2019

Machine Learning

ASSIGNMENT 1

BY GABRIELLA DI GREGORIO 15624188

Contents

Task 1

1.1 Description of Polynomial Regression

Linear regression is a commonly used type of predictive analysis which attempts to model the relationship between two variables by fitting a linear equation to observed data. It is used for finding a linear relationship between a target and one or more predictions so one variable is considered to be an explanatory variable, and the other is considered to be a dependent variable. (Stat.yale.edu, n.d.) Polynomial Regression is used when two variables in a data set are correlated but the relationship is not linear. So, even if a linear regression line would not fit the data well, it does not mean there is no correlation – polynomial regression can be used to fit a polynomial line to achieve a minimum error or minimum cost function. (Pant, 2019) This can be used to create models that can predict data. A dataset can be split into a training set and a test set. The training set is used to teach the model a general pattern, which polynomial regression is used to find the best pattern to use for predictions. The test set can then be used to assess how well the model can make predictions based on the regression line.

Error Function used for Regression:

Mean-Squared Error (MSE):

This is the most common Loss Function and is calculated by taking the difference between the model's predictions and the real values, squaring that, and then averaging it out across the entire dataset.

The Equation (where N is the number of samples being tested against):

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Figure 1: MSE (Seif, 2019)

This function is good for ensuring that the trained model has no outlier predictions with huge errors, since the MSE puts larger 'weight' on these errors due to the squaring. (Seif, 2019) However, if the model makes a poor prediction, the squaring magnifies the error. Despite this, outliers are often unimportant in most practical cases since the aim is usually a well-rounded model that performs sufficiently for the majority of the time.

Mean Absolute Error (MAE):

This Function can be described very similarly to MSE but provides almost exactly opposite properties. It is calculated by taking the difference between the model's predictions and the real values, applying the absolute value to that difference, then averaging it out across the dataset.

The **Mean Absolute Error**(MAE) is the average of all absolute errors. The formula is:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|\boldsymbol{x}_i - \boldsymbol{x}|$$

Where:

- n = the number of errors,
- Σ = summation symbol (which means "add them all up"),
- $|x_i - x|$ = the absolute errors.

Figure 2: MAE (Statistics How To, 2016)

The advantage of MAE is opposite to the disadvantage of MSE – all the errors will be weighted on the same linear scale. Therefore, unlike MSE, outliers will not be outweighed and the loss function provides a generic even measure of how well the model performs. However, if outliers are of importance, MAE won't be as effective since the large errors from the outliers will be weighted the same as lower errors. This could result in a model which is mostly accurate, but prone to making occasional poor predictions.

Huber Loss:

Huber loss is less sensitive to outliers in data than the squared error loss and is differentiable at 0. It is basically absolute error, which becomes quadratic when error is small. How small that error has to be to make it quadratic depends on a hyperparameter, δ (delta), which can be tuned. Huber loss approaches MAE when $\delta \sim 0$ and MSE when $\delta \sim \infty$ (large numbers). (Grover, 2018) This function effectively combines the best of both from MSE and MAE. Using the MAE for larger loss values mitigates the weight that is put on outliers so that a well-rounded model is still produced. At the same time, MSE is used for the smaller loss values to maintain a quadratic function near the centre. This has the effect of magnifying the loss values as long as they are greater than 1. Once the loss for those data points dips below 1, the quadratic function down-weights them to focus the training on the higher-error data points. (Seif, 2019)

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \leq \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

Figure 3: Huber Loss (Grover, 2018)

The choice of delta is critical because it determines what will be considered an outlier. Residuals larger than delta are minimized with L1 (which is less sensitive to large outliers), while residuals smaller than delta are minimized appropriately with L2. One big problem with using MAE for training of neural networks is its constantly large gradient, which can lead to missing minima at the end of training using gradient descent. For MSE, gradient decreases as the loss gets close to its minima, making it more precise. Huber Loss is useful in such cases, as it curves around the minima which decreases the gradient and it is more robust to outliers than MSE. Therefore, it combines good properties from both MSE and MAE. However, the problem with Huber loss is that it may be necessary to train hyperparameter delta which is an iterative process. (Grover, 2018)

Least-Squares Solution:

The most common method for fitting a regression line is the method of least-squares. This method calculates the best-fitting line for the observed data by minimizing the sum of the squares of the vertical deviations from each data point to the line (if a point lies on the fitted line exactly, then its vertical deviation is 0). Because the deviations are first squared, then summed, there are no cancellations between positive and negative values. (Stat.yale.edu, n.d.) This method is good for predicting where other points on the line might lie since the residuals can be treated as a continuous quantity where derivatives can be found. (Statistics How To, 2014) However, outliers can have a disproportionate effect since the line may fall closer to anomalous points than it should.

