Contents

Polynomial Regression

Description

What is Polynomial Regression?

Considered to be a form of multiple linear regression, polynomial regression is a form of regression analysis whereby an nth degree polynomial is modelled using an independent variable (x) and a dependent variable (y), establishing an estimated relationship between the two variables.

A polynomial itself is simply 'a mathematical expression consisting of one or more algebraic terms each of which consists of a constant multiplied by one or more variables raised to a nonnegative integral power' (Merriam-Webster.com, 2020).

The independent variable cannot be changed; thus it is classified as independent in the first place. The dependent variable however is named in such a manner due to the consideration that their value is reliant upon a mathematical function, rule, or law. Due to these dependencies, the dependent variable can be changed by modifying the independent variable, but not vice-versa.

The equation for polynomial regression can be summarised as follows:

$$y_i = \beta_0 + \beta_1 X_i + \beta_2 X_i^2 + \cdots + \beta_m X_i^m + \varepsilon_i \ (i = 1{,}2 \ldots n),$$

Figure 1 – polynomial regression

where:

y is the dependent variable.

β are the weights.

X is the dependent variable.

ε is the error rate.

This can also be defined as a matrix equation:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_3 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \vdots \\ \beta_n \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

Figure 2 – polynomial regression as matrix equation

<u>What are Error Functions in Regression?</u>

Error functions are terms in regression used to add bias to the proposed prediction model, and predict how close said model is to actual results.

<u>What is Linear Squares model?</u>

The linear squares model is a form of regression where the line of best fit is approximated for a set of data (Watson 1967). Each point calculated represents the relationship between the actual point and the predicted point.

<u>What is Least Squares Solution?</u>

The least squares solution is used to predict the authenticity of a prediction model of regression analysis (Miller 2006). It takes the sum of residuals from a set of data and attempts to minimise said sum squared.

<u>How are Polynomial Features used in Linear Regression?</u>

Polynomial features are used in linear regression to assist in transforming the line of best fit to a degree of n. Each feature is used to predict the co-ordinates (or data points) for the line of best fit (Yoshinga et Al, 2009)

<u>What is the Difference between Training and Test sets' Performance?</u>

The training set is often larger in proportion to the test set, and it is used to generate a valid model for the base system to use. The aptly named test set is used to test the overall effectiveness of the model. Providing a model with a training and test set allows one to verify it's accuracy.

Implementation

The following modules were used in the implementation of K-means clustering using the 'task1.csv' data:

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

Figure 3 – importing modules for polynomial regression

```
regData = pd.read_csv("Task1Data.csv") #The file path for the regression data.
```

Figure 4 – reading data

The data is then split into two separate arrays – X and Y respectively – based off the columns in the .csv file and then sorted to accommodate plotting on the graph (figure 5).

```
#The data is split into two variables for plotting: X and Y.
Data = regData.iloc[:, 1:3]
#The data is then sorted to accommodate plotting once polynomial regression has been done.
Data.sort_values(by=['x'], inplace=True)
```

Figure 5 – true data for polynomial regression

This makes no difference to the integrity of the data itself.

Afterwards, a preliminary graph plotting the initial data is created to provide a baseline ideology of how the regression models should realistically progress in respect to the true data.

