Assignment 1 – Algorithms for Data Mining

What is Ridge Regression?

Ridge Regression is basically an extension for linear regression, and a way in which by regularising a linear regression model you can get more accurate predictions from a set of data. It's considered useful in statistics, supervised learning and data mining for this reason. You must find the Sum of Squared Errors (SSE), which is given by the weighted sum of squared errors if the heteroscedastic errors option is not equal to constant variance. In Linear Regression ($y_i = \alpha + \beta x_i + \varepsilon_i$), the SSE is most commonly given by the sum of squares of estimates (ε_i), so:

$$SSE = \sum_{i=1}^{n} (\varepsilon_i)^2 = \sum_{i=1}^{n} \big(y_i - (\alpha + \beta x_i)\big)^2$$

Where α is the estimated values of the constant term a and β is the estimated value of the slope coefficient b.

The Mean Squared Error (MSE) is found by the SSE divided by the number of points (n). The RMSE, square root of MSE, can be used to evaluate the performance of the regression line used. MSE:

Done for each point through to n and divided by the total number of points (n)

$$\mathrm{MSE} = \frac{1}{n} \sum_{i-1}^{n} (Y_i - \hat{Y}_i)^2.$$

The actual y value is subtracted by the predicted y value to give error, then squared to remove negatives

Linear Regression is a way of modelling relationships between data in a linear way, with a straight line, for the purpose of predicting future data. It is done by finding a line with minimum SSE. In the case of one explanatory variable, the process is known as simple linear regression. When more than one explanatory variable is used, the process is known as multiple linear regression. The mean of points can be used to find a centroid to plot from, or occasionally the median point. Often there is a non-linear relationship in data, and this can lead to high bias with a simple linear model. Thus, a more complex model might be necessary in this case. Linear regression are often, but not always, fitted using the least squares approach.

With least squares, the value of y can be found from the y axis intercept added to the slope value multiplied by x. This can give us a point along the line. Least squares measures the difference between actual values of y and the predicted values of y, squares it, and adds up the squared difference of every point to the line in order to get the SSE. Minimising SSE gives the best line. We can predict that the line generated from linear regression accurately reflects the relationship between x and y when we have a lot of measurements. Additional feature variables in linear regression can measure more data against each other. R^2 is the same as with simple linear regression, but with other planes. A curved line over a non-linear dataset using least squares can minimise bias. With little training data though, this complex method alone can be very inaccurate at displaying the relationship between the testing data as well. This leads to high variance in the line generated, which is overfit to the training data. When a line is overfit to training data like this, it can create problems when predicting data. When a testing set is plotted for example, it wouldn't be represented well by the line overfit to the training data because it is usually curved too greatly to minimise bias and thus fit said training data well. Therefore, you

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

would also say that variance is high. Overall with training and testing data combined, SSE would be high. Overfitting is thus associated with high variance and low bias. Underfitting on the other hand, would be where the model doesn't fit the data well enough. Underfitting is associated with low variance but high bias. An example of underfitting would be where a dataset's trend is curved, and a straight line is used to represent the data for predicting future data. As x and y values of new data become greater or lesser (depending on the trend), the underfit line would, like the overfit line, mean greater SSE. To get an accurate model, it is best to balance both bias and variance in order to avoid underfitting and overfitting in order to minimise SSE. With Ridge Regression, a new line must be found that doesn't fit the training data as well, which is done by introducing a small amount of bias into how the new line is fit, known as the weight penalty. This penalty allows for more accurate long-term predictions, as variance is minimised through this method. It is found using lambda multiplied by the slope variable squared of j through to p, with p being number of features of x. This gives us more accurate coefficients for prediction. Unlike Least Squares where just the sum of squared residuals is minimised, Ridge Regression also minimises lambda (regularisation factor) multiplied by the slope squared, which results in a lower ridge regression penalty than a Least Squares fit. Lambda is used to change the slope of the line to make it accurate, and we can use Cross Validation to find the most appropriate value for lambda. This is done in order to limit the ridge regression penalty.

