Harry Langham
LAN15624847 CMP3751M

Machine Learning Assignment 1 Report

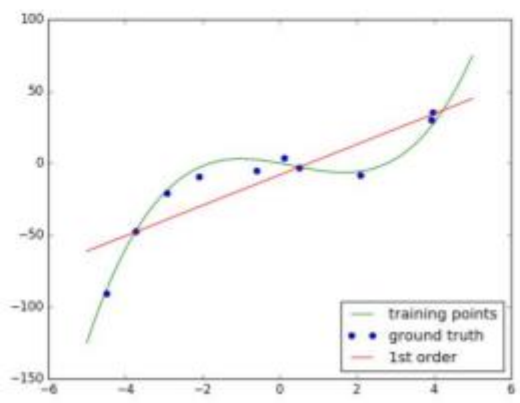
Task 1:

Section 1.1 Description of Polynomial Regression:

Polynomial regression is used for when a linear regression is under-fitted for a set of data that doesn't have a linear relationship between x and y.

Equation:

$$\hat{y}_n = w_0 + \sum_{i+1}^{d} w_i \; x_n^i$$



This graph will be used as an example, the set of data is related and has a correlation but its not in a linear relationship so a linear regression wouldn't fit the data very well. Therefore, Polynomial regression is used as that is more fitting to this type of data, on the graph both linear and polynomial regression is shown. The green line follows the data much more clearly than the red line as its polynomial and can fit anon-linear relationship better than under-fitting the relationship with a linear line.

R-Square

Using R-Square we can find out how close the regression line fits the data. The best fit you can get is at a R-Square value of 1. This is found using the equation:

$$R^2 = 1 - \frac{\text{Residual sum of squares}}{\text{Total sum of squares}} = 1 - \frac{\sum_{n=1}^{N}(\hat{y}_n - y_n)^2}{\sum_{n=1}^{N}(y_n - \bar{y})^2}$$

This determines how much of the total variation in y is explained by the variation in x. In the diagram shown above, the R-Squared value would be low for the linear regression and a lot closer to 1 for the polynomial regression.

Expression of SSE

$$\epsilon = \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_N \end{bmatrix}$$

L_2 norm square

$$E = \epsilon_1^2 + \epsilon_2^2 + \ldots + \epsilon_N^2 = \sum_{n}^{N} \epsilon_n^2 = \|\epsilon\|_2^2 ,$$

where $\epsilon_n = y_n - \hat{y}_n = y_n - w_0 - w_1 x_n$. 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

We see SSE is a quadratic function of w_0 and w_1

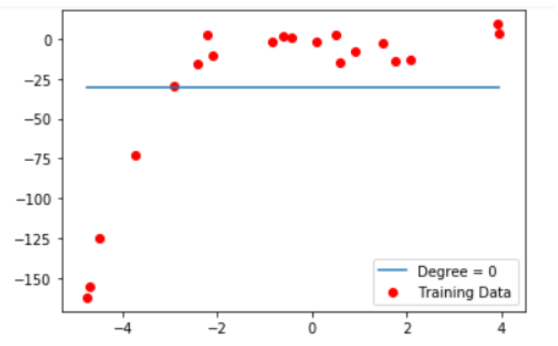
To obtain the R-Square you need to complete the SSE (Sum of squared errors) first and this is found with the distance between the linear line and the n-th data point. This can be found in the equation on the left. This is used in linear regression to minimise the sum of errors of the vertical distances of data points to the line.

In the end, training set error goes down over time and test set error goes up due to the amount of testing that is done with the data. The overall performance of the training data was better after each degree was calculated and the test data was used, in the end the training set had a much lower RMSE than the test set whereas at the start of the learning it was a lot higher.