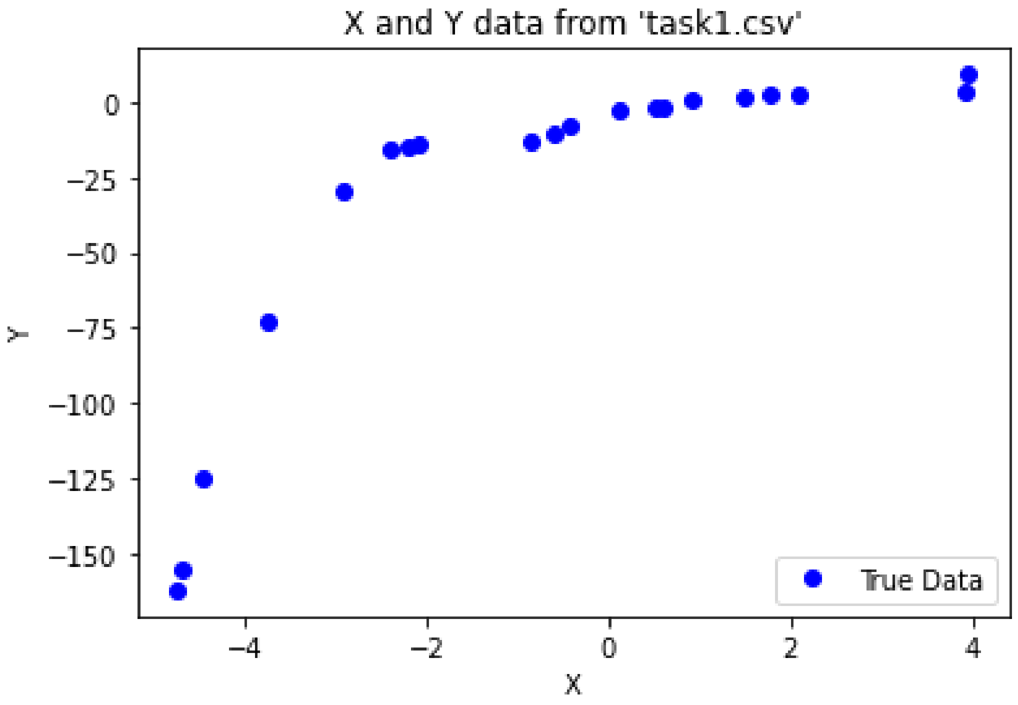


Figure 6 – true data for polynomial regression

This graph, and all following graphs, was designed using the matplotlib.pyplot package, as demonstrated below:

```
#Plotting the data initially.
plt.plot(x, y, 'bo', label="True Data") #Plots data onto a graph, assigns it a colour, and labels it.
plt.xlabel("X") #Sets the X label
plt.ylabel("Y") #Sets the Y label
plt.title("X and Y data from 'task1.csv'")
plt.legend(loc="lower right") #Uses labels to create a legend.
plt.show() #Shows the graph.
```

Figure 7 – plotting graphs

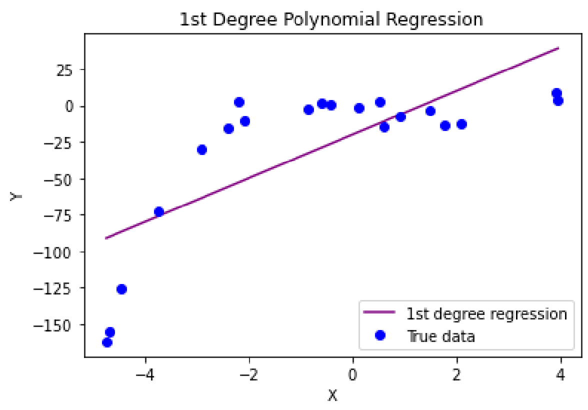
Polynomial Regression Formula
The following formula has been programmed to compute polynomial coefficients, making extensive use of numpy (as 'np') matrix operations:

```
def pol_regression(features_train, y_train, degree): #Function to compute polynomial coefficients.
    X = np.ones(features_train.shape) #Creates a premade array to increase computational speed.
    for i in range(1,degree+1): #To the nth degree.
        X = np.column_stack((X, features_train ** i)) #Creates an array of X values multiplied by power of i (to n).
    XX = X.transpose().dot(X) #Creates a Vandermode matrix from X.
    w = np.linalg.solve(XX, X.transpose().dot(y_train)) #Creates polynomial weights.
    pol_coefficients = X.dot(w) #Creates polynomial coefficients.
    return pol_coefficients #Returns them.
```

Figure 8 – polynomial regression code

This has been applied to the 'task1.csv' data to degrees of 1, 2, 3, 6, and 10, as shown in figures 6 through 10.

