(centroid[i], centroids[i])

        if centroids == centroid:
            end = True

        else:
            centroids = centroid

    return centroid, cluster_assigned #returns values centroid and cluster_assigned list
```

For our final function we have the K-Means function that will take on parameters dataset, which is the dataset it is reading the data from and K, which is the number of iterations used for the cluster. We start by using one of the previous functions initialise_centroids() to randomly get the first k amount of centroids. I use a while loop because I want to run the K-means algorithm multiple times to find the final cluster. I make use of a while loop Boolean evaluation where if one condition is met then I will set the end condition to True and the while loop should stop. With regards to the algorithm I create a new array called cluster_assigned, this array will store the data points to its closest centroid. The way I want to do this is in a two dimensional array because I want one column to store centroid and the other column to match the data point to that centroid cluster.

To calculate what data point is closest to what centroid, I make use of the Euclidean_distance function to find distance between two points. I use a base distance of 100 and I use the distance to determine the lowest distance, this is where I will compare the distance calculated from the function to the distance manually set, if the calculated distance is lower then I will set the manual distance to calculated distance. If it was lower than I would set another variable cluster to the iteration the loop is on to assign which data point that cluster matches.

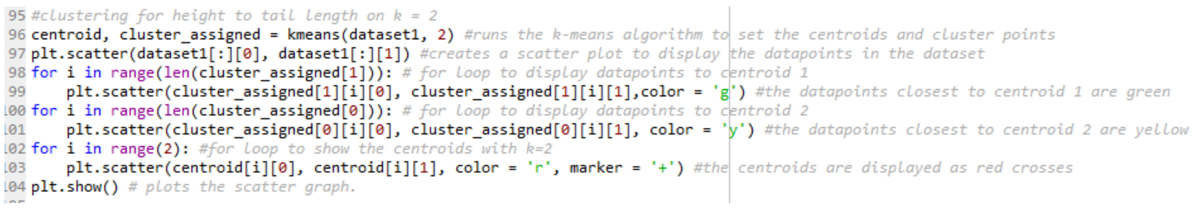
This is the Assign part of the K-means algorithm next is to move onto the update step where I will update the new centroid points based on the cluster. How I did this is create a new centroid array that will be used to store the new centroid points and I will go through another nested for loop which will iterate K amount of times. In this loop I will calculate the sum of the points in each cluster and find the average of all those points. The average for x and y will be the new coordinates for my new centroid points. The new centroid points will then be stored in the array just made centroid and this is where I will make use of the Euclidean distance function again to compute the distance between the old centroid plot and the new centroid plot. The idea of this is to minimise the distance between the two plots to be 0. This takes me back to the while loop from the start of the function where I have a while loop Boolean expression, if both the old centroid and the new centroid matched each other I would set the end call to True which would then end the while loop and return the centroid values and the cluster_assigned array, both values will be used to create scatter plots to display the K-means algorithm.

This is the idea of this algorithm, I will keep iterating the function until the centroids match each other and the final cluster will be complete.



```
#main
dataset1 = get_data(1) #gets the dataset for height and tail length
dataset2 = get_data(2) #gets the dataset for height and leg length

```

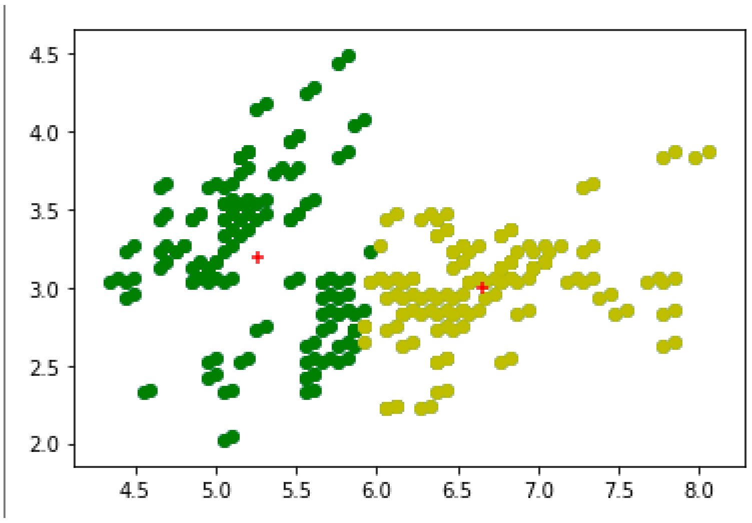
The main is where we output the results for the K-means algorithm, for the first part we will make use of the get_data() function to create datasets so we can use the data for the K-means algorithm. Dataset1 will hold the data for height and tail length and Dataset2 will hold the data for height and leg length.