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Cost function

Penalty term is squared magnitude of coefficient, added on in ridge regression to give the objective function

The total number of y, x or other values is n. The total amount of features is p. The iteration of n is i, and the iteration of p is j. B is the objective function.

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

train_data = pd.read_csv ('regression_train_assignment2019.csv')

y_train = np.array(list(train_data['y']))
x_train = np.array(train_data['x'])
features0_train = np.array(list(train_data['features0']))
features1_train = np.array(list(train_data['features1']))
features2_train = np.array(list(train_data['features2']))
features3_train = np.array(list(train_data['features3']))
features4_train = np.array(list(train_data['features4']))
features5_train = np.array(list(train_data['features5']))
features6_train = np.array(list(train_data['features6']))
features7_train = np.array(list(train_data['features7']))
features8_train = np.array(list(train_data['features8']))
features9_train = np.array(list(train_data['features9']))
features10_train = np.array(list(train_data['features10']))
features11_train = np.array(list(train_data['features11']))
features_train = np.column_stack((features0_train, features1_train, features2_train, features3_train, features4_train, featu

test_data = pd.read_csv ('regression_plotting_assignment2019.csv')

x_test = np.array(test_data['x'])
features0_test = np.array(list(test_data['features0']))
features1_test = np.array(list(test_data['features1']))
features2_test = np.array(list(test_data['features2']))
features3_test = np.array(list(test_data['features3']))
features4_test = np.array(list(test_data['features4']))
features5_test = np.array(list(test_data['features5']))
features6_test = np.array(list(test_data['features6']))
features7_test = np.array(list(test_data['features7']))
features8_test = np.array(list(test_data['features8']))
features9_test = np.array(list(test_data['features9']))
features10_test = np.array(list(test_data['features10']))
features11_test = np.array(list(test_data['features11']))
features_test = np.column_stack((features0_test, features1_test, features2_test, features3_test, features4_test, features5_t

n = len(y_train)

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

def getWeightsForPolynomialFit(x, y, degree):
    X = getPolynomialDataMatrix(x, degree)

    XX = X.transpose().dot(X)
    w = np.linalg.solve(XX, X.transpose().dot(y))

    return w
```

Here you can see I have imported pandas to load the dataset from the excel spreadsheet using read_csv function, and I've imported numpy for dealing with matrix operations. I've collected training and testing features into a column stack so it can be input into ridge regression and individual features can be called using their column indices 0-11. I've also took training x and y values as well as testing x values into lists. I took the value of n for the training data using the len() function on y_train. You can also see my polynomial matrix operation functions here, which create a list of ones to account for offset and does the feature expansion up to the input degree for a given data set (x). Another computes the optimal beta values given the input data x, output data y and the desired degree of the polynomial.

```
def ridge_regression(features_train, y_train, regularisationFactor):
    n = len(y_train)

    fNum = 0
    Ex_squared = 0
    Exy = 0
    Ex = 0
    Ey = 0

    betas0 = np.zeros(n) # array of b (y-intercept) am i using this for a predicted line or what
    betas1 = np.zeros(n) # array of m (slope)
    SSEs = np.zeros(n) # sums of squared residuals for each features
    rr_biases = np.zeros(n) # array of ridge regression biases
    weightings = np.zeros(n)

    while fNum < n:
        i = 0
        while i < n:
            Ex_squared = Ex_squared + features_train[i][fNum]**2
            Exy = Exy + y_train[i] * features_train[i][fNum]
            Ex = Ex + features_train[i][fNum]
            Ey = Ey + y_train[i]
            i+=1

        avg_y = Ey / n
        avg_x = Ex / n
        #centroid = np.array(avg_x, avg_y)

        #B1 parameter
        m = (n * Exy - Ex * Ey) / (n * Ex_squared - Ex**2)
        betas1[fNum] = m