Polynomial Regression Graphs



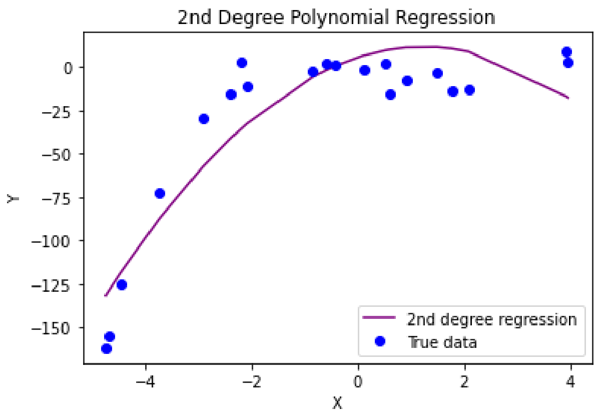
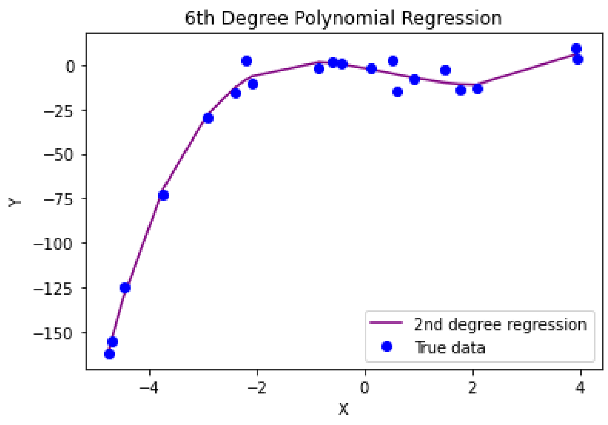


Figure 9 - *regression to the 1st degree* Figure 10 - *regression to the 2nd degree*



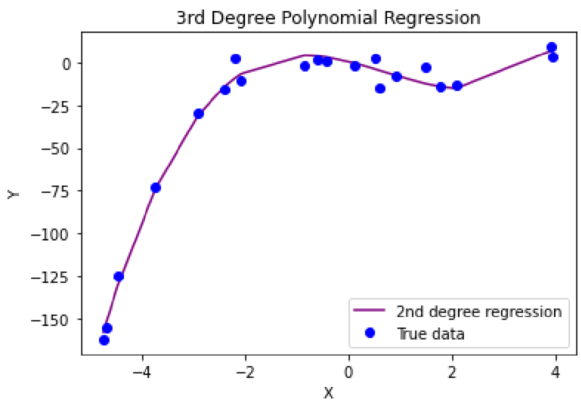


Figure 11 - regression to the 3rd degree *Figure 12 - regression to the 6th degree*

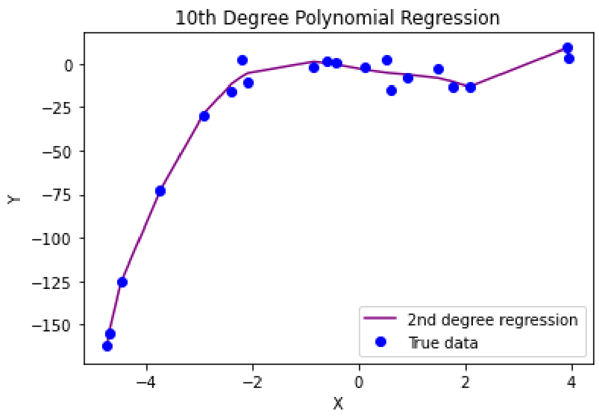


Figure 13 - regression to the 10th degree

This was achieved by using the 'pol_regression' function defined above in the following context:

```
#Polynomial Regression of the 1st degree.
y_1 = pol_regression(xData, yData, 1) # 1st degree regression.
plt.plot(xData, y_1, 'purple', label='1st degree regression') #Plots regression data.
plt.plot(xData, yData, 'bo', label='True data') #Plots true data.
plt.xlabel("X")
plt.ylabel("Y")
plt.title("1st Degree Polynomial Regression")
plt.legend(loc='lower right')
plt.show()
```

Figure 14 – using regression function

For each degree of polynomial requested, this code was used in similar context with only the change in degree necessitating a change. The code cannot compute a polynomial with 0 degrees, as there is no possible change that can be observed from such a degree, thus error handling was implemented to detect an input degree of 0 or lower and convert it to a degree of 1:

Evaluation

Root Mean Squared Error Formula

In order to calculate the RMSE of each training and test set, the major important feature that had to be calculated were the polynomial weights for each degree: those could be used to calculate polynomial regression on training/test sets of varying sizes. Those coefficients could then be used to calculate the RMSE of the set, and thus the efficacy of the method used.

