Task 1

Ridge Regression Description:

Linear regression models –
A Linear regression model is used to predict values based on features of pre-existing data, to do this a line of best fit is calculated in the form of the equation "$Y \approx \beta 0 + \beta 1X$" (James et al, 2013, 61).

MSE and SSE –
Both SSE and MSE are attempts to evaluate how well a model created by a regression function fits the data and by extension how useful it is in predicting data points.
SSE stands for Sum of Squared Errors which is the sum of differences between the line of best fit created by and the data points, the data is squared so that if data points are below the regression model, causing negative error numbers, do not make the line look better by decreasing the overall sum.
MSE stands for Mean Squared Error which is the SSE divided by the number of data points used. This gives the average error.

Least squares solution –
To find the optimal result for regressions, the SSE or RMSE must be reduced as much as possible, giving the least square solution.

Use of features in linear regression –
When using data the dependant variables may be affected by more than one independent variable, to make use of this, each independent variable is given weight calculated from the dependant variable and will then be used for prediction

Overfitting and underfitting -
When finding the best fit for the data the aim is to reduce error but there is a point in which the fit is the best it can be. While trying to attain this best fit line underfitting usually occurs, where there is too much error as the line doesn't fit the data. After finding best fit any increased reduction in error will cause overfitting, where the line tries to fit the data so specifically that it is unusable for making predictions.

Intuition of the weight penalty term –
The penalty term/s is used for ridge regression in an attempt to penalise overfitting.

Objective function of ridge regression –
The objective function for ridge regression is the RMSE (Root Mean Squared Error), used to see how closely the functions predictions are to the actual values, this is used on data with all dependant variables already collected.

Ridge Regression Implementation:

When running the code initially, the main function is called. Firstly, the regularization factors are stored in an array and then the following functions are called:

Input, trainassess, ridge_regression, rid_reg_plot and lastly points_plot.

All the functions are discussed below, Input and trainassess are only called once while the rest are called equal to the amount of regularization factors.

```
reg_fact_train = np.array([0.000001, 0.0001, 0.01, 0.1]) #10-6, 10-4, 10-2, 10-1
(d_tr, d_pl) = input('regression_train_assignment2019.csv', 'regression_plotting_assignment2019.csv')
i = 0
(tr_x, tr_y, tr_f, pl_x, pl_f) = trainassess(d_tr, d_pl)
#(x_tr, y_tr, tr_a) = trainassess(d_tr)
for regs in reg_fact_train:
    (par) = ridge_regression(tr_f, tr_y, regs)
    (pl_y) = rid_reg_plot(pl_f, par)
    points_plot(tr_x, tr_y, pl_x, pl_y, regs)
    i+=1



```

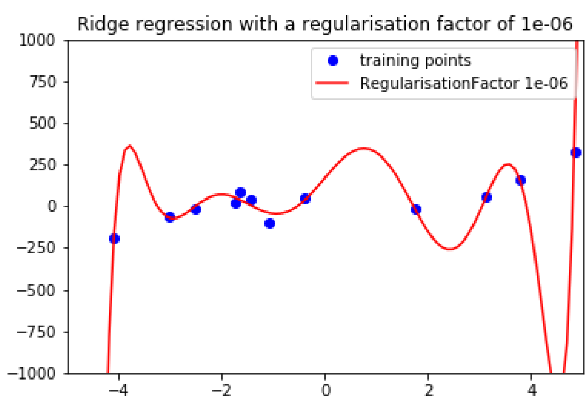
Both Input and trainassess are used for importing data, initially all training and plotting data are placed within Data Frames before being split up into their X values, Features and Y values for training.

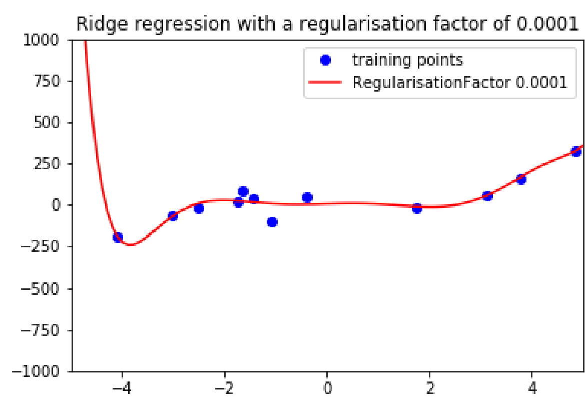
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

def input(train1, test1): #Takes csv files and imports them as dataframes
    #data_train = pd.DataFrame.from_csv(train1)
    #data_plot = pd.DataFrame.from_csv(test1)
    data_train = pd.read_csv(train1)
    data_plot = pd.read_csv(test1, index_col=None)
    return data_train, data_plot