        #B0 parameter
        b = (Ey - m * Ex) / n
        betas0[fNum] = b

        i = 0
        while i < n:
            SSEs[fNum] = SSEs[fNum] + (y_train[i] - (m * features_train[i][fNum] + b))**2
            i+=1

        rr_penalty = regularisationFactor * m**2
        #w = getWeightsForPolynomialFit(features_train[:,fNum], y_train, fNum) #
        #weightings[fNum] = w + rr_penalty

        rr_biases[fNum] = SSEs[fNum] + rr_penalty
        fNum+=1

    parameters = rr_biases
    return parameters

def eval_regression(parameters, features, y): # for rmse, get error between predicted and actual values
    # your brilliant code comes here
    eval_n = len(y)
    SEs = np.zeros(eval_n)
    SSE = 0
    learned_ys = features * parameters
    avg_ys = np.zeros(eval_n)

    i = 0
    while i < eval_n:
        avg_ys[i] = np.sum(learned_ys[i,:])/2
        print(learned_ys)
        SEs[i] = (avg_ys[i] - y[i])**2
        SSE+=SEs[i]
        i+=1

    rmse = np.sqrt(SSE / eval_n)
    return rmse
```

Initialised n, then used a while loop to perform numerous sums in order to find the gradient (m) and the intercept (b) of the line, as shown by the code. Getting the slope or B1 parameter involved using the sum of x and y, sum of x, sum of y, sum of squares of x, and sum of x squared. I then used another while loop to calculate the sum of the squared residuals/errors. Generated the penalty

using the regularisation factor multiplied by the slope squared, then finding the overall biases using the SSE and ridge regression penalty.

You can also see above that I implemented a function for evaluation, taking the learned y values gotten from multiplying the parameters with features to predict y, then using a while loop to gather errors squared between actual y values and the predicted y values. Following this, to get the root mean squared error, I made sure SSE was divided by n and square rooted as necessary.