To achieve this preliminary goal, first the function to calculate polynomial regression was re-implemented and modified to only calculate and return the weights of regression, given x, y, and the necessary degree.

```
def get_weights(features_train, y_train, degree): #Function to compute polynomial coefficients.
    if(degree <= 0): # Error handling to catch an invalid degree
        degree = 1
    X = np.ones(features_train.shape) #Creates a premade array to increase computational speed.
    for i in range(1,degree+1): #To the nth degree.
        X = np.column_stack((X, features_train ** i)) #Creates an array of X values multiplied by power of i (to n).
    XX = X.transpose().dot(X) #Creates a Vandermode matrix from X.
    w = np.linalg.solve(XX, X.transpose().dot(y_train)) #Creates polynomial weights.
    return w # Returns weights alone.
```

Figure 15 – code to retrieve polynomial weights

As this returns the weights alone, another function was created to calculate polynomial coefficients, and thus regression, using x, y, and pre-calculated weights. Those weights could now be interchanged between training and test sets.

```
def tt_regression(features_train, y_train, weights):
    X = np.ones(features_train.shape) #Creates a premade array to increase computational speed.
    # No need for error handling regarding degree, as that has already been handled in the function to calculate weights.
    for i in range(1,len(weights)): #To the nth degree.
        X = np.column_stack((X, features_train ** i)) #Creates an array of X values multiplied by power of i (to n).
    pol_coefficients = X.dot(weights)  # Calculates predicted values
    return pol_coefficients # Returns polynomial coefficients
```

Figure 16 – polynomial regression with pre-set weights

Finally, with each set group now being trained using the same weights, the RMSE could be fairly and accurately tested, which was done so using standard RMSE calculations (figure something)

```
def eval_pol_regression(parameters, x, y): #Computes RMSE of each regression
    rmse = 0 #Sets to 0 by default
    for i in range(len(x)): #For every value in X
        rmse +=(parameters[i] - y[i]) ** 2 #Adds the predicted output minus the actual output squared to the RMSE
    rmse = rmse / len(x) #Once that loop is finished, the final RMSE is calculated by dividing it by the input array size.
    return rmse #Returns the RMSE
```

Figure 17 – polynomial regression analysis using RMSE

Once again, this process was repeated for every necessary degree and set; the results of which were thereupon combined to create a chart showing the overall RMSE of each set and to what degree.