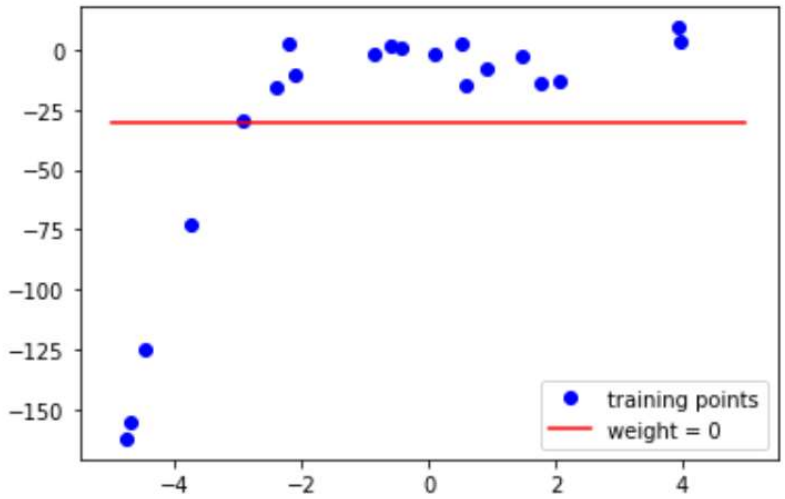
1.2 Implementation of Polynomial Regression

```
#CMP3751M Machine Learning Assignment 1
#Gabriella Di Gregorio 15624188
#Task 1.2

import pandas as pd
import numpy as np
import numpy.linalg as linalg
import matplotlib.pyplot as plt

#Reads in the data using pandas.read_csv because pandas.DataFrame.from_csv is outdated and discouraged.
Dataset = pd.read_csv('CMP3751M_ML_Assignment 1_Task1 - dataset - pol_regression.csv')

#Assings the x's in the data to the variable x, and the y's in the data to variable y
x = Dataset['x'].values
y = Dataset['y'].values
```

```
#function that does the feature expansion up to a certain degree for a given data set x
def getPolynomialDataMatrix(x, degree):
    #creates an array of 1s in the shape of x
    X = np.ones(x.shape)
    #stacks all the data into a 3D array (matrix)
    for i in range(1,degree + 1):
        X = np.column_stack((X, x ** i))
    return X
```

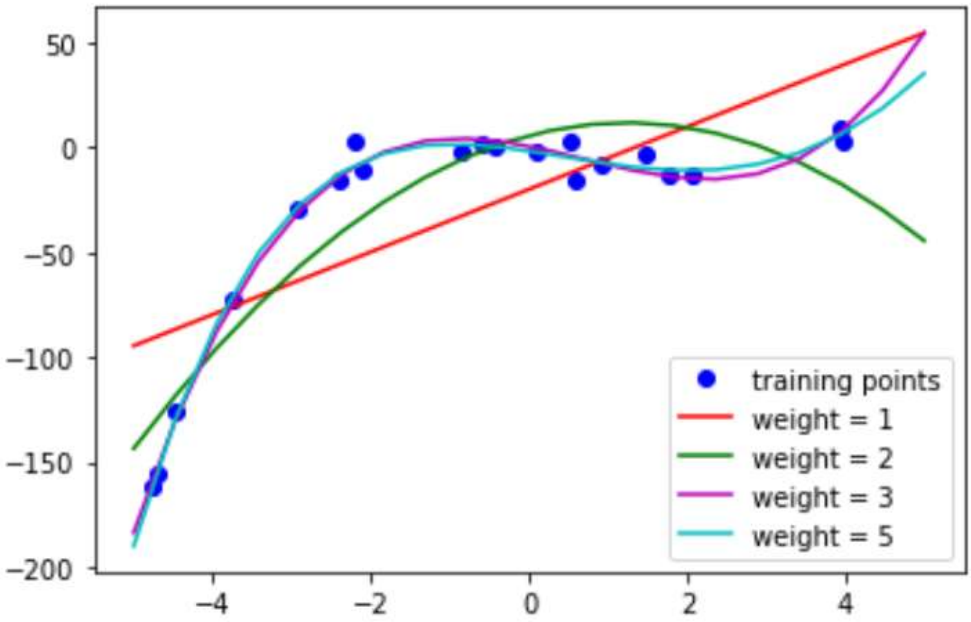
```
#function that computes the optimal values given the input data x, output data y and the desired degree of the polynomial.
#renamed from eval_pol_regression(parameters, x, y, degree)
def getWeightsForPolynomialFit(x,y,degree):
    X = getPolynomialDataMatrix(x, degree)

    #Transposes the matrix then multiplies it by the original
    XX = X.transpose().dot(X)
    #numyp.linalg.solve solves linear matrix equations
    w = np.linalg.solve(XX, X.transpose().dot(y))

    return w
```

```
#numpy.linspace returns evenly spaced numbers over a specified interval (Docs.scipy.org, 2019), between -5 and 5 in this case, within the length of x
x_plot = np.linspace(-5,5,len(x))

#plots the first graph using a weight of 0
plt.figure(1)
#plots the training points as blue dots (o's)
plt.plot(x,y,'bo')
#using the getWeightsForPolynomialFit function for 0 would give an empty array error
#since it remains the same, the mean of y is an alternative way to represent it
w0 = np.mean(y)
#gets the matrix between the specified interval of the degree of 0
X0 = getPolynomialDataMatrix(x_plot,0)
#multiplies the matrix by the mean of y (weight 0)
y0 = X0.dot(w0)
#plots the specified range for the x axis, the matrix multiplied by the weight for the y axis, and the colour to display the lines
plt.plot(x_plot,y0,'r')
#adds a key to show what the points and line(s) mean on the graph, and where to put it
plt.legend(('training points', 'weight = 0'), loc = 'lower right')
```