```
126 regularisationFactors = np.array([0.000001, 0.0001, 0.01, 0.1, 1, 10, 100, 1000])
127 reg_n = len(regularisationFactors)
128 #rmses = np.zeros(reg_n)
129 parameters = ridge_regression(features_train, y_train, regularisationFactors[0])
130
131 y_test_l = features_test * parameters # multiply the parameter vector with test set features for output y values
132
133 features = np.vstack([features_train, features_test]) # combine features for plotting
134 y = np.vstack([y_train, y_test_l])
135 x = np.array([x_train, x_test])
136
137 plt.figure(2)
138
139 colors = {0:'r',1:'b',2:'g',3:'m',4:'c',5:'bo'}
140
141 #rng = range[-5, 5]
142
143 plt.scatter(x_train, y_train, alpha=1)
144 plt.scatter(features_test, y_test_l, alpha=1)
145
146 fNum = 0
147 while fNum < n:
148     Xtest = getPolynomialDataMatrix(features_train[:,fNum], fNum+1)
149     ytest = Xtest.dot(parameters[fNum])
150     plt.plot(x_test, ytest, fNum)
151     plt.scatter(x_test, ytest, fNum)
152     fNum+=1
153
154 axes = plt.gca()
155 axes.set_ylim([-1000, 1000])
156 axes.set_xlim([-5, 5])
157
158 plt.title("Ridge Regression")
159 plt.xlabel("x")
160 plt.ylabel("y")
161
162 rmse = eval_regression(parameters, features, y)
```

Regularisation factor is given in an array. Parameters is called, x values are stacked as well as y values, and features are stacked too. I used vstack for this because it enables numpy arrays to be stacked vertically. I then used matplotlib to set colours, labelled x and y axis, giving my plot a title of “Ridge Regression”. I set y values between -1000 and 1000 as required. Used a while loop to go through each feature to plot the lines, calling my function from before.

What is K-Means Clustering?

Step 1 – Selection of "K": Select the number of clusters you want to identify in your data, denoted by "K". K can be selected by observing the data personally on a chart or comparing variance to when other values of K are used. If the latter method is used, an "elbow plot" can be used to determine where sharp reductions in variance end, thus determining the ideal value for K. The higher the value of K, the lower the variation. Variation is measured from the squares of Euclidean distances between each point and its nearest centroid. The total variation is known as the objective function. This must be minimised, as the k-means algorithm is run multiple times to find the lowest possible value for the objective function.

Step 2 – Initialising Centroids: Randomly assign k distinct data points from the dataset. These are the centroids, which are used to cluster the rest of the datapoints around. The best set of centroids are those with minimised objective function.

Step 3 – Assignment Step: Measure the Euclidean Distance between each point and the centroids, assign each point to the nearest centroid to it, then continue assigning points to clusters this way for each point. This is known as the assignment step. In my algorithm, I recorded how many points were in each cluster and what the total values of x and y for each cluster where to carry out the next step.

$$d(\mathbf{p}, \mathbf{q}) = d(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

Figure 3 – Euclidean Distance formula

Euclidean distance is found by subtracting the x and y values from each other at each point, squaring the result from each subtraction to ensure they're positive, then summing together the results from this and square rooting the whole result to counter the fact that the values taken from subtraction are squared to make them positive. This provides the value for Euclidean distance.