def trainassess (d_train, d_plot): #Importing the x,y and features from training. Importing x and Features from plotting
    x_train = d_train['x'].values
    y_train = d_train['y'].values
    f_train = d_train.iloc[:, 3:15].values
    x_plot =  d_plot['x'].values
    f_plot = d_plot.iloc[:, 2:14].values
    #print(f_plot)
    return  x_train, y_train, f_train, x_plot, f_plot

```

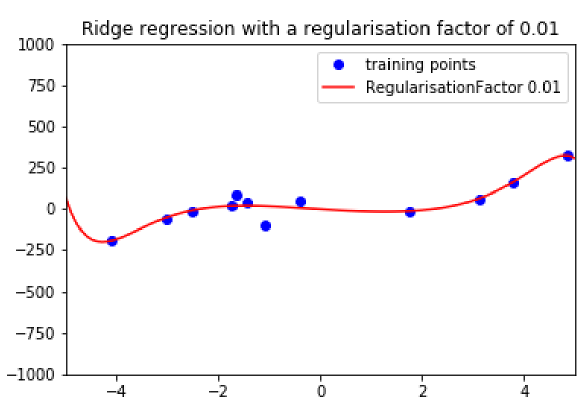
After this the loop starts, initially calling the Ridge Regression equation that calculates the predicted weights each feature must have to give an accurate Y value in this case, this value is changed dependant on the regularization factor. These weights are then used in the next function with the plotting (test) features to predict the y-values for the test data. Finally, the x and y values for both datasets are plotted within the same plot

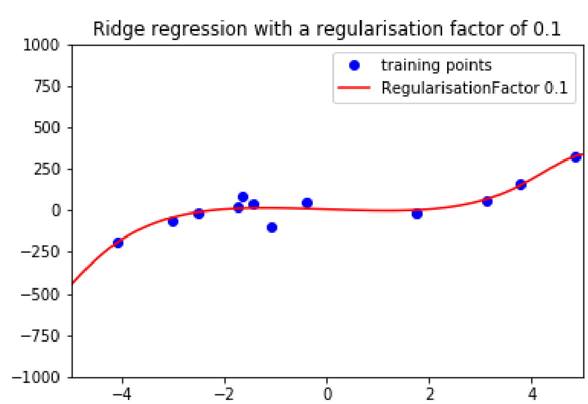
```python
def ridge_regression(features_train, y_train, regularisationFactor): #Completes Ridge Regression equation

    parameters = features_train.transpose().dot(features_train)
    parameters = parameters + (regularisationFactor * (np.identity(parameters.shape[1])))
    parameters2 = features_train.transpose().dot(y_train)
    parameters = np.linalg.solve(parameters, parameters2)
    return parameters

def rid_reg_plot(f_plot, parameters): #Calculate the predicted Y values by dot producting features with weights
   i = 0
   y_calc_plot = np.zeros(len(f_plot))
   while i < len(f_plot):
       curr_feat = f_plot[i, :]
       y_calc_plot[i] = curr_feat.dot(parameters)
       i+=1
   return  y_calc_plot

def points_plot(x_train, y_train, x_plot, y_calc_plot, regfact): #Plotting of the training and plot data
    plt.figure()
    #plt.clf()
    plt.plot(x_train, y_train, 'bo')
    plt.plot(x_plot, y_calc_plot, 'r')
    #plt.plot(train[0],train[0], 'go')
    plt.legend(('training points', 'RegularisationFactor ' + str(regfact)))
    plt.title('Ridge regression with a regularisation factor of ' + str(regfact))
    plt.xlim((-5, 5))
    plt.ylim((-1000, 1000))
    #plt.hold(True)
    plt.savefig('RidgeRegPlot' + str(regfact)+ '.png')
    plt.show()

```

The Plotting above gives the following data for each iteration of the regularization loop:









<u>Ridge Regression Evaluation:</u>

To evaluate the ridge regression a larger list of regularization factors is used, this time the functions called are:

Input, randomtrain, traineval, ridge_regression, eval_regression and lastly plotting_rmse.

Input, randomtrain, traineval and plotting_rmse are called once while the rest are called equal to the amount of regularization factors.

```
reg_fact_eval = np.array([0.000001, 0.0001, 0.01, 0.1, 1, 10, 100, 1000])
(d_tr, d_pl) = input('regression_train_assignment2019.csv', 'regression_plotting_assignment2019.csv')
train_rmse = np.zeros(len(reg_fact_eval))
test_rmse = np.zeros(len(reg_fact_eval))
(d_tr_rand, d_pl_rand) = randomtrain(d_tr)
(tr_x, tr_y, tr_f, pl_x, pl_y, pl_f) = traineval(d_tr_rand, d_pl_rand)

j = 0
for regs in reg_fact_eval:
    (par) = ridge_regression(tr_f, tr_y, regs)
    (train_rmse[j]) = eval_regression(par, tr_f, tr_y)
    (test_rmse[j]) = eval_regression(par, pl_f, pl_y)
    j+=1