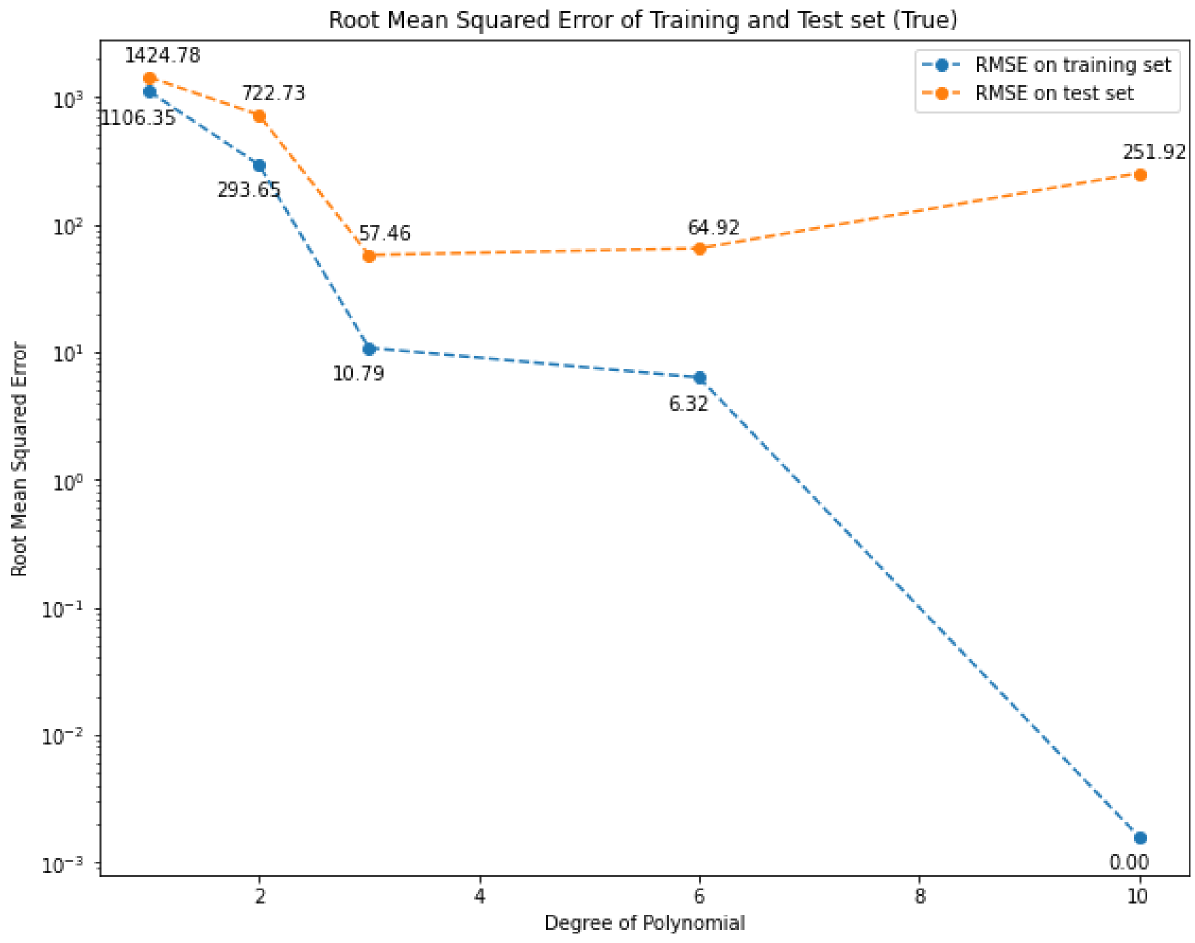


Figure 18 - RMSE on training and test sets (true)

From this chart alone, it is clear that the least effective degrees (respectively) are a degree of 1 and 2. These degrees have the highest RMSE values for both sets, proving them as ineffective for data in similar shape. To a degree of 3 however, the application differs between each set, as the training set RMSE continues to regress efficiently through degrees 6 and 10, whereas the test set finds it's RMSE minimum at degree 3, steadily raising yet again through further degrees.

Using this information it is possible to extrapolate the following:

- The training set overfits itself at degree 6 and 10.
- This data does not suit a polynomial of degree 1 or 2 – though applicable, it has a high RMSE for each set.
- From the given degrees, the most accurate and efficient is a polynomial regression function of degree 3 (cubic regression) as this is most likely to be accurately used on other data sets.
- There is a potential that a degree of 3 is not the most effective regression degree available for these two data sets. From degree 2 to 3, the reduction is comparable to (on average) an approximate 94% decrease in RMSE. Though a reduction of this quantity indicates a near maximum efficiency, it is possible, though unlikely, that the RMSE could continue to decrease on degrees of 4 or 5.

In case of data being unfairly quantifiable and biased as true data, the RMSE of each set was calcualted again, but with each set originating their data from a shuffled parent data set. As the data is shuffled it is incredibly unlikely to find a repeat of one chart (there are 20 items in each data set, thus a possible 20! different orders for the items to be in, or 2.432902e+18 possible combinations), making it difficult to record such data.

However, a number of examples of this has been recorded nonetheless, alongside the differentiation used to calculate it.