Step 4 – Update Step: Find the mean for all objects in each cluster and cluster using the mean values this time. This requires you to have the total number of points in each cluster as well as, in my case, the total x and y values of data points in each cluster. The mean points of each cluster are used as centroids. This step can be derived from the objective function.

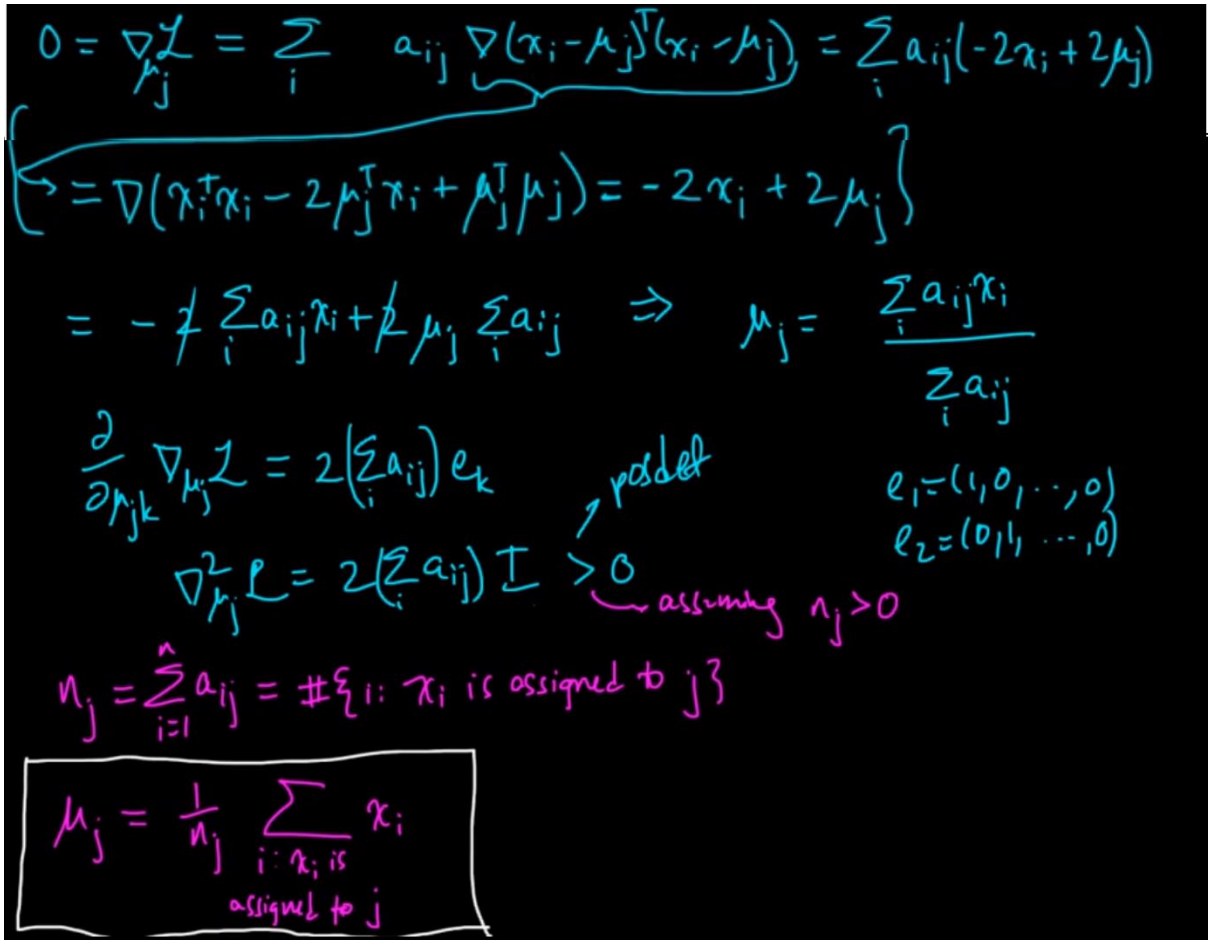


Figure 4 – Calculations to derive the formula for new centroids (empirical mean of points)

Step 5 – Assessment Step: Variance to the mean is added together within each cluster in order to assess the quality of the clustering. Steps 2, 3 and 4 are repeated until the total variance, given as the objective function, is minimised. Usually it's repeated a set number of times until eventually the final centroids and clustering assignments are taken from the iteration with the lowest objective function to find the best clustering.

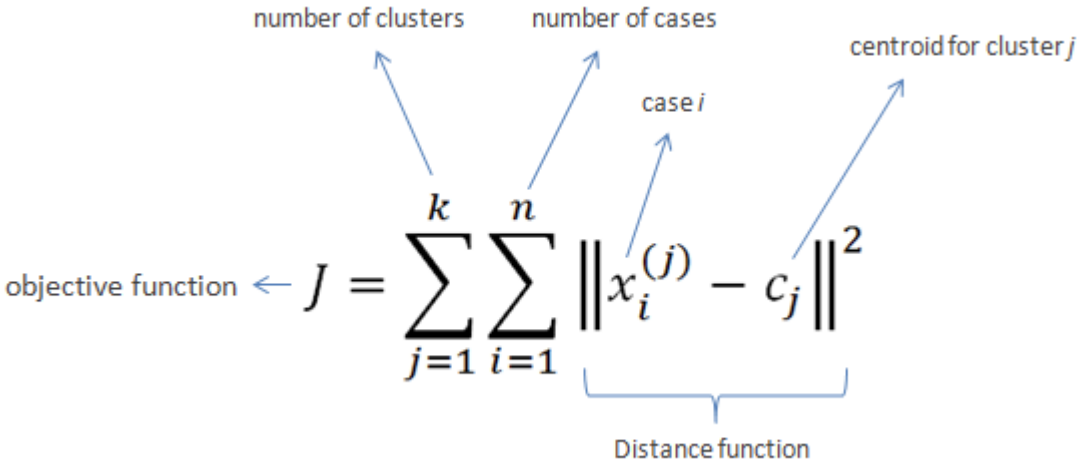


Figure 5 – How to calculate the objective function

As can be seen from the above formula, the objective function is found by the sum of the distances from each point (n) to the centroids (k) of their given cluster. Squaring is done in the distance function to eliminate negative values. The objective function is made up of the distances in all k clusters.

One of the advantages of K-Means Clustering is that it's relatively easy to implement, given that it involves repeating a number of steps and can be achieved through iteration. Another is that K-Means may be faster than hierarchical clustering as long as K is a small value when there's a high number of variables. It can also provide higher clusters when compared to hierarchical clustering. Furthermore, an instance can change cluster when centroids are recomputed.

On the other hand, one of the disadvantages of K-Means Clustering is the random choice of initial clusters as it can lead to different clustering results from other times it's ran. Results can often differ, and it can take a long time to find accurate results when the dataset is quite large. It can also be considered a drawback that K must always be determined before the algorithm is ran rather than having a calculated way to determine k, although this could be implemented to some extent based on measuring sharpness in the reduction in variation for each increasing value of k.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
%matplotlib inline

data = pd.read_csv('CMP3744M_ADM_Assignment 1_Task2 - dataset - plants.csv')
stem_length = np.array(list(data['stem_length']))
stem_diameter = np.array(list(data['stem_diameter']))
leaf_length = np.array(list(data['leaf_length']))
leaf_width = np.array(list(data['leaf_width']))

x = np.concatenate((stem_length, leaf_length), axis=None)
y = np.concatenate((stem_diameter, leaf_width), axis=None)
dataset = np.column_stack((x, y))

stemset = np.column_stack((stem_length, stem_diameter))
leafset = np.column_stack((leaf_length, leaf_width))

n1 = len(stem_length) # length of stemset of leafset
n = n1 * 2 # length of datasets put together

def compute_euclidean_distance(vec_1, vec_2):
    # your code comes here
    # find distance between centroid and point
    # vec_1 is the centroid, while vec_2 is the point
    # sqrt((x1 - x2)^2 - (y1 - y2)^2)
    distance = np.linalg.norm(vec_1-vec_2)
    return distance

def initialise_centroids(dataset, k):
    # your code comes here
    # centroids[i] = [x, y]
    centroids = dataset[np.random.randint(dataset.shape[0], size=k), :]
    return centroids
```