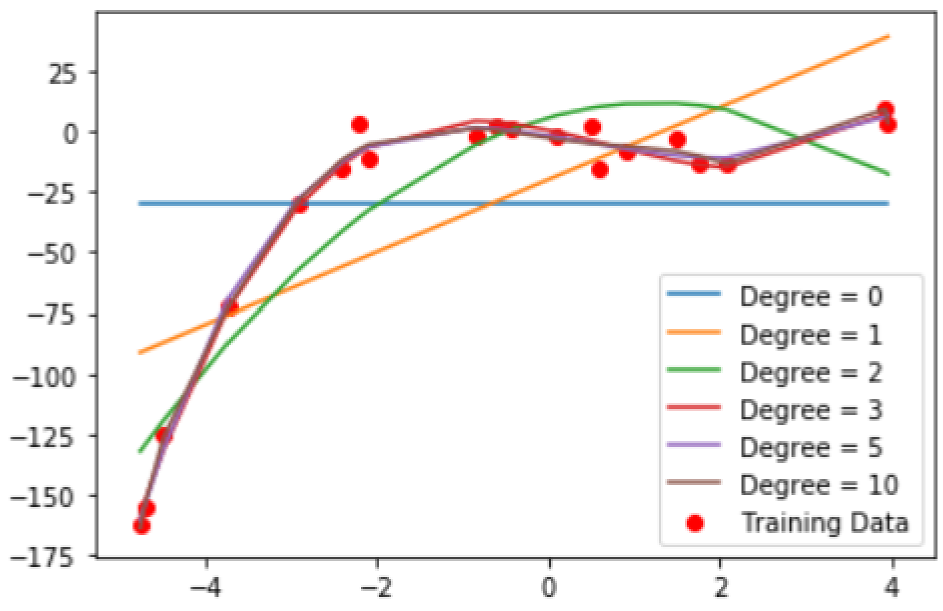
Section 1.2 Implementation of Polynomial Regression:

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

def main():
    dataset = pd.read_csv("CMP3751M_ML_Assignment 1_Task1 - dataset - pol_regression.csv", header=None)
    dataset = np.array(dataset)
    dataset = np.delete(dataset, 0, axis=0) #Remove column headings
    dataset = dataset.astype(np.float) #Converts the string to a float
    dataset = dataset[:,1:] #Remove the first column of index numbers

    dataset = dataset[dataset[:,0].argsort()] #Sorted based on the x value column

    x = dataset[:,0]
    y = dataset[:,1]

    degrees = [0,1,2,3,5,10]