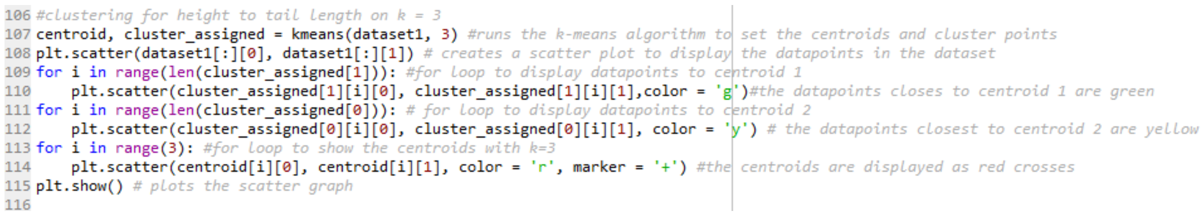
```
#clustering for height to tail length on k = 2
centroid, cluster_assigned = kmeans(dataset1, 2) #runs the k-means algorithm to set the centroids and cluster points
plt.scatter(dataset1[:][0], dataset1[:][1]) #creates a scatter plot to display the datapoints in the dataset
for i in range(len(cluster_assigned[1])): # for loop to display datapoints to centroid 1
    plt.scatter(cluster_assigned[1][i][0], cluster_assigned[1][i][1],color = 'g') #the datapoints closest to centroid 1 are green
for i in range(len(cluster_assigned[0])): # for loop to display datapoints to centroid 2
    plt.scatter(cluster_assigned[0][i][0], cluster_assigned[0][i][1], color = 'y') #the datapoints closest to centroid 2 are yellow
for i in range(2): #for loop to show the centroids with k=2
    plt.scatter(centroid[i][0], centroid[i][1], color = 'r', marker = '+') #the centroids are displayed as red crosses
plt.show() # plots the scatter graph.
```

For my first graph I make use of the first dataset where I will be handling the data between height and tail length, I will be using K = 2 so there will be two iterations of the K-means clustering. I first plot the scatter graph with all the points for height and tail length I then apply all the clusters from the K-means function and display the scatters in different colours. Finally I then output the centroids at the final points for each iteration.

This is the output:



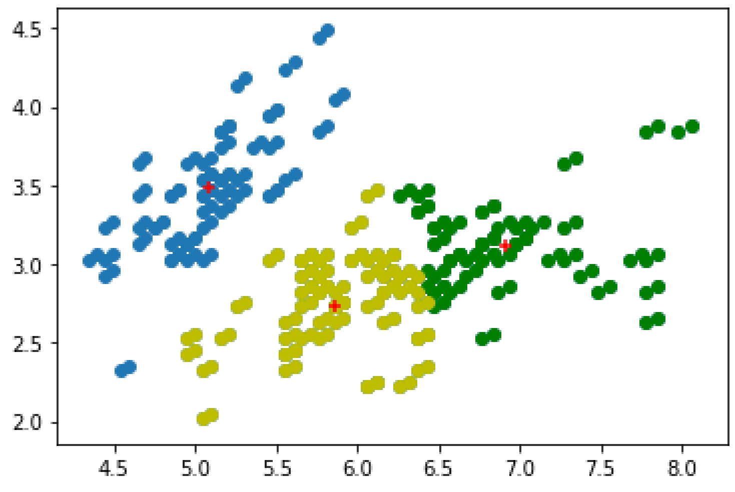
The red crosses indicate the centroid and the green and yellow blobs indicate the cluster to their respectful centroid.