Using Jupyter notebook to type on and run my Python file within, I started off by importing the necessary libraries, giving them shorthands like "np" for numpy or "pd" for pandas. The library matplotlib was imported for plotting, whereas numpy was for matrix manipulation and pandas was for importing the dataset from excel using "read_csv" and following that with the file location. As can be seen from the code, I created a list of each column in the excel file, then concatenated stem_length with leaf_length for x values and concatenated stem_diameter and leaf_width for y values, putting them all in one dataset with column_stack to run through the algorithm. I also made

separate datasets for stem and leaf for later plotting and observation. Also calculated the n (length) variables for the whole dataset and the separate datasets using len(stem_length).

You can also see my function for computing Euclidean distance, which was simply done by importing the vector for centroid and the vector for a point, and using linalg.norm from numpy which ensures each distance is positive without having to square each value and square root the sum. As commented, vec_1 is the centroid and vec_2 is the point being iterated through within the k-means function.

I also found a simple way to initialise the centroids, although I went through a few different methods before realising a one-line way in which I could just use numpy's random.randint to select k random rows from the dataset and put them into a 2-D numpy array (centroids).

```
def kmeans(dataset, k):
    # iterate through each point, computing euclidean distance between each randomly initalised centroid and the point
    # distance is compared with each centroid, and point is assigned to the closest centroid
    cluster_assigned = np.zeros(n) # array to store cluster assignments and add to dataset when algorithm is complete
    points_per_cluster = np.zeros(k) # array to store number of data points for each centroid
    xytot_per_cluster = np.zeros((k, 2)) # stores the total x any y for all points in a cluster
    dis_total = np.zeros(k) # array to store total distance for each centroid's cluster

    maxstep = 100
    step = 0
    while step < maxstep: # or (step > 50 and var == min(obj_func))
        # centroids are initialised using appropriate function, returned as array that has k rows and 2 columns of x and y
        centroids = initialise_centroids(dataset, k)
        new_centroids = np.zeros((k, 2)) # for comparison of new and old centroids

        done = False
        while done == False: # kmeans function runs until new and old centroids are equal
            var = 0 # stores sum of squared euclidean distance of every cluster
            i = 0 # i is for iterating through each point
            while i < n: # iterate through each point until n is reached
                point = np.array([dataset[i][0], dataset[i][1]]) # set point of interest
                dis_array = np.zeros(k) # create array to store distances between the point and centroids

                u = 0 # u is for iterating through each centroid
                while u < k: # iterate through to k in order to calculate each distance and store in an array
                    distance = compute_euclidean_distance(centroids[u], point) # compute distance between point and centroi
                    dis_array[u] = distance # store distances in array
                    u+=1

                u = 0 # u is for iterating through each distance in this case
                while u < k: # iterate through to k in order to compare each distance
                    if dis_array[u] == min(dis_array): # if distance u is smallest
                        cluster_assigned[i] = u+1 # add assigned cluster to cluster_assigned array
                        xytot_per_cluster[u] = xytot_per_cluster[u] + point # so far adding up x and y values in each cluste
                        points_per_cluster[u] = points_per_cluster[u] + 1
                        dis_total[u] = dis_total[u] + min(dis_array)
                        var = var + min(dis_array)**2
                    u+=1
                i+=1

            u = 0 # u is for iterating through each centroid
            while u < k: # iterate through to k to assign new centroids based on mean of points in each cluster
                new_centroids[u] = xytot_per_cluster[u] / points_per_cluster[u] # find mean of x and y values for each clust
                u+=1

            dis_total.fill(0) # reset all dis_total values to 0 for next step
            points_per_cluster.fill(0) # reset all points_per_centroid values to 0 for next step
            xytot_per_cluster.fill(0)

            if centroids.all() == new_centroids.all():
                done = True
            else:
                centroids = new_centroids

        if step == 0: # if and else statements to store the objective function in each step
            obj_func = np.array(var)
            cluster_assigned_steps = np.array(cluster_assigned) # create array of cluster assignments in each step to be com
            centroid_steps = np.array(centroids.flatten()) # create array to store centroids from each step
        else:
            obj_func = np.append(obj_func, var)
            cluster_assigned_steps = np.vstack([cluster_assigned_steps, cluster_assigned])
            centroid_steps = np.vstack([centroid_steps, centroids.flatten()])
        step+=1
```