    #Regressing and plotting polynomials to the degrees 0,1,2,3,5,10
    for i in range(len(degrees)):
        plt.clf()
        weights, predictedY = pol_regression(x, y, degrees[i])
        plt.plot(x, predictedY)
        plt.scatter(x, y, color="red")
        plt.legend(("Degree = " + str(degrees[i]), "Training Data"), loc = 'lower right')
        plt.show()
```

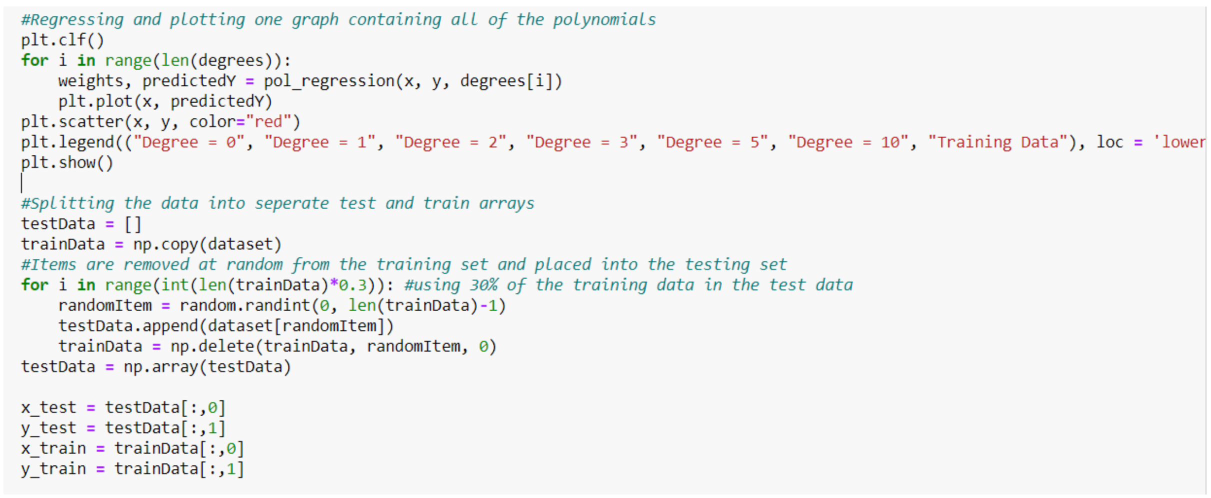
Plotting the graph gives me this:



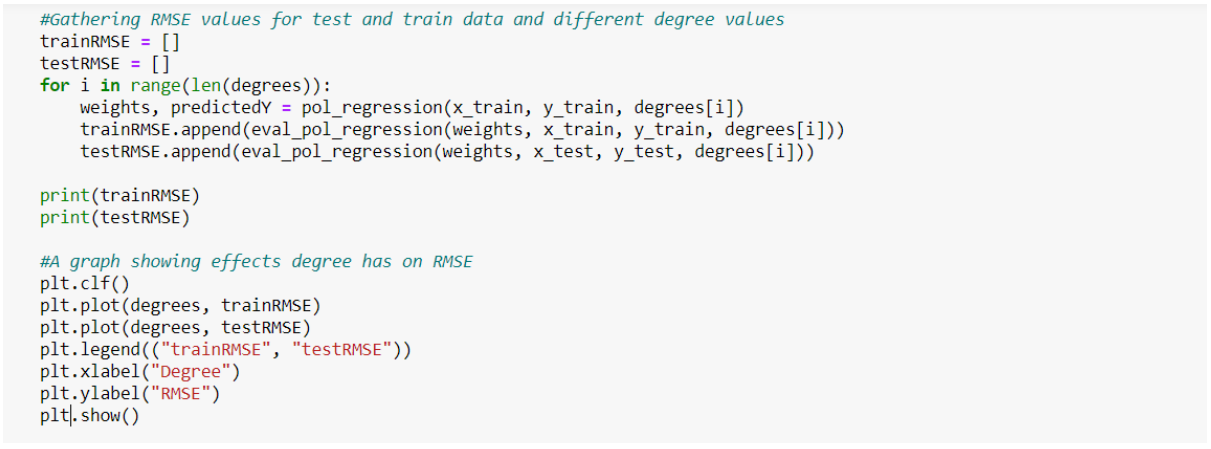
This is with 0 degree, so it will have a straight line because it's not fitted well. Instead of putting all of the individual graphs on this document I have included the graph with all of the lines on.



Showing all of the degrees fitted to the training data, you can see that as the degrees increase, the fit becomes a lot better as with 1 degree, the fit is linear and until degree 3, it doesn't follow the full trend of the data. For this I would choose 3 as my degree as it fits the best without overfitting which can happen with a high degrees.



```
#Regressing and plotting one graph containing all of the polynomials
plt.clf()
for i in range(len(degrees)):
    weights, predictedY = pol_regression(x, y, degrees[i])
    plt.plot(x, predictedY)
plt.scatter(x, y, color="red")
plt.legend(("Degree = 0", "Degree = 1", "Degree = 2", "Degree = 3", "Degree = 5", "Degree = 10", "Training Data"), loc = 'lower
plt.show()

#Splitting the data into seperate test and train arrays
testData = []
trainData = np.copy(dataset)
#Items are removed at random from the training set and placed into the testing set
for i in range(int(len(trainData)*0.3)): #using 30% of the training data in the test data
    randomItem = random.randint(0, len(trainData)-1)
    testData.append(dataset[randomItem])
    trainData = np.delete(trainData, randomItem, 0)
testData = np.array(testData)

x_test = testData[:,0]
y_test = testData[:,1]
x_train = trainData[:,0]
y_train = trainData[:,1]
```



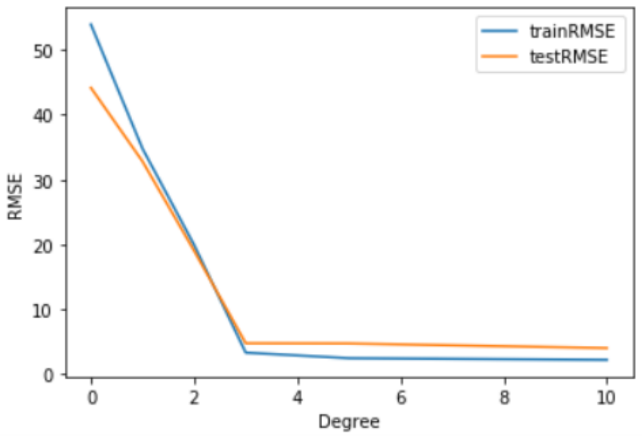
```
#Gathering RMSE values for test and train data and different degree values
trainRMSE = []
testRMSE = []
for i in range(len(degrees)):
    weights, predictedY = pol_regression(x_train, y_train, degrees[i])
    trainRMSE.append(eval_pol_regression(weights, x_train, y_train, degrees[i]))
    testRMSE.append(eval_pol_regression(weights, x_test, y_test, degrees[i]))

print(trainRMSE)
print(testRMSE)