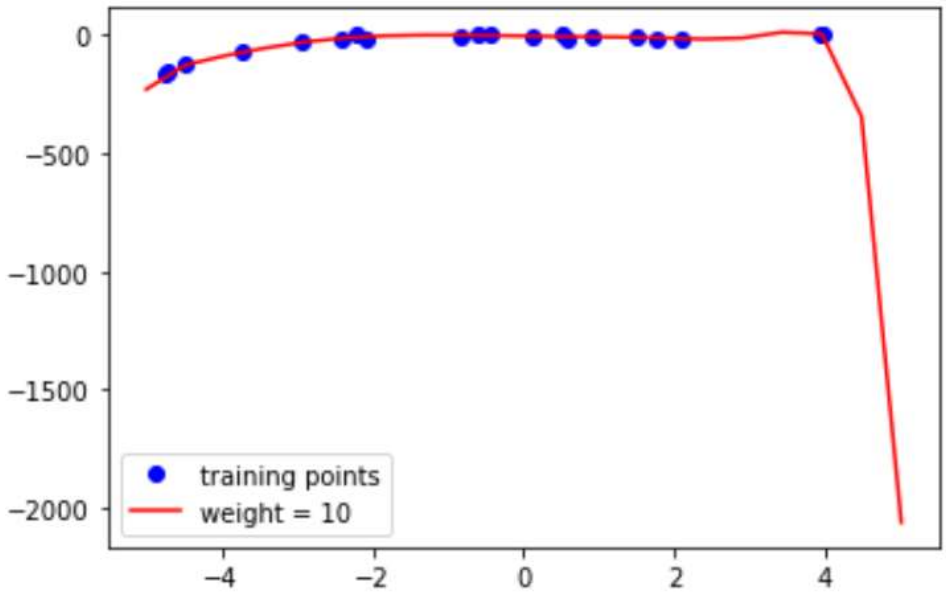
The degree of 0 simply represents the mean of the y's and is clearly an underfit. Underfitting refers to a model that can neither model the training data nor generalize to new data. An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data. (Brownlee, 2019)

```
#the second graph shows weights 1,2,3,and 5
plt.figure(2)
plt.plot(x,y,'bo')
#for all other weights, the function is used to calculate it by passing x, y and desired degree (1 in this case)
w1 = getWeightsForPolynomialFit(x,y,1)
#the rest is the same as for 0
X1 = getPolynomialDataMatrix(x_plot,1)
y1 = X1.dot(w1)
plt.plot(x_plot,y1,'r')

w2 = getWeightsForPolynomialFit(x,y,2)
X2 = getPolynomialDataMatrix(x_plot,2)
y2 = X2.dot(w2)
plt.plot(x_plot,y2,'g')

w3 = getWeightsForPolynomialFit(x,y,3)
X3 = getPolynomialDataMatrix(x_plot,3)
y3 = X3.dot(w3)
plt.plot(x_plot,y3,'m')

w5 = getWeightsForPolynomialFit(x,y,5)
X5 = getPolynomialDataMatrix(x_plot,5)
y5 = X5.dot(w5)
plt.plot(x_plot,y5,'c')
plt.legend(('training points', 'weight = 1', 'weight = 2', 'weight = 3', 'weight = 5'), loc = 'lower right')
```



Weight 1 is commonly known as the 'line of best fit' and is also an underfit and would lead to very inaccurate predictions but is more useful than 0 because it at least represents a pattern. Weights 3 and 5 are very hard to distinguish between, they both look almost like perfect fits. This could mean that either of them could be used in a model to predict accurate results and may also mean that a weight of 4 would be a happy medium.

```
#the weight of 10 could not be displayed on the same graph as the rest because it makes it too unclear
#the process for calculating/displaying is the same
plt.figure(3)
plt.plot(x,y,'bo')
w10 = getWeightsForPolynomialFit(x,y,10)
X10 = getPolynomialDataMatrix(x_plot,10)
y10 = X10.dot(w10)
plt.plot(x_plot,y10,'r')
plt.legend(('training points', 'weight = 10'), loc = 'lower left')
```