The above code shows the assignment of arrays/variables and looping structure of my k-means algorithm function. You can see that I started by setting arrays of zeros which are used for calculating necessary values during the assignment stage. They're zero values so that they can be added to during the assignment process. One of these is the cluster_assigned array for storing the assignment values of each data point, stored in a big 2-D array for each step called cluster_assigned_steps and added onto the dataset according to which step is found to be optimal. Others include points_per_cluster and xytot_per_cluster, which are used for finding the mean point

of each cluster. The values of these arrays are set back to 0 at the end of each loop through the assignment stage, which is ended when centroids and new_centroids (found using mean points of each cluster) are found to be equivalents using an if statement. If not ended, the old centroid values are updated with new centroid values and the process restarts. Using a Boolean value to end the loop allowed me to repeat the loop without complications such as centroids being updated and causing a problem. The assignment stage works by iterating through each point through to n, computing Euclidean distance to each centroid by iterating through to k and storing each distance in an array, which is then used in an if statement to determine the minimum distance. This is how cluster assignments are made, and the point in which other stats are found including the objective function (given as var individually but stored as obj_func in each step). You can see how throughout the program, simple while loops are used to iterate through arrays of data and if statements are also used. This is a method I found easy to code mathematical algorithms with and simple to observe my code with.

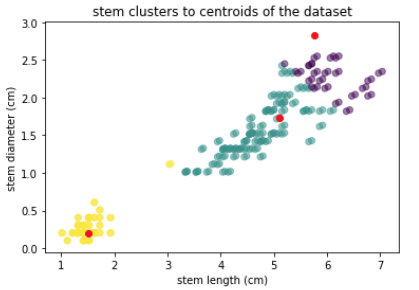
Centroids are initialised at the beginning of each step, with the number of iterations through the algorithm being decided by the variable maxstep and the iteration step incremented after each run. You can change maxstep depending on how many times you want to run through the algorithm. If using a larger dataset, maxstep might be a bigger value. You can see before the step variable increments that vstack is used to combine stats data from every step, such as var into obj_func, cluster_assigned into cluster_assigned_steps, and centroids into centroid_steps.