#A graph showing effects degree has on RMSE
plt.clf()
plt.plot(degrees, trainRMSE)
plt.plot(degrees, testRMSE)
plt.legend(("trainRMSE", "testRMSE"))
plt.xlabel("Degree")
plt.ylabel("RMSE")
plt.show()
```

This shows the train and test RMSE which I plotted. The trend is that as the degrees go up, the RMSE goes down but stays around as soon as it goes to a polynomial regression fit because you can't remove any more of the error.

```python
def pol_regression(features_train,  y_train, degree):
    #calculating weights using least squares solution
    vanderVector = np.vander(features_train, degree + 1)
    vanderTransposed = vanderVector.transpose()
    weights = (np.linalg.inv(vanderTransposed.dot(vanderVector))).dot(vanderTransposed).dot(y_train)
    #Using training x values to calculate predicted y values
    predictedY = vanderVector.dot(weights)

    #The predicted y values can be used to calculate RMSE with the actual y values
    return weights, predictedY
```

Section 1.3 Evaluation:

```python
def eval_pol_regression(coefficients, x, y, degree):
    #x has to be in a vander vector to calculate y
    vanderVector = np.vander(x, degree+1)
    #using new x values and coefficients, predicted y is calculated
    predictedY = vanderVector.dot(coefficients)
    #predicted evaluated by RMSE from y value
    xMinusY = predictedY - y
    rmse = math.sqrt(sum(np.square(xMinusY))/len(y))
    return rmse