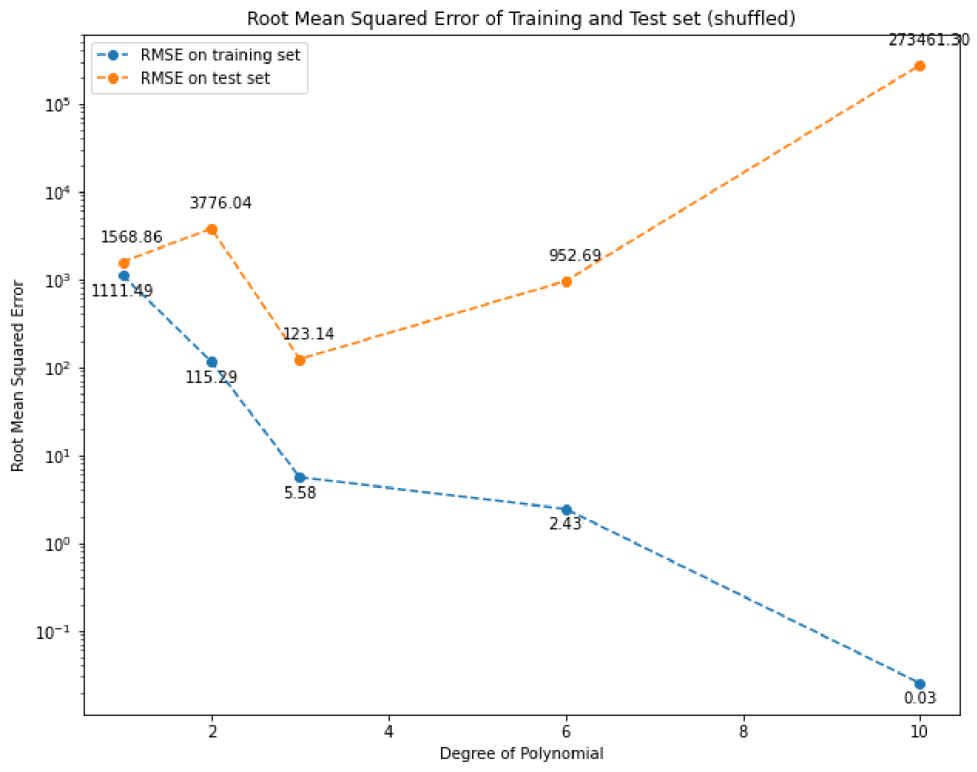


Figure 19 - RMSE on training and test sets (shuffled) – 1

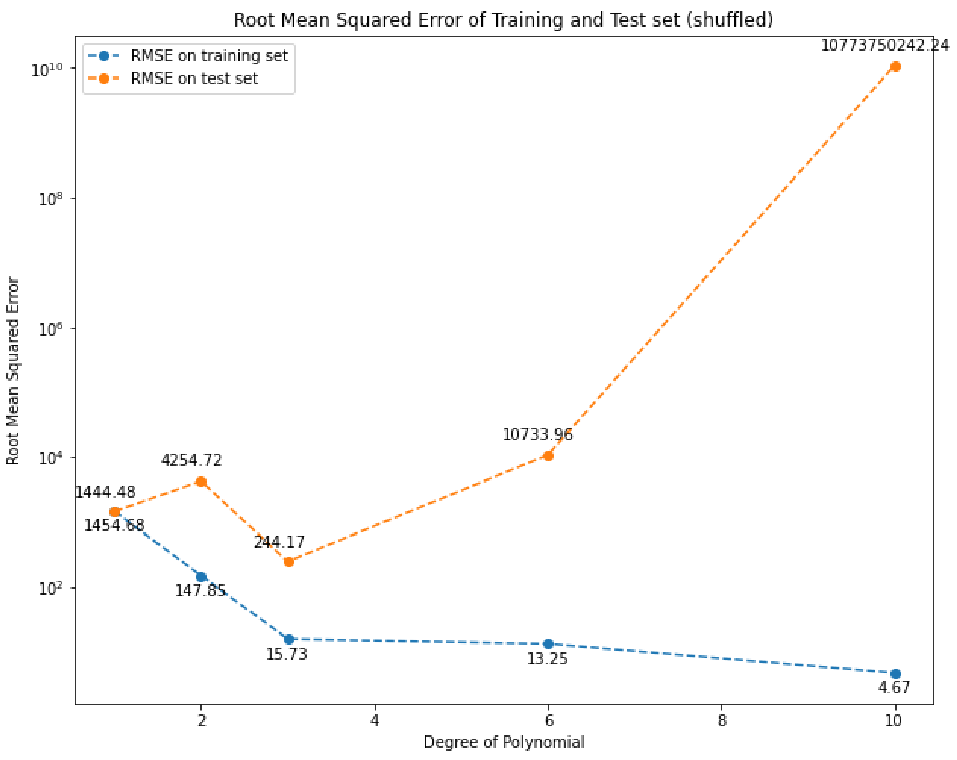


Figure 20 - RMSE on training and test sets (shuffled) – 2

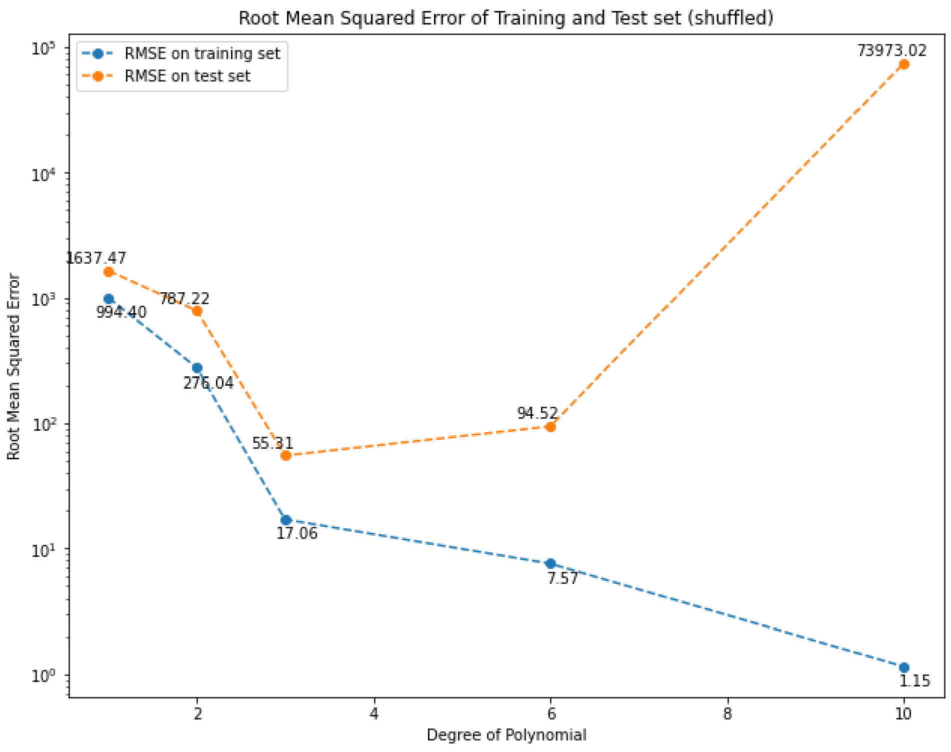


Figure 21 - RMSE on training and test sets (shuffled) – 2

These three charts (figures 16 to 18) show a similar trend amongst one-anther that, despite arcing changes in the effectiveness of earlier degrees, polynomial regression to a degree of 3 is the most effective in this scenario

K-Means Clustering

Description

<u>What is K-means Clustering?</u>

K-means clustering is a method of vector quantization whereby a number of k clusters is produced in a of data. Each data point is linked to a centroid – the central point of each cluster – normally through the lowest mean distance of each data point to every centroid.

<u>What are its Strengths and Weaknesses?</u>

- It is computationally fast and maintains its effectiveness when used on large datasets. It is adaptable and scalable.
- The approximation of clustering can often be sub-par when centroids are randomly initialised.
- K-means clustering is heavily influenced by outliers in data. This can be mitigated through their removal.
- It is ineffective when used with clustered data in varying size and density.

<u>What is an Objective Function?</u>

The overall objective of K-means clustering is to reduce the amount of intra-cluster variance by partitioning the data into k sets.

<u>What is a Centroid?</u>

A centroid is the centre of a cluster of data, calculated either at random or through consistent iteration of each data cluster, where the overall distance is calculated from every data point to each centroid, producing new centroids as the overall mean changes.

<u>What is the Euclidean Distance?</u>

Euclidean distance is the length of a line segment between two points. This distance is often calculated via the formula:

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

Figure 22 – Euclidean Distance formula

where:

d is the Euclidean distance.

p is one point (x and y are p_1 and p_2)

q is the second (x and y are q_1 and q_2)

This formula expands to every dimension establishes, so in shorthand the formula can also be written as:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 \ldots (p_n - q_n)^2},$$

Figure 23 – Expanded Euclidean Distance formula

where,

n is the number of dimensions.

Whilst a suitable formula for calculating Euclidean distance, it is does become more inaccurate as more dimensions are added – a theorem known as the curse of dimensionality. There are methods to reduce the number of dimensions whilst maintaining data integrity, but such formulas are not required for this assessment.

What is the Assignment Step?