plotting_rmse(reg_fact_eval, train_rmse, reg_fact_eval, test_rmse)
```

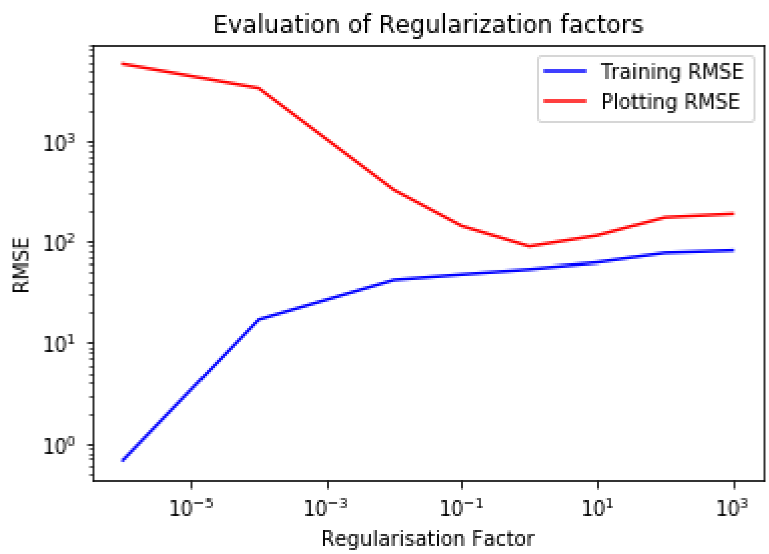
Similar to in the previous implementation, both files are imported but this time only the training data is used, because of its Y values. The data is firstly randomised every iteration and then split in a 70:30 split within the randomtrain function before being outputted back to the main function before being split up within the traineval function, taking x, y and features

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

def input(train1, test1): #Takes csv files and imports them as dataframes
    #data_train = pd.DataFrame.from_csv(train1)
    #data_plot = pd.DataFrame.from_csv(test1)
    data_train = pd.read_csv(train1)
    data_plot = pd.read_csv(test1, index_col=None)
    return data_train, data_plot

def randomtrain(input_train): #Randomises which part of the train file will be used for training or plotting
    length = len(input_train)
    #np.random.shuffle(input_train[0])
    output = input_train.sample(frac=1)
    seventy = int(round(0.7*(length)))
    #print(seventy)
    output_train = output[0:seventy]
    output_plot = output[seventy:length]
    return output_train, output_plot

def traineval(d_tr, d_pl): #Importing the correct data (features) from the train file specifically
    x_train = d_tr['x'].values
    y_train = d_tr['y'].values
    f_train = d_tr.iloc[:, 3:15].values
    x_plot =  d_pl['x'].values
    y_plot = d_pl['y'].values
    f_plot = d_pl.iloc[:, 3:15].values
    return  x_train, y_train, f_train, x_plot, y_plot, f_plot

```

The ridge_regression function is the exact same as during implementation, the new function eval_regression is called twice both for the training and the test data. Firstly, the predicted y-value for the entire dataset is calculated. Then, using this new array, the RMSE (Root Mean Square Error) is found.

```
def ridge_regression(features_train, y_train, regularisationFactor): #Computes the weights for the training features

    parameters = features_train.transpose().dot(features_train)
    parameters = parameters + (regularisationFactor * (np.identity(parameters.shape[1])))
    parameters2 = features_train.transpose().dot(y_train)
    parameters = np.linalg.solve(parameters, parameters2)
    return parameters

def eval_regression(parameters, features, y): #Calculates the y hat value and then compares it to y to calculate the rmse
    i = 0
    y_comp = np.zeros(len(features))
    while i < len(features):
        curr_feat = features[i, :]
        y_comp[i] = curr_feat.dot(parameters)
        i+=1
    sq_er = np.square(y - y_comp)
    mean_sq_er = np.mean(sq_er)
    rmse =  np.sqrt(mean_sq_er)
    return rmse

def plotting_rmse(reg_fact_train, train_rmse, reg_fact_eval, test_rmse): #Plots RMSE of both training and testing data
    plt.figure()
    plt.loglog(reg_fact_train, train_rmse, 'b')
    plt.loglog(reg_fact_eval, test_rmse, 'r')
    plt.title('Evaluation of Regularization factors')
    plt.legend(('Training RMSE', 'Plotting RMSE'))
    plt.xlabel('Regularisation Factor')
    plt.ylabel('RMSE')
    plt.savefig('RMSEdata.png')
    plt.show()

```

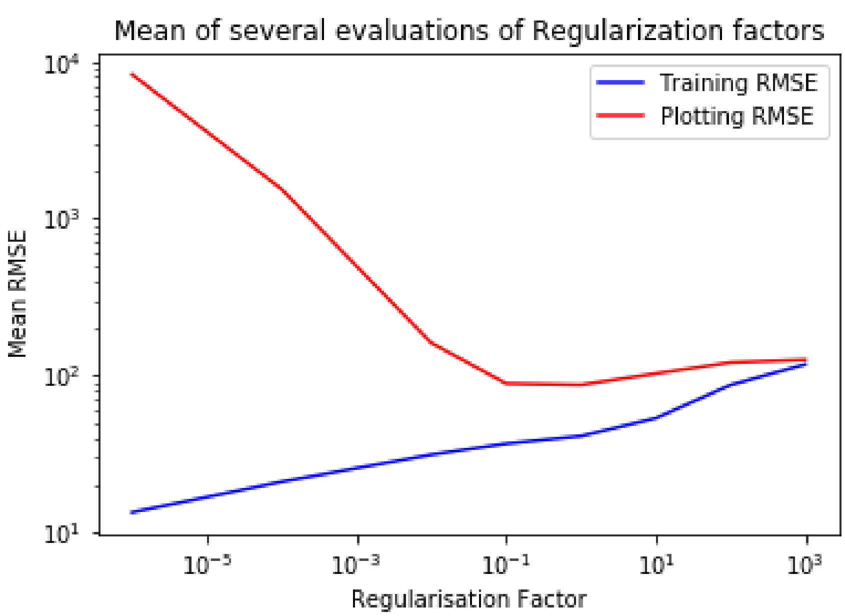
The plotted RMSEs are shown below:



The RMSE's can change quite a lot each time and so a mean of several RMSEs gives a much more accurate display, the only extra code added is the ten loops (rmse_num) that are iterated and the mean values calculated

```

reg_fact_eval = np.array([0.000001, 0.0001, 0.01, 0.1, 1, 10, 100, 1000])
rmse_num = 10
train_rmse = np.zeros(len(reg_fact_eval))
test_rmse = np.zeros(len(reg_fact_eval))
#tr_rmse_mean = np.zeros((rmse_num, len(reg_fact_eval)))
tr_rmse_mean = np.zeros(len(reg_fact_eval))
#te_rmse_mean = np.zeros((rmse_num, len(reg_fact_eval)))
te_rmse_mean = np.zeros(len(reg_fact_eval))
i=0
while i < rmse_num:
    (d_tr, d_pl) = input('regression_train_assignment2019.csv', 'regression_plotting_assignment2019.csv')
    (d_tr, d_pl) = randomtrain(d_tr)
    (tr_x, tr_y, tr_f, pl_x, pl_y, pl_f) = traineval(d_tr, d_pl)
    j = 0
    for regs in reg_fact_eval:
        (par) = ridge_regression(tr_f, tr_y, regs)
        (train_rmse[j]) = eval_regression(par, tr_f, tr_y)
        (test_rmse[j]) = eval_regression(par, pl_f, pl_y)
        tr_rmse_mean[j] += train_rmse[j]
        te_rmse_mean[j] += test_rmse[j]
        j+=1
    i+=1
tr_mean = tr_rmse_mean / rmse_num
te_mean = te_rmse_mean / rmse_num
#tr_mean = np.mean
plotting_rmse(reg_fact_eval, tr_mean, reg_fact_eval, te_mean)
```

The plotting shows that given the data 10 to the power of -1 is the best Regularization factor:



Task 2

K-Means Clustering Description:

Centroids –

Centroids are the cluster centre points in which data can be assigned and so the number of centroids is equal to the number of clusters needed.

Euclidean distance –

This is the distance between centroids and each data point, taking the square distance of all features from the centroids. The square is taken to make sure that any distances that are lower than the centroid in any dimension do not reduce the Euclidean distance unnecessarily. This value is also used to calculate errors.

Assignment step –

The data points are all assigned to the closest centroid, calculated via the Euclidean distance.

Update step –

Once all data points have been assigned to their closest centroids the mean of the of their values are taken, this will be the new centroid, iteration follows until the new centroids have not changed from the previous and a convergence is reached, the clusters are then which data points are closest to each of the final centroids.

Objective function –

The objective function is the value we are attempting to reduce and use to evaluate how well the clustering fits the data. For this the RSS for each iteration of values of k can show that the errors is decreasing and the clustering is not getting further away, if the objective is to find out how many clusters are necessary and by extension what the best value of k is, then the final RSS can be found for each k value and compared. If as the number of clusters increases the aggregate distance decreases steeply then at the point the gradient becomes shallow or where the “mountain ends” and the “rubbles’ begins” (Zhong, M. 2019)

K-Means Clustering Implementation:

The code loops for the two k-values, each time importing the csv file, this first set of code was used to ensure no issues occurred when using more than two features so stem_data only stores stem_length and stem_diameter. Later code will use all features. The kmeans function is then called which will itself call several functions, the only other functions are for plotting both the clusters in a two dimensional space and to show the objective.

```
k = [3, 4]
i = 0
while i < len(k):
    #data_train = np.loadtxt(open("CMP3744M_ADM_Assignment 1_Task2 - dataset - plants.csv", "rb"), delimiter=",")
    Cluster_data = pd.read_csv('CMP3744M_ADM_Assignment 1_Task2 - dataset - plants.csv', index_col=None)
    stem_data = Cluster_data.iloc[:,0:2].values

    (cent, clu_as, iterat, obj) = kmeans(stem_data, k[i])
    #print(clu_as)
    #print(cent)
    x_1 = Cluster_data.iloc[:,0].values
    y_1 = Cluster_data.iloc[:,1].values
    x_2 = Cluster_data.iloc[:,2].values
    y_2 = Cluster_data.iloc[:,3].values
    #x_train = data_train[:,0]
    #y_train = data_train[:,1]
    iteration = np.array(range(1,(iterat+1)))
    #print(iteration)
    #k = KMeans(n_clusters=3).fit_predict(stem_data)
    #plt.scatter(x_train1, x_train2, c=k)
    centx = cent[:,0]
    centy = cent[:,1]
    #print(stem_data[:,0].shape)
    plot_clustering(x_1,y_1, clu_as, centx, centy, k[i])
    plot_iter_obj(iteration, obj, k[i])
    i+=1
```