main()
#Runs the main script to out the graphs in the output window
```



This shows the train and test error as the degrees increase. The error (RMSE) remains the same after degree 3 so that is why I would use that as there is no risk of using too high a degree and overfitting the model. Degree 0 doesn't use much of the data and has a very underfitting line. Degree 1 is a linear line and fits the data, but a polynomial regression line is needed instead to fully fit the line. Degree 2 fits it lot better but because it's will eventually go down if you extrapolated the line for a bigger set of results whereas 3, 5 and 10 are fitting well.

Task 2:

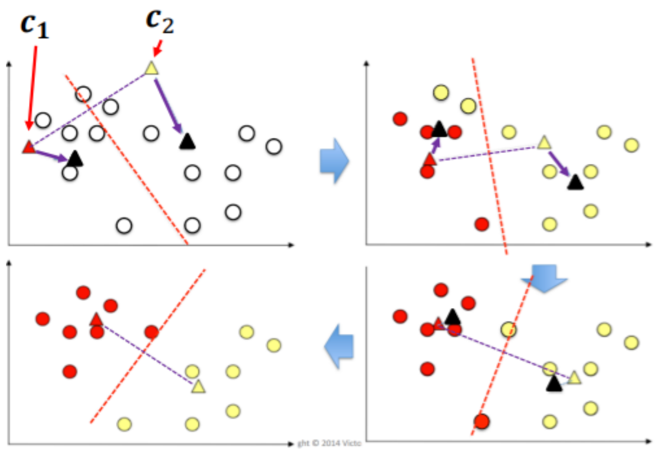
Section 2.1 Description of K-Means Clustering:

K-Means clustering is a method of getting data points into clusters using centroids that will then be put on a diagram. The clusters gather the data points in relation to value/position, so if you were to have the different clusters data points in a colour then they would have a clear grouping. In the lecture materials, the K-means clustering is when you enter the data points as a vector/array and place the centroids at random locations until convergence. This is when all the points are assigned to a cluster (for cluster = 1 to cluster = K), and during this the centroids are assigned to the mean of all points in a cluster and repeated until the cluster assignments don't change. This loop updates the cluster assignment and centroid assignment with the update step until it doesn't change as stated before.