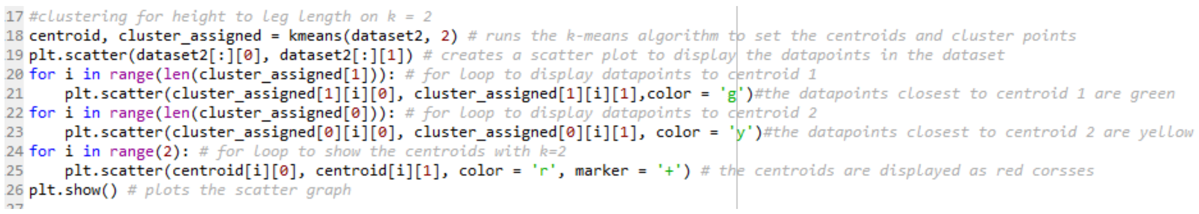
```
#clustering for height to tail length on k = 3
centroid, cluster_assigned = kmeans(dataset1, 3) #runs the k-means algorithm to set the centroids and cluster points
plt.scatter(dataset1[:][0], dataset1[:][1]) # creates a scatter plot to display the datapoints in the dataset
for i in range(len(cluster_assigned[1])): #for Loop to display datapoints to centroid 1
    plt.scatter(cluster_assigned[1][i][0], cluster_assigned[1][i][1],color = 'g')#the datapoints closes to centroid 1 are green
for i in range(len(cluster_assigned[0])): # for Loop to display datapoints to centroid 2
    plt.scatter(cluster_assigned[0][i][0], cluster_assigned[0][i][1], color = 'y') # the datapoints closest to centroid 2 are yellow
for i in range(3): #for Loop to show the centroids with k=3
    plt.scatter(centroid[i][0], centroid[i][1], color = 'r', marker = '+') #the centroids are displayed as red crosses
plt.show() # plots the scatter graph
```

For this plot I’m working again with the same dataset, which is the height and tail length but instead of working with two iterations I’m using three instead. Displaying the scatter plot is the same as before it’s just the change is in the final for loop where I apply the centroids, instead of it iterating twice it will iterate three times to match k.

This is the output:



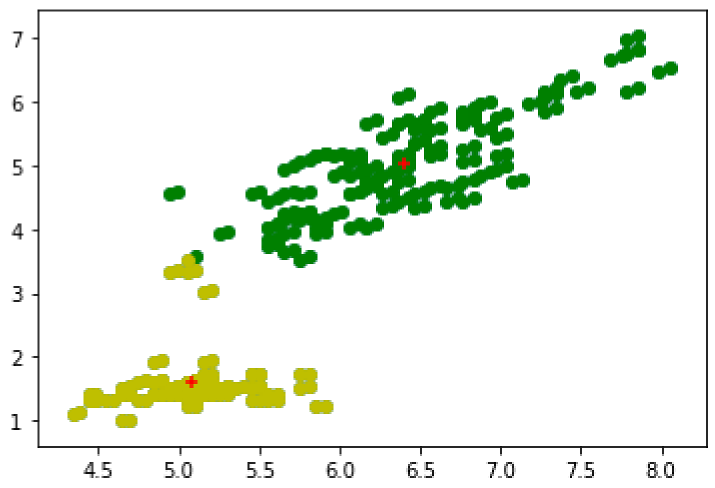
This time we have three different colours to express the cluster while also showing three different centroid positions. Again each colour, green, yellow and blue match to their corresponding centroid in their cluster.



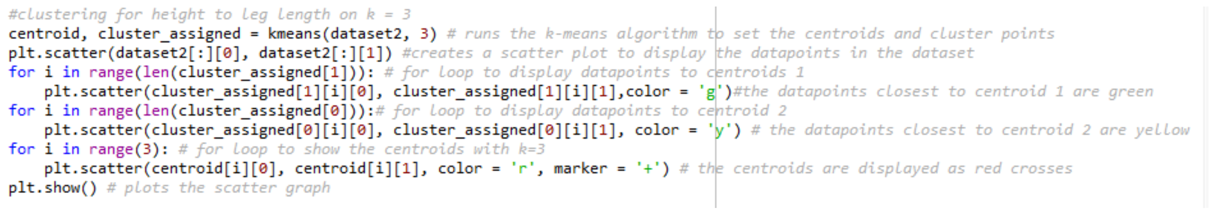
```
#clustering for height to leg length on k = 2
centroid, cluster_assigned = kmeans(dataset2, 2) # runs the k-means algorithm to set the centroids and cluster points
plt.scatter(dataset2[:][0], dataset2[:][1]) # creates a scatter plot to display the datapoints in the dataset
for i in range(len(cluster_assigned[1])): # for loop to display datapoints to centroid 1
    plt.scatter(cluster_assigned[1][i][0], cluster_assigned[1][i][1],color = 'g')#the datapoints closest to centroid 1 are green
for i in range(len(cluster_assigned[0])): # for loop to display datapoints to centroid 2
    plt.scatter(cluster_assigned[0][i][0], cluster_assigned[0][i][1], color = 'y')#the datapoints closest to centroid 2 are yellow
for i in range(2): # for loop to show the centroids with k=2
    plt.scatter(centroid[i][0], centroid[i][1], color = 'r', marker = '+') # the centroids are displayed as red corsses
plt.show() # plots the scatter graph
```

For the second dataset, I will be working on the data points that involve height and leg length. To start off same as before I will be having K = 2 so this K-means algorithm will iterate two times and show two different clusters, one being yellow and one being green and show the respected centroid in that cluster.