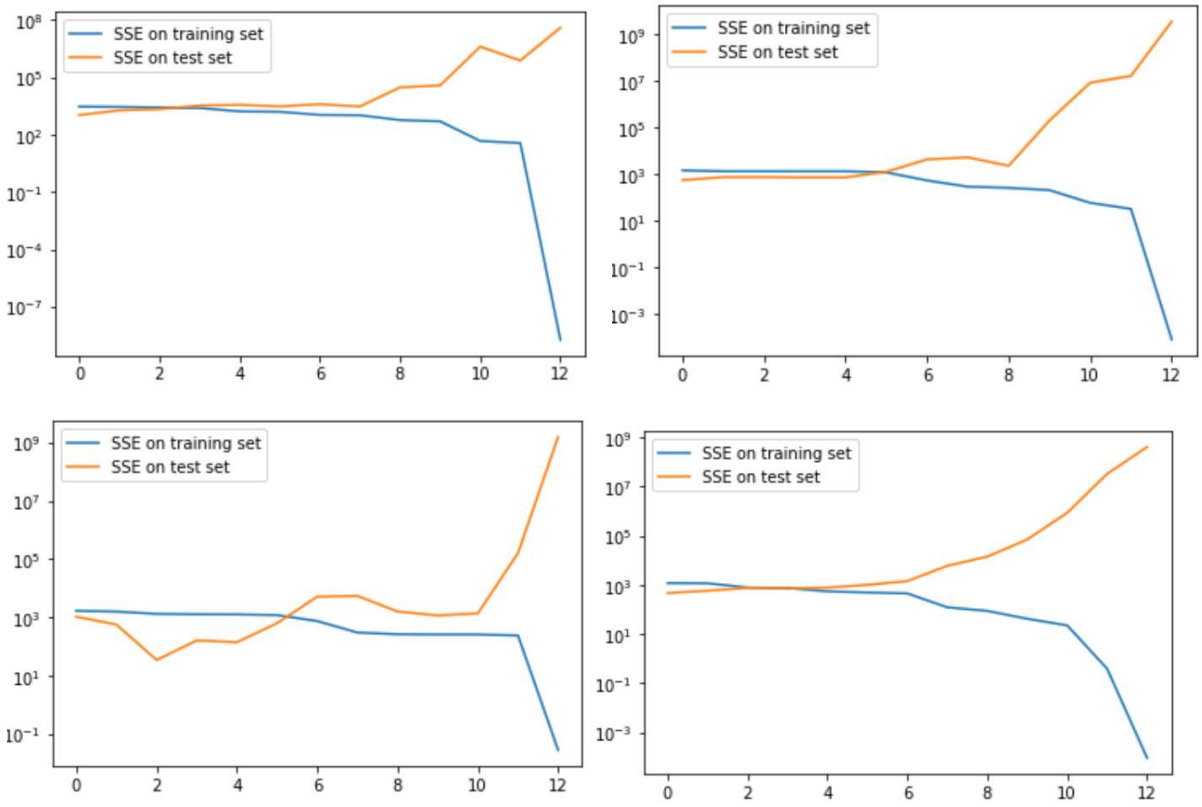


Using a degree of 10 is an example of an overfit. Overfitting refers to a model that models the training data too well. Overfitting happens when a model learns the detail and 'noise' in the training

data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the model's ability to generalize. (Brownlee, 2019) This means that the model would essentially be just memorising the points of the training data rather than learning an actual pattern to predict results. Weight 10 goes through every single point and drops off at 5 because there are no datapoints beyond that. This is not a good representation because it is not definite that a new dataset would have no points higher than 5.

1.3 Evaluation

```
#CMP3751M Machine Learning Assignment 1
#Gabriella Di Gregorio 15624188
#Task 1.3

#function to split the data into a training set and a testing set
def train_test_split(X, y, train_split):
    #the values are first randomly shuffled, so they change each time and the test is fair
    np.random.shuffle(X)
    np.random.shuffle(y)
    #initialises empty arrays to store the x and y train and test sets
    X_train = []
    X_test = []
    y_train = []
    y_test = []
    #goes through all of the values, and if they are after the length multiplied by the split, put them in the test data, if they are before it, put it in the train data
    for position in range(len(X)):
        if position >= len(X)*train_split:
            X_test.append(X[position])
            y_test.append(y[position])
        else:
            X_train.append(X[position])
            y_train.append(y[position])
    return X_train, y_train, X_test, y_test

#makes the data split into 70% training, and 30% testing
x_train, y_train, x_test, y_test = train_test_split(x, y, 0.7)

#numpy.squeeze remove single-dimensional entries from the shape of an array and numpy.as array converts the input to an array
x_train = np.squeeze(np.asarray(x_train))
y_train = np.squeeze(np.asarray(y_train))
x_test = np.squeeze(np.asarray(x_test))
y_test = np.squeeze(np.asarray(x_test))

#The following computes the squarred error on the training and test set for each of these polynomials and plots them as a function of the degree of the polynomial.
#Creates arrays of 0s with size 13 (14-1 because 14 is 70% of the data) by 1
SSEtrain = np.zeros((13,1))
SSEtest = np.zeros((13,1))

#for all values within the maximum length of 14 (up to 70%), get the matrix of the train and test set and the weights
for i in range(1,14):

    Xtrain = getPolynomialDataMatrix(x_train, i)
    Xtest = getPolynomialDataMatrix(x_test, i)

    w = getWeightsForPolynomialFit(x_train, y_train, i)

    #calculates the sum of squared errors for the train and test set by taking the mean of each training and test matrices subtracted from them multiplied by the weight, squared.
    SSEtrain[i - 1] = np.mean((Xtrain.dot(w) - y_train)**2)
    SSEtest[i - 1] = np.mean((Xtest.dot(w) - y_test)**2)

#Plots the graph
plt.figure();
#Due to the huge differences in the error, uses a log-scale plot for the y-axis.
plt.semilogy(range(len(SSEtrain)), SSEtrain)
plt.semilogy(range(len(SSEtest)), SSEtest)
plt.legend(('SSE on training set', 'SSE on test set'))
plt.show()
```



These graphs represent the Sum of Squared Errors on both the Training Set and on the Test Set after it has been run 4 different times. It changes each time because the data is shuffled before it is split into the training and test set, so each one will be different each time. The general patterns show that the difference between the errors converges at the start, with low degrees. Both the first and the last graph meet/overlap at around weight 3. The second graph does not meet until around weight 5 but the lines stay very close together between degrees 2 and 4. The third graph looks fairly different to the others and crosses over at one small point between 5 and 6, the lines also diverge between 1 and 4 but this may be fairly anomalous. Based on these 4 samples alone, it is still hard to tell whether a weight of 3 or 5 would give more accurate predictions and more samples may need to be generated to estimate this fairly. However, it seems like 3 is accurate most od the time since the squared errors of the test and train set overlap around this point in 3 out of 4 graphs.

Task 2

2.1 Description of K-Means Clustering

K-Means Clustering is a simple yet powerful algorithm in Data Science and is a popular unsupervised Machine Learning algorithm. Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes. (Garbade, 2018) K-Means Clustering is used when the data is unlabelled, without categories or groups, and the goal of the algorithm is to find groups of data - K represents the number of groups. It can discover underlying patterns in data by looking for a fixed number of clusters in a dataset. A cluster is a collection of data points collected together because of some similarities.