Within the kmeans function a number of values are set including which cluster the variables have been assigned to, a comparison cluster array, the total of the Euclidean distances for each data point and the sum of all Euclidean distances.
This function calls several functions within itself, including:

Initialise_centroids and compute_euclidean_distance

The initialise_centroids is called at the start and should only be called once if the centroids are allocated correctly, compute_euclidean_distance is called every time the distance from the centroids is needed, they will be discussed in more detail below.

```python
def kmeans(dataset, k): # k=3 or 4
    chk_centroids_zero = True
    #objective = 0.0
    while chk_centroids_zero == True:
        clust_assi = np.ones(dataset[:,0].size) #Current cluster assigned to the dataset
        clust_comp = np.zeros(dataset[:,0].size) #Previous cluster assigned to the dataset, used for comparison
        objective = np.array([])
        objective_sum = np.array([]) #Used to hold the objective function
        cluster_assigned = np.empty(dataset[:,0].size) #Once completed clusters will be assigned here
        euc_xy_tot = np.zeros([k, 2]) #The total euclidian distance for x and y
        euc_clus_tot = np.zeros(k)
        mean_tot = np.zeros([k, 2])
        loops = 0
        #while chk_centroids_zero == True:
        centroids = initialise_centroids(dataset, k)
        #while
        #print(centroids)
        while np.array_equal(clust_assi, clust_comp) != True:
            objective_sum = np.array([]) #Reset Calculations of Euclidean error each loop
            euc_xy_tot = np.zeros([k, 2])
            euc_clus_tot = np.zeros(k)
            #print(clust_assi, clust_comp)
            #clust_comp = np.zeros(dataset[:,0].size)
            clust_comp[:] = clust_assi[:]
            #print(clust_comp)
            i = 0
            while i < dataset[:,0].size : #Loops through all 300 values

                dist_data = dataset[i, :] # takes the ith x and y values
                #print(dist_data)
                #calculates the euc dist for each datapoint per centroid
                dist = compute_euclidean_distance(dist_data, centroids)
                #print (dist)
                mindist = np.argmin(dist)
                shortest_dist = dist[mindist]
                #print(shortest_dist)
                #print(mindist)
                objective_sum = np.append(objective_sum, shortest_dist)
                euc_xy_tot[mindist, 0] += dataset[i, 0]
                euc_xy_tot[mindist, 1] += dataset[i, 1]
                clust_assi[i] = mindist
                #print(clust_assi[i])
                euc_clus_tot[mindist] += 1
                i+=1
            objective = np.append(objective, np.sum(objective_sum))


            if 0 not in euc_clus_tot: #Used to make sure the centroids have data points within them
                chk_centroids_zero = False
                #print(objective)
            j=0
            while j < k :
                if euc_xy_tot[j, 0] != 0 and euc_xy_tot[j, 1] != 0:
                    mean_tot[j, 0 ] =  euc_xy_tot[j, 0] / euc_clus_tot[j]
                    mean_tot[j, 1 ] =  euc_xy_tot[j, 1] / euc_clus_tot[j]
                j+=1
            #print(mean_tot)
            centroids[:] = mean_tot[:]
            #print(centroids)
            #print(euc_xy_tot)
            #print(clust_assi, clust_comp)

            loops+=1
        #print(objective)
        cluster_assigned = clust_assi[:]
    return centroids, cluster_assigned, loops, objective
```

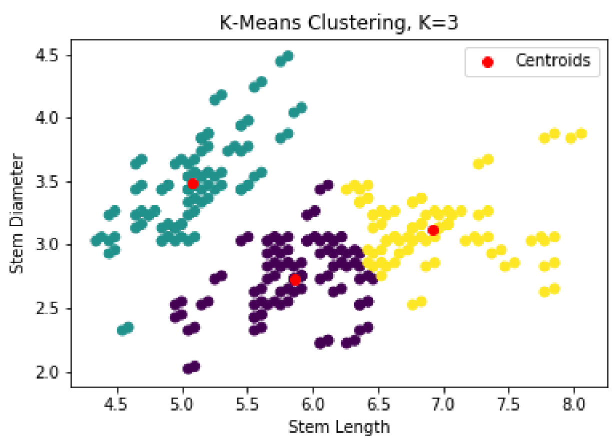
Within the initialise_centroids function random float values are chosen from between the upper and lower bounds of the dataset, this is to reduce the chance the centroids will be so far outside the dataset that no data points will be assigned to them, leading to the eventual centroid being at 0,0 on a two dimensional graph. The compute_euclidean_distance function is called for each data point and compares it to all centroids and their features.