This is the output:



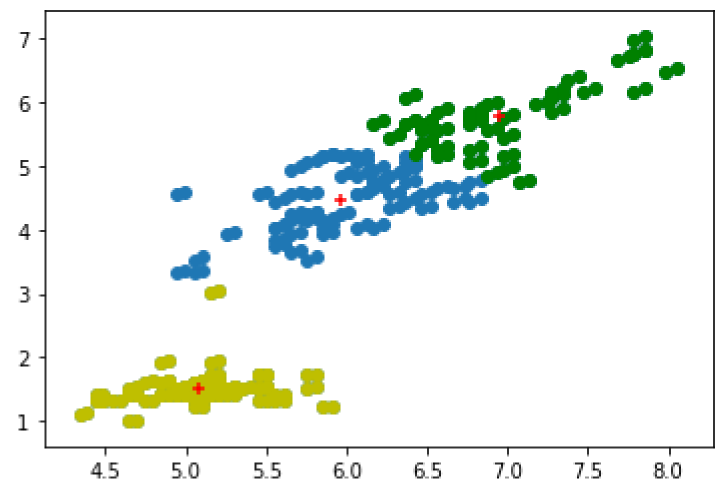
Different scatter plot now both showing the optimal cluster for both centroids.



```
#clustering for height to leg length on k = 3
centroid, cluster_assigned = kmeans(dataset2, 3) # runs the k-means algorithm to set the centroids and cluster points
plt.scatter(dataset2[:][0], dataset2[:][1]) #creates a scatter plot to display the datapoints in the dataset
for i in range(len(cluster_assigned[1])): # for loop to display datapoints to centroids 1
    plt.scatter(cluster_assigned[1][i][0], cluster_assigned[1][i][1],color = 'g')#the datapoints closest to centroid 1 are green
for i in range(len(cluster_assigned[0])):# for loop to display datapoints to centroid 2
    plt.scatter(cluster_assigned[0][i][0], cluster_assigned[0][i][1], color = 'y') # the datapoints closest to centroid 2 are yellow
for i in range(3): # for loop to show the centroids with k=3
    plt.scatter(centroid[i][0], centroid[i][1], color = 'r', marker = '+') # the centroids are displayed as red crosses
plt.show() # plots the scatter graph
```

For the last graph I will be using the dataset 2 again for height and leg length but instead of two degrees I will be using three degrees so the K-means algorithm will iterate three times. Just like before the final loop is three instead of two because that's how many centroids are gonna be shown in the scatter plot.

This is the output:



Again you can see that we have three distinct clusters split up with their corresponding centroids. All clusters are expressed with colours yellow, blue and green.

Reference List

1. Dabbura, I. K-means Clustering :Algorithm, (September 17, 2018_ Applications, Evaluation Methods, and Drawbacks
2. : https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a
3. Kaushik,M., Mathur, B. (June 2014) International Journal of Software & Hardware Research in Engineering Volume 2 Issue 6 'Comparative Study of K-Means and Hierarchical Clustering Techniques' ISSN No: 2347-4890 page 95. https://s3.amazonaws.com/academia.edu.documents/48663588/2014_Comparative_Study_of_K-Means_and.pdf?response-content-disposition=inline%3B%20filename%3DComparative_Study_of_K-Means_and_Hierarc.pdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWOWYYGZ2Y53UL3A%2F20191128%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20191128T035607Z&X-Amz-Expires=3600&X-Amz-SignedHeaders=host&X-Amz-Signature=4586ad3747572dd89c7619364ec9ca8a11e3c7c94b4c5497984ed7f6407f0250