The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. The results produced by K-Means are the centroids of the K clusters, which can be used to label new data, and labels for the training data as each data point is assigned to a single cluster. (Trevino, 2016) A centroid is the imaginary or real location representing the centre of the

cluster. The K-Means algorithm identifies k number of centroids and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible. (Garbade, 2018) So, each centroid of a cluster is a collection of feature values which define the resulting groups and examining these centroid feature weights can be used to qualitatively interpret what kind of group each cluster represents. (Trevino, 2016) The algorithm is called K-means because it refers to the averaging of the data, by finding the k number of centroids. To process the learning data, the K-means algorithm starts with a first group of randomly selected centroids, which are used as the starting points for every cluster, and then performs iterative calculations to optimize the positions of the centroids. It halts creating and optimizing clusters when either the centroids have stabilized — there is no change in their values because the clustering has been successful, or the defined number of iterations has been achieved. (Garbade, 2018)

The Euclidean Distance between two points in either a plane or 3-dimensional space measures the length of a segment connecting the two points and is considered the most obvious way of representing distance between two points. (Rosalind, n.d.) Euclidean Distance computes the root of square difference between co-ordinates of pair of objects using the formula:

$$Dist_{XY} = \sqrt{\sum_{k=1}^{m} (X_{ik} - X_{jk})^2}$$

Figure 4: Euclidean Distance Formula (Singh, Yadav and Rana, 2013)

The k-means clustering algorithm uses the Euclidean Distance to measure the similarities between objects. When it was compared to two other distance metrics, Manhattan Distance and Chebyshev Distance, it was found that Euclidean Distance measures can unequally weight underlying factors but the distortion in K-Means using Manhattan distance metric is less than that of k-means using Euclidean distance metric. As a conclusion, the K-means, which is implemented using Euclidean distance metric gives best result and K-means based on Manhattan distance metric's performance, is worst. (Singh, Yadav and Rana, 2013) When used in the K-Means Algorithm, Euclidean Distance between an observation and initial cluster centroids is calculated. Based on Euclidean Distance each observation is assigned to one of the clusters - based on minimum distance. It is important to note that the scale of measurements influences Euclidean Distance, so variable standardisation becomes necessary. (DnI Institute, 2015) The metric is used in the algorithm like so:

Euclidean Distance = $\sqrt{(X_H - H_1)^2 + (X_W - W_1)^2}$

Where

X_H: Observation value of variable Height

H_1: Centroid value of Cluster 1 for variable Height

X_W: Observation Value of variable Weight

W_1: Centroid value of cluster 1 for variable Weight

Figure 5: Euclidean Distance in K-Means (DnI Institute, 2015)

Given an initial set of K-means, the algorithm proceeds by alternating between two steps; assignment and update. In the assignment step, each data point is assigned to the nearest mean. Each observation is assigned to the cluster whose mean has the least squared Euclidean Distance which is intuitively the closest mean. In the update step, the model parameters which are the means are adjusted to match the sample means of the data points that they are responsible for. (MacKay, 2003) The new centroids of the observations are calculated in the new clusters. The algorithm has converged when the assignments no longer change but does not guarantee to find the optimum.

An advantage of K-means is that it is relatively simple to implement, the math can be coded, and Python offers useful libraries such as Pandas, NumPy, and Math. Furthermore, it is adaptable since it scales to large data sets, adapts to new examples, and generalizes to clusters of different shapes and sizes. (Google Developers, n.d.) When compared with hierarchical clustering, K-means is usually computationally faster if variables are large, but k is small. K-Means also produces tighter clusters than hierarchical clustering, especially if the clusters are globular. (Playwidtech, 2013) However, K-means clustering also has some drawbacks. One of which is that K must be chosen manually, and it is dependent on initial values. For a low K, this dependence can be mitigated by running k-means several times with different initial values and picking the best result. As K increases, advanced versions of k-means are necessary to pick better values of the initial centroids (called k-means seeding). (Google Developers, n.d.) Additionally, although K-means is good in capturing the structure of data if clusters have a spherical-like shape as it will always try to construct a spherical shape around the centroid. Because of this, when the clusters have a complicated geometric shapes, K-means does a poor job in clustering the data. K-means algorithm doesn't let data points that are far-away from each other share the same cluster even if they obviously belong to the same cluster. (Dabbura, 2018)