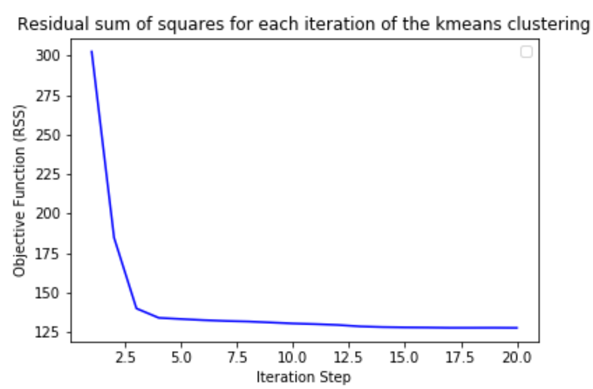
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random as rd


def initialise_centroids(dataset, k): #k=3 or 4, called once at start of code
    centroids = np.empty([k, 2]) #Create an empty array
    i = 0
    kmax = np.zeros([k])
    #dtmin = int(np.min(dataset))
    #dtmax = int(np.max(dataset))
    dtmin = np.min(dataset)
    dtmax = np.max(dataset)

    #dtminx = np.argmin(dataset[:,0])
    #dtminx2 = int(dataset[dtminx, 0])
    #dtmaxx = np.argmax(dataset[:,0])
    #dtmaxx2 =int(dataset[dtmaxx, 0])

    #dtminy = np.argmin(dataset[:,1])
    #dtminy2 = int(dataset[dtminy, 1])
    #dtmaxy = np.argmax(dataset[:,1])
    #dtmaxy2 = int(dataset[dtmaxy, 1])

    while i < k: #going through array until all centroids have a random x and y value
        #centroids[i,0] = rd.randint(dtmin,dtmax)
        #centroids[i,1] = rd.randint(dtmin,dtmax)
        centroids[i,0] = rd.uniform(dtmin,dtmax)
        centroids[i,1] = rd.uniform(dtmin,dtmax)
        i+=1
    return centroids


def compute_euclidean_distance(dtst, centr):
    distance = np.empty([centr[:,0].size])
    #print(test)
    i = 0
    while i < centr[:,0].size :
        #Euclidian distance = the sum of data point values - centroid values then squared
        #distance[i] = np.sqrt(np.square(dtst[0] - centr[i,0]) + np.square(dtst[1] - centr[i,1]))
        distance[i] = np.sqrt(np.square(dtst[0] - centr[i,0]) + np.square(dtst[1] - centr[i,1]))
        i+=1
    #distance = np.sum
    return distance

```

The plotting functions called after kmeans completes display the data below

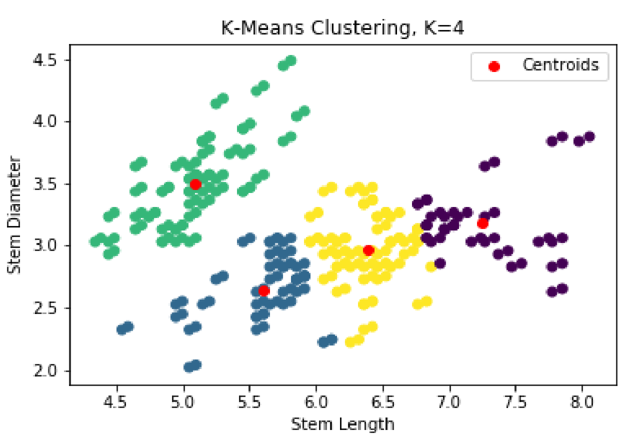
```python

def plot_clustering(x1,y1, clus, centx, centy, k):
    plt.figure()
    plt.scatter(x1, y1, c=clus)
    plt.scatter(centx, centy, c='r', label='Centroids')
    plt.legend()
    plt.title('K-Means Clustering, K=' + str(k))
    plt.xlabel('Stem Length')
    plt.ylabel('Stem Diameter')
    plt.savefig('TwoFeatureCluster' + str(k) + '.png')
    plt.show()