Euclidean distance is used in K-Means Clustering and the equation for this is:

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

This calculates the distance between the data points and centroids. This finds the nearest centroid cj in the form of $argmin_j D(X_i, c_j)$



This shows the clustering and the use of centroids to create the clusters (seen as red and yellow). This groups them into 2 different clusters which is good to help train using certain sets in the area of machine learning.

$$J(C) = \sum_{j=1}^{K} \sum_{x_i \to c_j} D(x_i, c_j)$$

Finally you have the minimising aggregate cluster distance and this is the over-arching equation for creating clusters and the total squared distance from the data point to centre of its cluster.

Strengths and weaknesses:

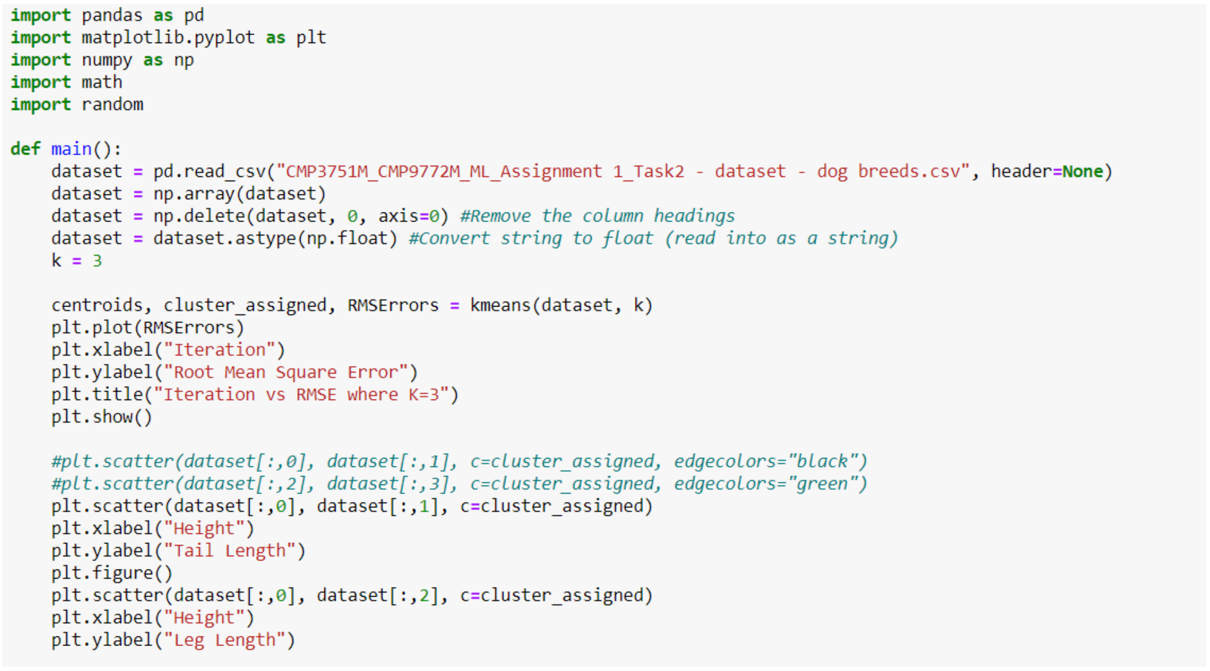
The strengths:

- The computational time for larger variables and data points are quite fast so can keep number of k small
- It produces tight clusters for a relatively large amount of data points

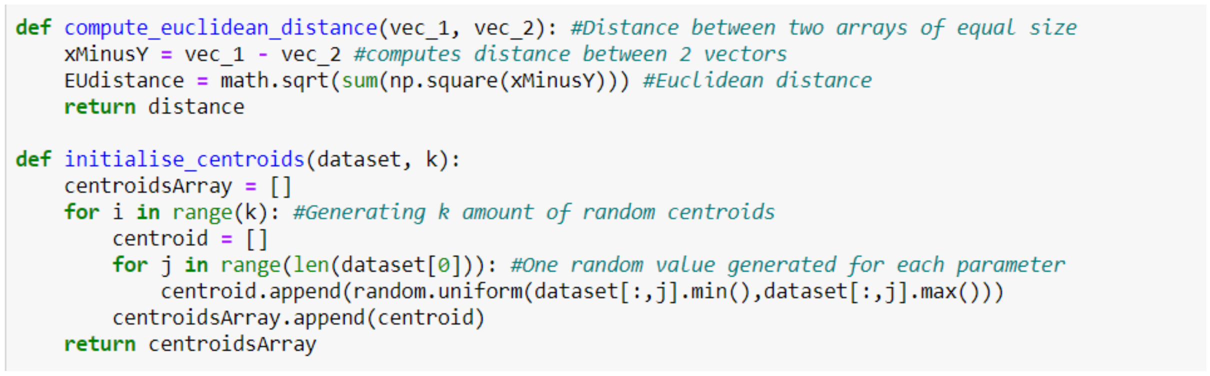
Weaknesses:

- Difficult to predict K-value as it can be hard to think, can be computed roughly with a diagram
- Doesn’t work well with clusters of different size and density.

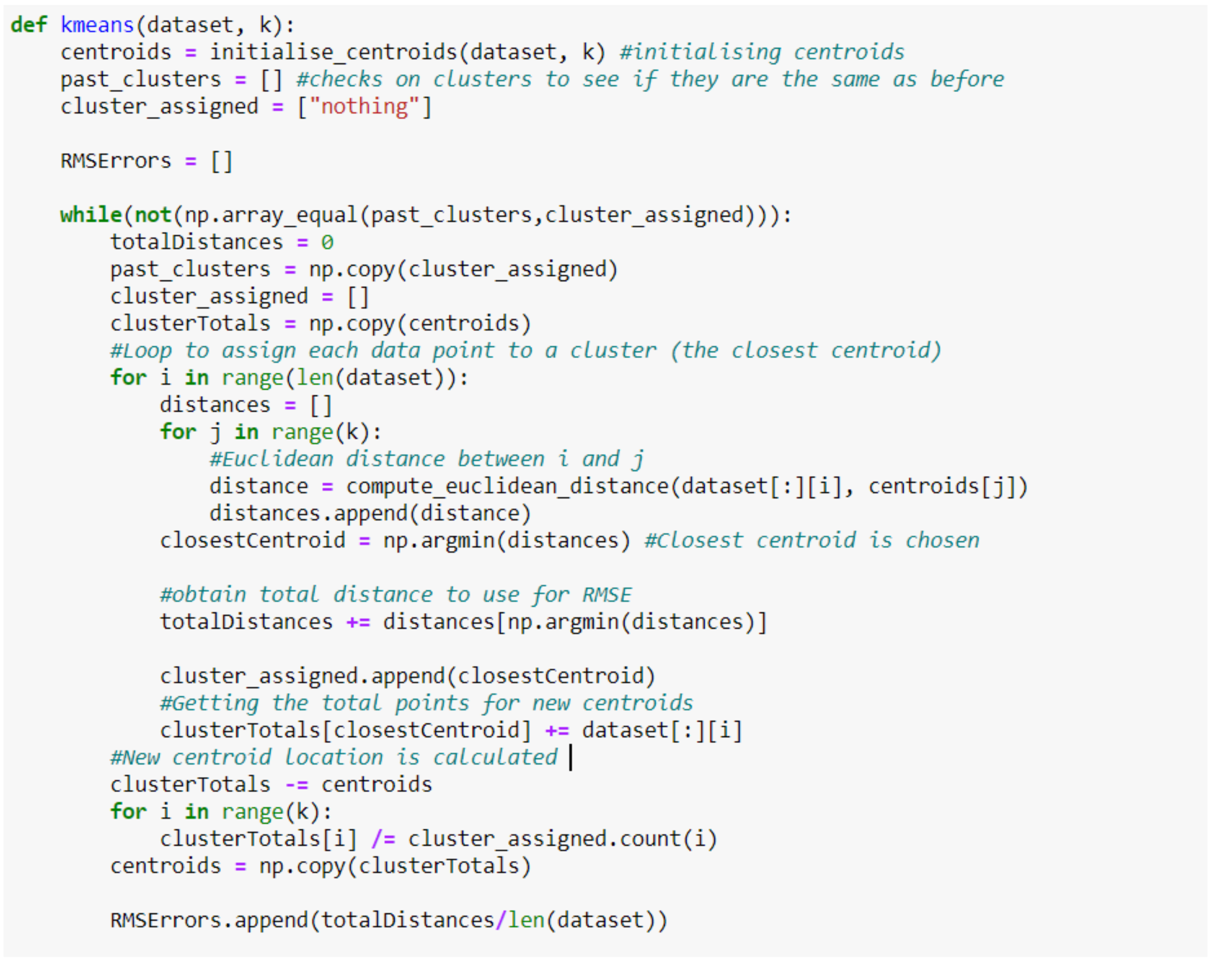
Section 2.2 Implementation of the K-Mean Clustering:



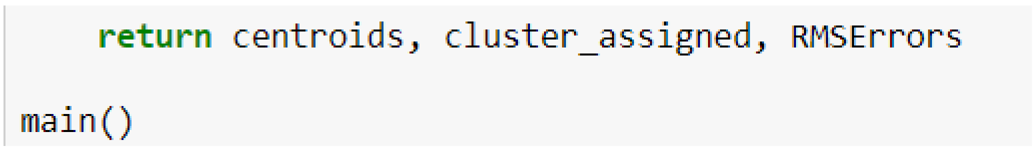
```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
import random