2.2 Implementation of K-Means Clustering

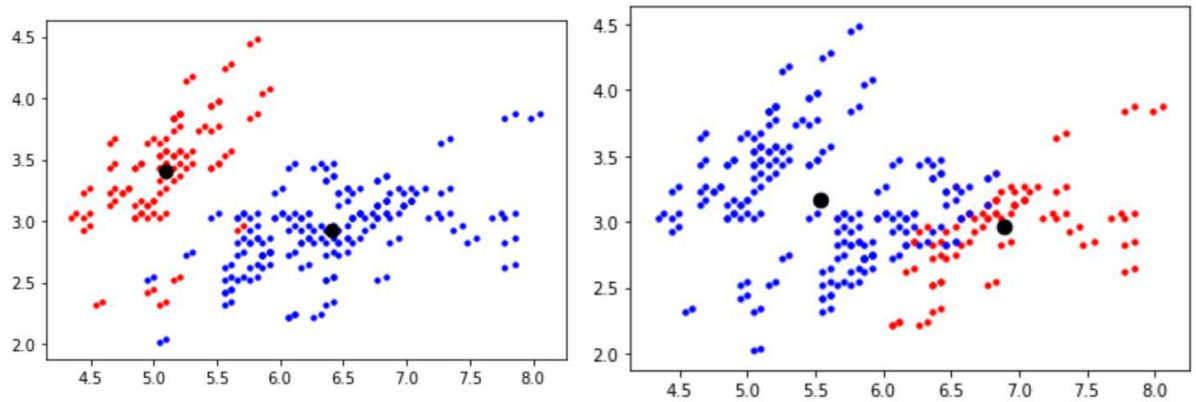
```
#CMP3751M Machine Learning Assignment 1
#Gabriella Di Gregorio 15624188
#Task 2.2

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math

#Reads in the data using pandas.read_csv because pandas.DataFrame.from_csv is outdated and discouraged.
dataset = pd.read_csv('CMP3751M_CMP9772M_ML_Assignment 1_Task2 - dataset - dog breeds.csv')


#Function to calculate Euclidean Distance, taking two features
#Renamed from compute_euclidean_distance(vec_1, vec_2)
def Euclidean_distance(feat_one, feat_two):

    #Initialises the Squared Distance to 0
    squared_distance = 0

    #Goes through each value in the first feature, and assumes the second is the same length
    for i in range(len(feat_one)):

            #for each value in the first feature, subtract the value of the second feature from the first, and square the answer
            #each result is added as a running total to the Squared Distance
            squared_distance += (feat_one[i] - feat_two[i])**2

    #To calculate the Euclidean Distance, Square Root the Squared Distance (uses the math library)
    ed = math.sqrt(squared_distance)

    return ed;
```

```
#A function to randomly choose values for k amount of centroids, using the values from the dataset
def initialise_centroids(dataset, k):
    #Uses the built-in numpy random to randomly choose centroid values between the lowest and highest value in the dataset,
    #for k amount of centroids, across 4 features/columns.
    return np.random.uniform(dataset.min(), dataset.max(), size=(k,4))

#numpy.random.uniform: Draw samples from a uniform distribution.
#Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high).
#Any value within the given interval is equally likely to be drawn by uniform. (Docs.scipy.org, 2018)


#Function to calculate the Kmeans, taking the data and k amount of centroids
def kmeans(dataset, k):
    #initialises centroids by calling the function to randomly assign them
    centroids = initialise_centroids(dataset, k)

    #initialises an empty array to store which cluster each piece of data is assigned to
    cluster_assigned = []

    #Creates an array with the same size as the centroids that can be used to detect whether the values have changed
    #numpy.empty_like: Return a new array with the same shape and type as a given array. (Docs.scipy.org, 2019)
    change = np.empty_like(centroids)


    while True:
        #Checks to see if the values of the centroids have changed, and stops the operation if not because this would mean a division by 0
        #numpy.allclose: Returns True if two arrays are element-wise equal within a tolerance. (Docs.scipy.org, 2019)
        if np.allclose(change, centroids) == True:
            break

        #assigns the values of the centroids to the change array so they are remembered and can be checked
        change = centroids

        #goes through each value in each row of the data
        for index, rows in dataset.iterrows():

            #initialises an empty array to store the Euclidean distances between each datapoint and the centroids
            distances = []

            #for the number of centroids (k), add the result of the calculation of euclidean distance function between the values in the rows and the centroids
            for i in range(k):
                distances.append(Euclidean_distance(rows, centroids[i]))

            #The following if statements are used to calculate the smallest distance value, depending on how many centroids there are.
            #The smallest of each is appended to the cluster assigned.
            #If there are only two centroids, then only two distance values need to be compared
            if k == 2:
                if distances[0] < distances[1]:
                    cluster_assigned.append(0)
                else:
                    cluster_assigned.append(1)
            #however, if there are three centroids, three values must be compared
            #if the first distance result is smaller than both the second and third, that can be appended,
            #if not, the second and third must be compared then the smallest of the two can be assigned to the cluster
            if k == 3:
                if distances[0] < distances[1] and distances[2]:
                    cluster_assigned.append(0)
                elif distances[1] < distances[2]:
                    cluster_assigned.append(1)
                else:
                    cluster_assigned.append(2)

        #initialises an empty array to store the new centroids, after the means have been calculated
        newcentroids = []
```

```
        #for all centroids, initialise the four features and a counter to 0
        for i in range(k):
            height, tail_length, leg_length, nose_circ, ctr = 0,0,0,0,0
            #For each row in the dataset, add up all of the first values in each row to the height (because this is a column),
            #and all of the second values in each row to the tail_length, and so on
            for index, rows in dataset.iterrows():
                if cluster_assigned[index] == i:
                    height+= rows[0]
                    tail_length+= rows[1]
                    leg_length+= rows[2]
                    nose_circ+= rows[3]
                    ctr+=1

            #if the counter is 0, use the old centroids to avoid a divide by 0 error
            if ctr == 0:
                newcentroids.append(centroids[i])
            else:
                #otherwise, append the means of each columns to the new centroid values
                newcentroids.append([height/ctr, tail_length/ctr, leg_length/ctr, nose_circ/ctr])
        centroids = newcentroids

    return centroids, cluster_assigned
```