def plot_iter_obj(iterations, obj_func, k): # inputs iterations, the objective function (euclidean)
    plt.figure()
    plt.plot(iterations, obj_func, c='b')
    plt.legend()
    plt.title('Residual sum of squares for each iteration of the kmeans clustering')
    plt.xlabel('Iteration Step')
    plt.ylabel('Objective Function (RSS)') #Sum of the sum of the euclidian distances per iteration
    plt.savefig('TwoFeatureRSS' + str(k) + '.png')
    plt.show()

```

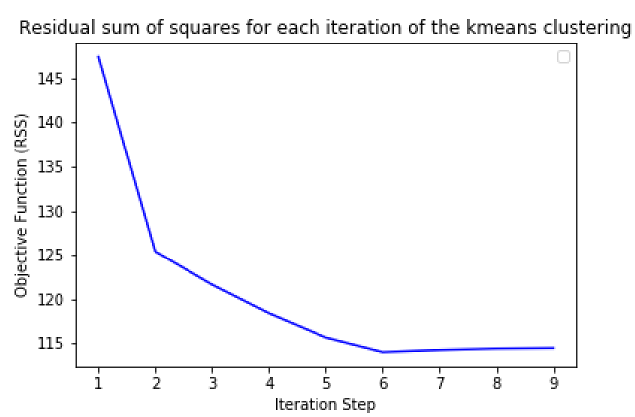
For K = 3 the plots show:





For K = 4 the plots show:





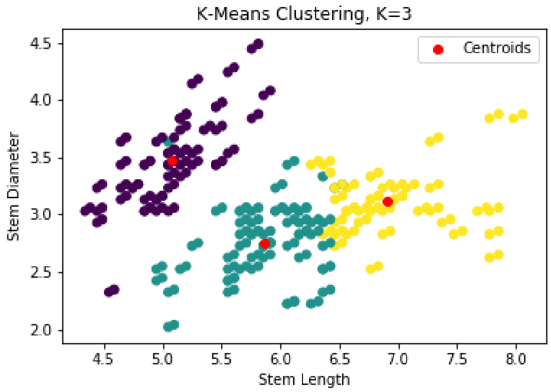
To allow for full use of the kmeans all four features must be used to group the data, small changes were made to all functions to be resilient to as many features as necessary, the entirety of the code will be shown below

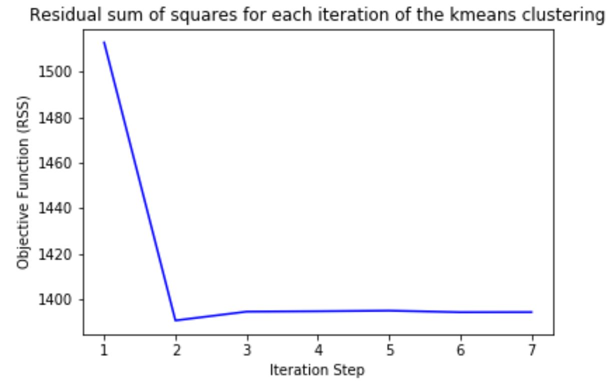
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random as rd


def initialise_centroids(dataset, k): #k=3 or 4, called once at start of code
    num_of_feat = dataset[0,:].size
    centroids = np.empty([k, num_of_feat]) #Create an empty array
    i = 0
    j = 0
    kmax = np.zeros([k])
    dtmin = np.min(dataset)
    dtmax = np.max(dataset)
    while j < num_of_feat:
        while i < k: #going through array until all centroids have a random x and y value
            centroids[i,j] = rd.uniform(dtmin,dtmax)
            i+=1
        i=0
        j+=1
    return centroids


def compute_euclidean_distance(dtst, centr):
    #print(dtst)
    num_of_feat = dtst.size
    distance = np.empty([centr[:,0].size])
    #print(test)
    i = 0
    j = 0
    while i < centr[:,0].size:
        while j < num_of_feat:
            #Euclidian distance = the sum of data point values - centroid values then squared
            #distance[i] = np.sqrt(np.square(dtst[0] - centr[i,0]) + np.square(dtst[1] - centr[i,1]))
            distance[i] += np.square(dtst[j] - centr[i,j])
            #print(distance[i])
            j+=1
        distance[i] = np.sqrt(distance[i])
        j=0
        i+=1
    #print(distance)
    return distance
```

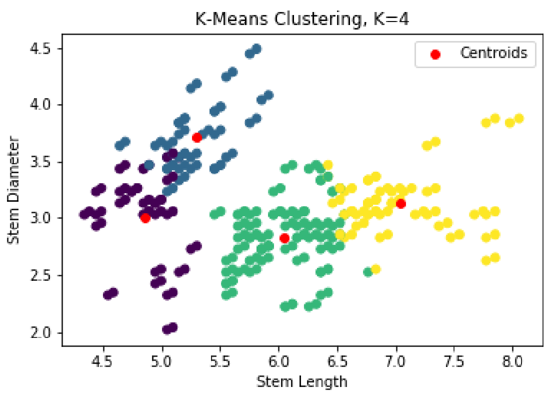
```
def kmeans(dataset, k): # k=3 or 4
    chk_centroids_zero = True
    num_of_feat = dataset[0,:].size
    #objective = 0.0
    while chk_centroids_zero == True:
        clust_assi = np.ones(dataset[:,0].size) #Current cluster assigned to the dataset
        clust_comp = np.zeros(dataset[:,0].size) #Previous cluster assigned to the dataset, used for comparison
        objective = np.array([]) #Used to hold the objective function
        objective_sum = np.array([]) #Used to hold the objective function
        cluster_assigned = np.empty(dataset[:,0].size) #Once completed clusters will be assigned here
        euc_xy_tot = np.zeros([k, num_of_feat]) #The total euclidian distance for all features
        euc_clus_tot = np.zeros(k)
        mean_tot = np.zeros([k, num_of_feat])
        loops = 0
        #while chk_centroids_zero == True:
        centroids = initialise_centroids(dataset, k)
        #while
        #print(centroids)
        while np.array_equal(clust_assi, clust_comp) != True:
            objective_sum = np.array([]) #Reset Calculations of Euclidean error each loop
            euc_xy_tot = np.zeros([k, num_of_feat])
            euc_clus_tot = np.zeros(k)
            #print(clust_assi, clust_comp)
            #clust_comp = np.zeros(dataset[:,0].size)
            clust_comp[:] = clust_assi[:]
            #print(clust_comp)
            i = 0
            while i < dataset[:,0].size : #loops through all 300 values

                dist_data = dataset[i, :] # takes the ith features
                #print(dist_data)
                dist = compute_euclidean_distance(dist_data, centroids) #calculates the euc dist for each datapoint per centr
                #print (dist)
                mindist = np.argmin(dist)
                shortest_dist = dist[mindist]
                #print(shortest_dist)
                #print(mindist)
                objective_sum = np.append(objective_sum, shortest_dist)
                euc_xy_tot[mindist, 0] += dataset[i, 0]
                euc_xy_tot[mindist, 1] += dataset[i, 1]
                clust_assi[i] = mindist
                #print(clust_assi[i])
                euc_clus_tot[mindist] += 1
                i+=1
            objective = np.append(objective, np.sum(objective_sum))
            #print(objective)

            if 0 not in euc_clus_tot: #Used to make sure the centroids have data points within them
                chk_centroids_zero = False
                #print(objective)
            j=0
            while j < k :
                if euc_xy_tot[j, 0] != 0 and euc_xy_tot[j, 1] != 0:
                    mean_tot[j, 0 ] =  euc_xy_tot[j, 0] / euc_clus_tot[j]
                    mean_tot[j, 1 ] =  euc_xy_tot[j, 1] / euc_clus_tot[j]
                j+=1
            #print(mean_tot)
            centroids[:] = mean_tot[:]
            #print(centroids)
            #print(euc_xy_tot)
            #print(clust_assi, clust_comp)

            loops+=1
        #print(objective)
        cluster_assigned = clust_assi[:]
    return centroids, cluster_assigned, loops, objective
```