def main():
    dataset = pd.read_csv("CMP3751M_CMP9772M_ML_Assignment 1_Task2 - dataset - dog breeds.csv", header=None)
    dataset = np.array(dataset)
    dataset = np.delete(dataset, 0, axis=0) #Remove the column headings
    dataset = dataset.astype(np.float) #Convert string to float (read into as a string)
    k = 3

    centroids, cluster_assigned, RMSErrors = kmeans(dataset, k)
    plt.plot(RMSErrors)
    plt.xlabel("Iteration")
    plt.ylabel("Root Mean Square Error")
    plt.title("Iteration vs RMSE where K=3")
    plt.show()

    #plt.scatter(dataset[:,0], dataset[:,1], c=cluster_assigned, edgecolors="black")
    #plt.scatter(dataset[:,2], dataset[:,3], c=cluster_assigned, edgecolors="green")
    plt.scatter(dataset[:,0], dataset[:,1], c=cluster_assigned)
    plt.xlabel("Height")
    plt.ylabel("Tail Length")
    plt.figure()
    plt.scatter(dataset[:,0], dataset[:,2], c=cluster_assigned)
    plt.xlabel("Height")
    plt.ylabel("Leg Length")
```



```python
def compute_euclidean_distance(vec_1, vec_2): #Distance between two arrays of equal size
    xMinusY = vec_1 - vec_2 #computes distance between 2 vectors
    EUdistance = math.sqrt(sum(np.square(xMinusY))) #Euclidean distance
    return distance

def initialise_centroids(dataset, k):
    centroidsArray = []
    for i in range(k): #Generating k amount of random centroids
        centroid = []
        for j in range(len(dataset[0])): #One random value generated for each parameter
            centroid.append(random.uniform(dataset[:,j].min(),dataset[:,j].max()))
        centroidsArray.append(centroid)
    return centroidsArray
```

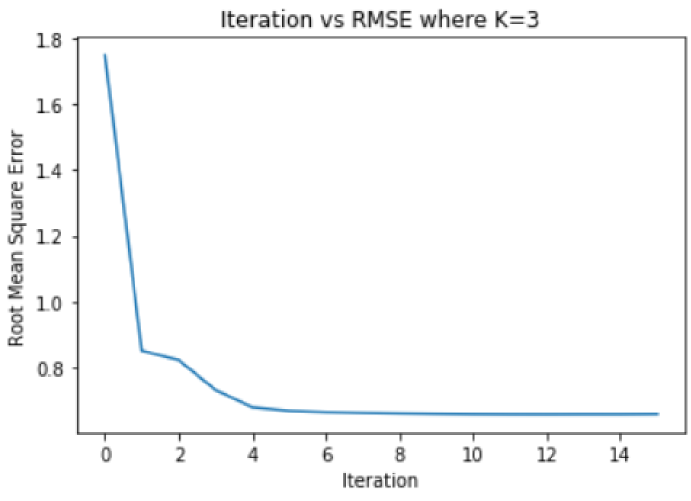


```
def kmeans(dataset, k):
    centroids = initialise_centroids(dataset, k) #initialising centroids
    past_clusters = [] #checks on clusters to see if they are the same as before
    cluster_assigned = ["nothing"]