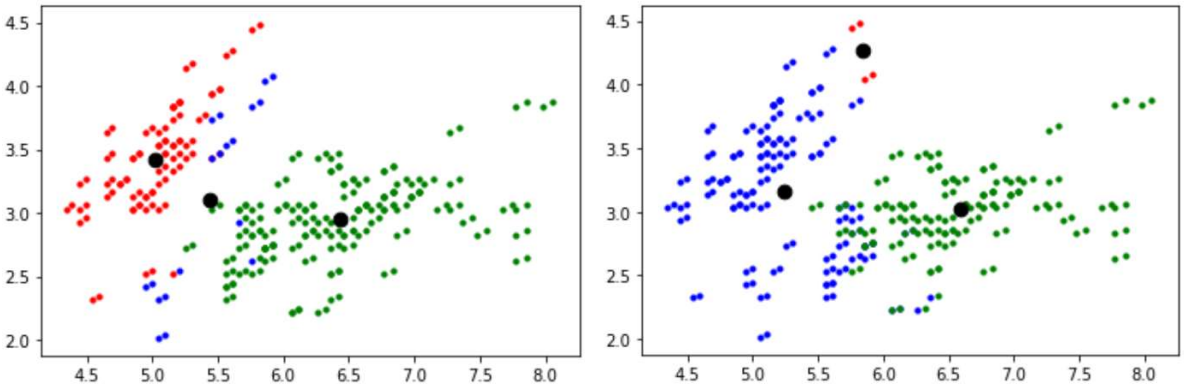
```
#K is the number of centroids, k = 2 is for two centroids, k = 3 is for three centroids
k = 2

#assigns the results of the kmeans function to 'cen' for centroids and 'clust' for cluster assigned
cen, clust = kmeans(dataset, k)
#makes each cluster represented by a different colour, red, blue, and green, so they are clear on a graph
colour = ['r', 'b', 'g']

#for each centroid, go through each row and plot the first [0] and second [1] (or third [2] and fourth [3]) with a size/scale of 10 and the assigned colour
for i in range(k):
    for index, row in dataset.iterrows():
        if clust[index] == i:
            plt.scatter(row[0],row[1], s=10, c=colour[i])

#Makes the centroids a dataframe
#pandas.DataFrame: Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).
#arithmetic operations align on both row and column labels. (Pandas.pydata.org, n.d.)
cen = pd.DataFrame(cen)
#plot the centroid points on a scattergraph, using a size/scale of 75 and the colour black.
plt.scatter(cen.values[:,0], cen.values[:,1], s=75, c='black')
#[:,0] for example means [first_row:last_row,column_0].
#For a 2-dimensional list/matrix/array, this notation gives all the values in column 0 (from all rows). (Stack Overflow, 2016)
#Displays the graph
plt.show()
```

Comparing height and tail length with 2 centroids:



These graph shows that there is no correlation between a dog's height and tail length, but they seem to be separated into distinct groups, dogs that are quite short but have long tails, and dogs that are quite tall but have generally shorter tails.

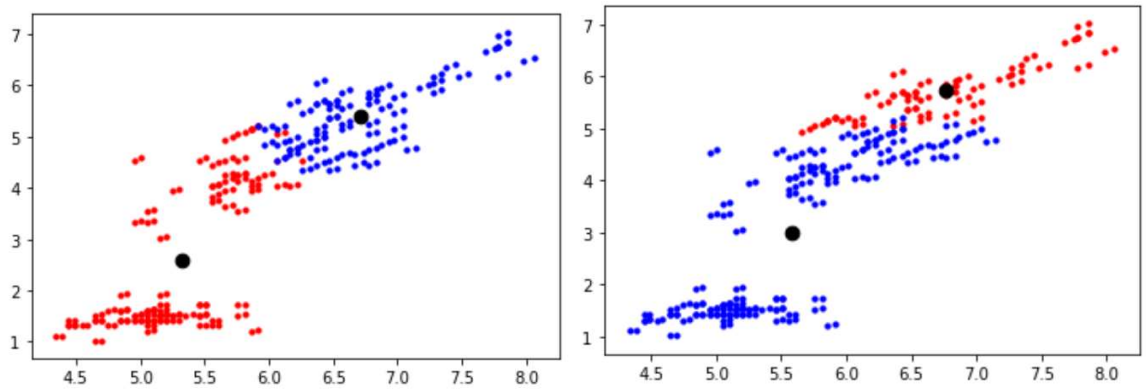
Comparing height and tail length with 3 centroids:



These graphs show that there are only 2 distinct groups as shown above, since there is often very few points clustered around 1 of the 3 centroids.

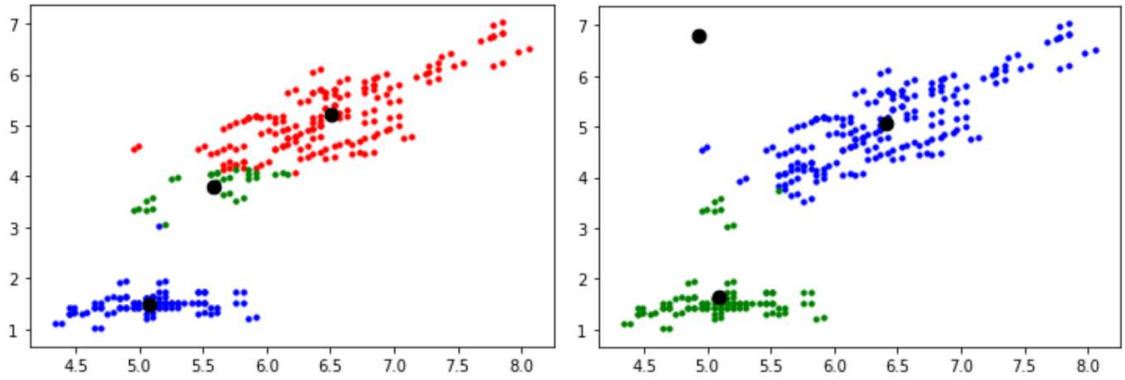
Comparing height and leg length with 2 centroids:

```
#for each centroid, go through each row and plot the first [0] and second [1] (or third [2] and fourth [3]) with a size/scale of 10 and the assigned colour
for i in range(k):
    for index, row in dataset.iterrows():
        if clust[index] == i:
            plt.scatter(row[0],row[2], s=10, c=colour[i])

#Makes the centroids a dataframe
#pandas.DataFrame: Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).
#arithmetic operations align on both row and column labels. (Pandas.pydata.org, n.d.)
cen = pd.DataFrame(cen)
#plot the centroid points on a scattergraph, using a size/scale of 75 and the colour black.
plt.scatter(cen.values[:,0], cen.values[:,2], s=75, c='black')
#[:,0] for example means [first_row:last_row,column_0].
#For a 2-dimensional list/matrix/array, this notation gives all the values in column 0 (from all rows). (Stack Overflow, 2016)
#Displays the graph
plt.show()
```