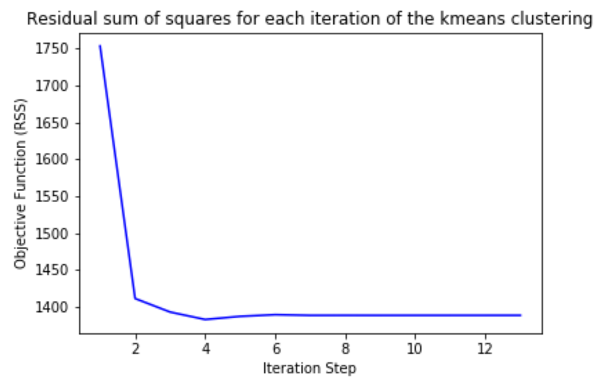
```python

def plot_clustering(x1,y1, clus, centx, centy, k):
    plt.figure()
    plt.scatter(x1, y1, c=clus)
    plt.scatter(centx, centy, c='r', label='Centroids')
    plt.legend()
    plt.title('K-Means Clustering, K=' + str(k))
    plt.xlabel('Stem Length')
    plt.ylabel('Stem Diameter')
    plt.savefig('FullCluster' + str(k) + '.png')
    plt.show()

def plot_iter_obj(iterations, obj_func, k): # inputs iterations, the objective function (euclidean)
    plt.figure()
    plt.plot(iterations, obj_func, c='b')
    #plt.legend()
    plt.title('Residual sum of squares for each iteration of the kmeans clustering')
    plt.xlabel('Iteration Step')
    plt.ylabel('Objective Function (RSS)') #Sum of the sum of the euclidian distances per iteration
    plt.savefig('FullRSS' + str(k) + '.png')
    plt.show()

k = [3, 4]
i = 0
while i < len(k):
    #data_train = np.loadtxt(open("CMP3744M_ADM_Assignment 1_Task2 - dataset - plants.csv", "rb"), delimiter=",")
    Cluster_data = pd.read_csv('CMP3744M_ADM_Assignment 1_Task2 - dataset - plants.csv', index_col=None)
    all_data = Cluster_data.iloc[:,0:4].values

    (cent, clu_as, iterat, obj) = kmeans(all_data, k[i])
    #print(clu_as)
    #print(cent)
    x_1 = Cluster_data.iloc[:,0].values
    y_1 = Cluster_data.iloc[:,1].values
    x_2 = Cluster_data.iloc[:,2].values
    y_2 = Cluster_data.iloc[:,3].values
    #x_train = data_train[:,0]
    #y_train = data_train[:,1]
    iteration = np.array(range(1,(iterat+1)))
    #print(iteration)
    #k = KMeans(n_clusters=3).fit_predict(stem_data)
    #plt.scatter(x_train1, x_train2, c=k)
    centx = cent[:,0]
    centy = cent[:,1]
    #print(stem_data[:,0].shape)
    plot_clustering(x_1,y_1, clu_as, centx, centy, k[i])
    plot_iter_obj(iteration, obj, k[i])
    i+=1
```

It produces slightly different results shown below, there is some overlap with different clusters as the features are create a space that is a higher dimension than that used to plot the graphs:









References:

Bishop, C.M. (2006) *Pattern recognition and machine learning*. Oxford: Springer

Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*. Cambridge, Massachusetts: The MIT Press

Han, J., Kamber, M. and Pei, J. (2012) *Data Mining: Concepts and Techniques*. Amsterdam: Morgan Kaufmann

James, G., Witten, D., Hastie, T. and Tibshirani, R. (2013) *An Introduction to Statistical Learning with Applications in R*. New York, United States: Springer

Zhong, M. (2019) *Clustering* [lecture]. Algorithms and Data Mining CMP3744M-1819, University of Lincoln, 21 February. Available from: https://blackboard.lincoln.ac.uk/bbcswebdav/pid-2201579-dt-content-rid-3878326_2/courses/CMP-ADM-1819/ADM_W5_Lec_Clustering_V2.pdf [accessed 17 March 2019]