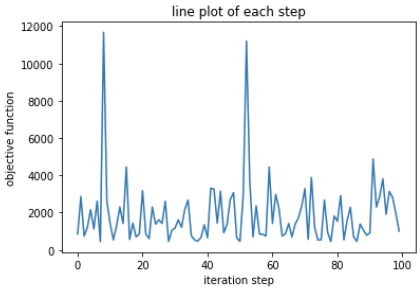
```
    i = 0
    while i < step: # iterate through each step
        if obj_func[i] == min(obj_func): # find if the step holds the minimum objective function
            optimal_step = i # assign the optimal step
        i+=1

    u = 0
    while u < k*2:
        centroid = np.array([centroid_steps[optimal_step][u], centroid_steps[optimal_step][u+1]])
        if u == 0:
            centroids = np.array(centroid)
        else:
            centroids = np.vstack([centroids, centroid])
        u+=2

    cluster_assigned = cluster_assigned_steps[optimal_step] # use optimal step variable to assign best cluster set
    clustered_dataset = np.column_stack((dataset, cluster_assigned)) # add cluster_assigned as a column to new dataset

    # stem scatter plot :
    plt.figure(1)

    colors = {0:'k',1:'b',2:'g',3:'y',4:'m'} # denotes cluster colours
    center_color = ['r'] # denotes centroid colour as red

    #leafset = clustered_dataset[:-n1, :] # delete first n1 rows of clustered_dataset to get leafset for plotting
    stemset = clustered_dataset[n1:, :] # delete last n1 rows of clustered_dataset to get stemset for plotting

    plt.scatter(stemset[:,0], stemset[:,1], c=stemset[:,2], alpha=0.5)
    plt.scatter(centroids[:,0],centroids[:,1], alpha=1, color=center_color[0])

    plt.title("stem clusters to centroids of the dataset")
    plt.xlabel("stem length (cm)")
    plt.ylabel("stem diameter (cm)")

    # line plot :
    plt.figure(2)

    its = list(range(step))

    plt.plot(its, obj_func)

    plt.title("line plot of each step")
    plt.xlabel("iteration step")
    plt.ylabel("objective function")

    return centroids, cluster_assigned
```

In the final part of my k-means function, I find the minimum objective function by iterating through each step and finding out the minimum objective function. This gives me the optimal step. From this, I can select the appropriate cluster assignments and centroids from the data stored along the way. Using column_stack on dataset and cluster_assigned gives the clustered_dataset, assigning a cluster to each point in the dataset.

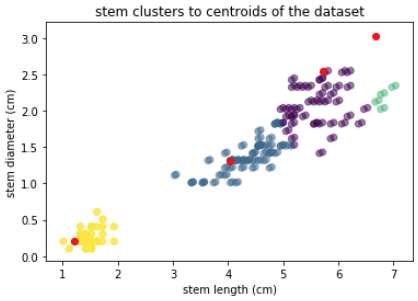
After this, the plotting is done using matplotlib. First the scatter plot using plt.scatter, then the line plot using plt.plot, and colours selected for the scatter plot. Centroids are denoted as big red dots. Y and X axis for each plot are labelled accordingly. The scatter plot shows just stem datapoints and their clusters where centroids are selected from the whole dataset put together.

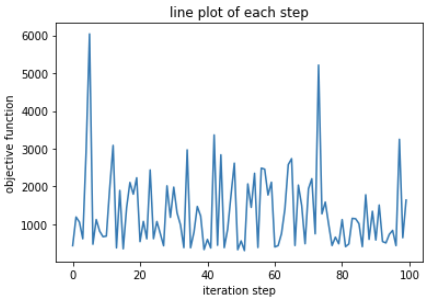
```
In [4]: 1 centroids_k3, cluster_assigned_k3 = kmeans(dataset, 3) # implement kmeans function (k=3) to get centroids and cluster values
```






```
In [3]: 1 centroids_k4, cluster_assigned_k4 = kmeans(dataset, 4) # implement kmeans function (k=4) to get centroids and cluster values
```





You can compare k values and how stemset fits the rest of the data from the resulting plots. Generally, the centroids lie along where the stem datapoints are. You can see from where k=3 that it is certainly possible to cluster the plants into 3 groups, judging by the general equal size of the clusters. If needed, I could easily plot points_per_cluster of the optimal step in order to observe this possibility further. Computing an average objective function for each k value could give a good measure of reduction of variation per k value, which can be observed with an elbow plot. This could also help to provide observations on how many categories (k) there should be for these plants.