These graphs show that there is a positive correlation between dogs' heights and leg lengths, taller dogs generally have longer legs. However, there is a fairly odd pattern in the bottom left, as at the start, as the dogs get taller, the leg lengths stay short for some time. This could represent breeds that are fairly long but have short legs such as sausage dogs.

Comparing height and leg length with 3 centroids:



These graphs better show the group of short dogs with varying leg lengths. However, there seems to be no real need for 3 centroids on this data either since there is often no data to cluster around 1 out of 3 points. There is a gap in the data for leg lengths between 2 and 3 which could cause these seemingly 2 separate groups.

References

Statistics How To. (2016). *Absolute Error & Mean Absolute Error (MAE)*. [online] Available at: https://www.statisticshowto.datasciencecentral.com/absolute-error/ [Accessed 11 Nov. 2019].

Brownlee, J. (2019). *Overfitting and Underfitting With Machine Learning Algorithms*. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/overfitting-and-underfitting-with-machine-learning-algorithms/ [Accessed 27 Nov. 2019].

Dabbura, I. (2018). *K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks*. [online] Towards Data Science | Medium. Available at: https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a [Accessed 19 Nov. 2019].

Garbade, D. (2018). *Understanding K-means Clustering in Machine Learning*. [online] Towards Data Science. Available at: https://towardsdatascience.com/understanding-k-means-clustering-in-machine-learning-6a6e67336aa1 [Accessed 12 Nov. 2019].

Rosalind. (n.d.). *Glossary | Euclidean distance*. [online] Available at: http://rosalind.info/glossary/euclidean-distance/ [Accessed 12 Nov. 2019].

Grover, P. (2018). *5 Regression Loss Functions All Machine Learners Should Know*. [online] Medium. Available at: https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0 [Accessed 11 Nov. 2019].

DnI Institute. (2015). *K Means Clustering Algorithm: Explained*. [online] Available at: http://dni-institute.in/blogs/k-means-clustering-algorithm-explained/#:~:targetText=Euclidean%20is%20one%20of%20the,clusters%20%2D%20based%20on%20minimum%20distance [Accessed 12 Nov. 2019].

Google Developers. (n.d.). *k-Means Advantages and Disadvantages | Clustering in Machine Learning*. [online] Available at: https://developers.google.com/machine-learning/clustering/algorithm/advantages-disadvantages [Accessed 19 Nov. 2019].

Playwidtech. (2013). *K-Means Clustering Advantages and Disadvantages*. [online] Available at: http://playwidtech.blogspot.com/2013/02/k-means-clustering-advantages-and.html [Accessed 19 Nov. 2019].

Statistics How To. (2014). *Least Squares Regression Line: Ordinary and Partial*. [online] Available at: https://www.statisticshowto.datasciencecentral.com/least-squares-regression-line/ [Accessed 11 Nov. 2019].

Stat.yale.edu. (n.d.). *Linear Regression*. [online] Available at: http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm [Accessed 11 Nov. 2019].

MacKay, D. (2003). *Information theory, inference, and learning algorithms*. Cambridge: Cambridge University Press, pp.284-292.

Docs.scipy.org. (2019). *NumPy v1.17 Manual*. [online] Available at: https://docs.scipy.org/doc/numpy/index.html [Accessed 21 Nov. 2019].

Pant, A. (2019). *Introduction to Linear Regression and Polynomial Regression*. [online] Towards Data Science. Available at: https://towardsdatascience.com/introduction-to-linear-regression-and-polynomial-regression-f8adc96f31cb [Accessed 11 Nov. 2019].

Stack Overflow. (2016). *plt.plot meaning of [:,0] and [:,1]*. [online] Available at: https://stackoverflow.com/questions/40557910/plt-plot-meaning-of-0-and-1/40558730 [Accessed 21 Nov. 2019].

Seif, G. (2019). *Understanding the 3 most common loss functions for Machine Learning Regression*. [online] Towards Data Science. Available at: https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3 [Accessed 11 Nov. 2019].

Singh, A., Yadav, A. and Rana, A. (2013). K-means with Three different Distance Metrics. *International Journal of Computer Applications*, 67(10), pp.13-17.

Trevino, A. (2016). *Introduction to K-means Clustering*. [online] Oracle. Available at: https://blogs.oracle.com/datascience/introduction-to-k-means-clustering [Accessed 12 Nov. 2019].