    RMSErrors = []

    while(not(np.array_equal(past_clusters,cluster_assigned))):
        totalDistances = 0
        past_clusters = np.copy(cluster_assigned)
        cluster_assigned = []
        clusterTotals = np.copy(centroids)
        #Loop to assign each data point to a cluster (the closest centroid)
        for i in range(len(dataset)):
            distances = []
            for j in range(k):
                #Euclidean distance between i and j
                distance = compute_euclidean_distance(dataset[:][i], centroids[j])
                distances.append(distance)
            closestCentroid = np.argmin(distances) #Closest centroid is chosen

            #obtain total distance to use for RMSE
            totalDistances += distances[np.argmin(distances)]

            cluster_assigned.append(closestCentroid)
            #Getting the total points for new centroids
            clusterTotals[closestCentroid] += dataset[:][i]
        #New centroid location is calculated
        clusterTotals -= centroids
        for i in range(k):
            clusterTotals[i] /= cluster_assigned.count(i)
        centroids = np.copy(clusterTotals)

        RMSErrors.append(totalDistances/len(dataset))
```

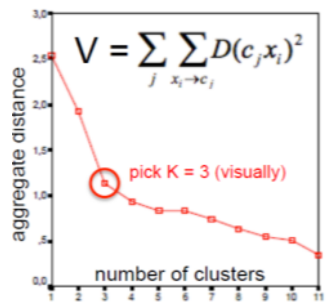


```
    return centroids, cluster_assigned, RMSErrors

main()
```

This is the full implementation of the K-Means Clustering, the output is below.

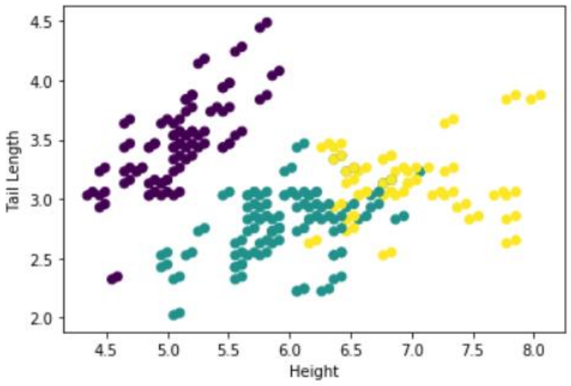


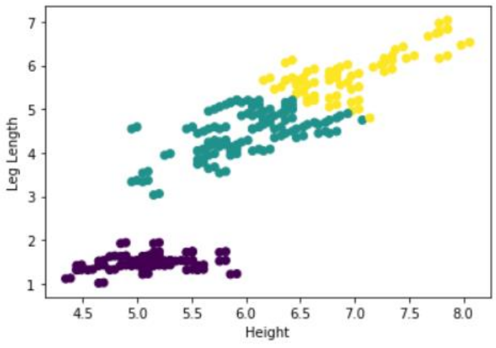
This shows that the RMSE error decreases as the iterations are ran through, with the RMSE staying around the same from 4 iterations onwards. The number of centroids is 3 so when they are iterated on to optimise the position of the centroids, the RMSE reduces massively to start with but some error must remain. This shows that the clusters should be 3 as the error with 3 clusters has the same error as more clusters so it will fit the data better than more clusters as it might overfit the data points.



This is the optimal number of clusters, as seen in the lecture this is the best choice. This diagram is similar to the one created in the first diagram from the implementation so must be the best option.

Data points plotted:





This is the dog breed data plotted with tail length and height on the left diagram. You can clearly see the 3 centroids, with multiple data points in the 3 different colours. As it's a 2d diagram the data points overlap but if you had a 3d diagram with the same data points then they would be separated in the z axis and be easily identifiable. Because in this algorithm, the centroids wouldn't allow the data points to be mixed about as much as the left diagram.

The right diagram plots the leg length against the height with the 3 centroids again, there is less variance between the data points. K is 3 for both diagrams, and the data values are moved into a cluster which are based on value and keeps the number of centroids low. These clusters are iterated on to optimise the position and when they are successful and there is no change then they outputted on the diagrams above. This creates a tighter correlation than other algorithms and is quite popular as a simple machine learning algorithm.