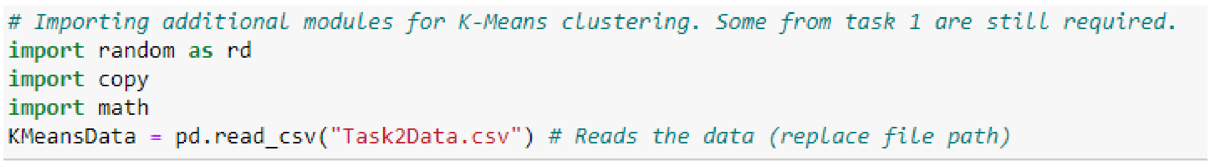
After the centroids are initialised the assignment step is taken where the Euclidean distance from each data point to every centroid is calculated, assigning each point to its closest centroid (Roy et al, 2008). This is the assignment step.

What is the Update Step?

The update step is the recursive step of K-means clustering. After the centroid initialisation and Assignment step is completed, the update step is used to generate more accurate clusters of data. The summation of each cluster is produced to generate an average data point (usually the mean) which acts as the new centroid, and the assignment step is then repeated to calculate new clusters. This process is repeated until the assignments do not change, or until another goal is reached (for example, once this process has iterated 50 times).

Implementation

Firstly, additional modules were required for this implementation of K-Means clustering to operate, and the data was loaded in from the required .csv file (figure 24)..



```
# Importing additional modules for K-Means clustering. Some from task 1 are still required.
import random as rd
import copy
import math
KMeansData = pd.read_csv("Task2Data.csv") # Reads the data (replace file path)
```

Figure 24 – loading modules and data

The following charts have been produced to show the implementation of K-means clustering algorithm. Due to the data being calculated using four features, and thus 4 dimensions, it is difficult to effectively plot it on a 2-dimensional chart. As such, this representation of K-means clustering is dramatically more succinct and easier to interpret when applied using data features that are evidently grouped as the axis for the graph. When data is spread out rather roughly, it is difficult to find the correct clusters when there are multiple hidden layers changing the real Euclidean distance from the perceived.

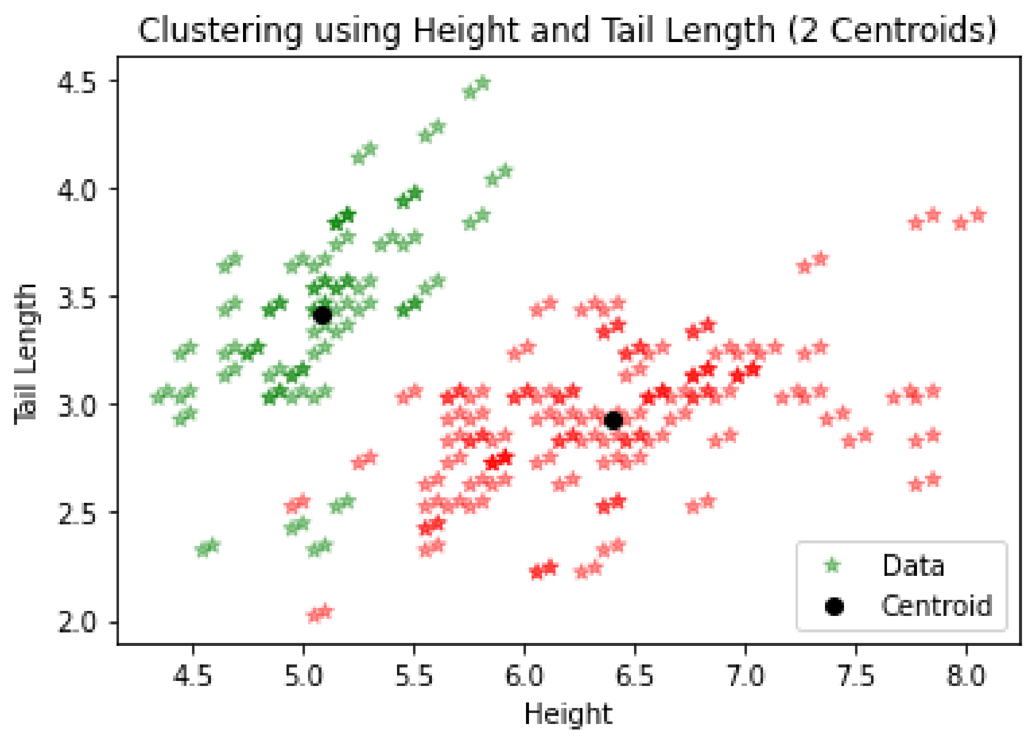


Figure 24 – K means clustering using 2 centroids and Height and Tail length as the axis (x, y)

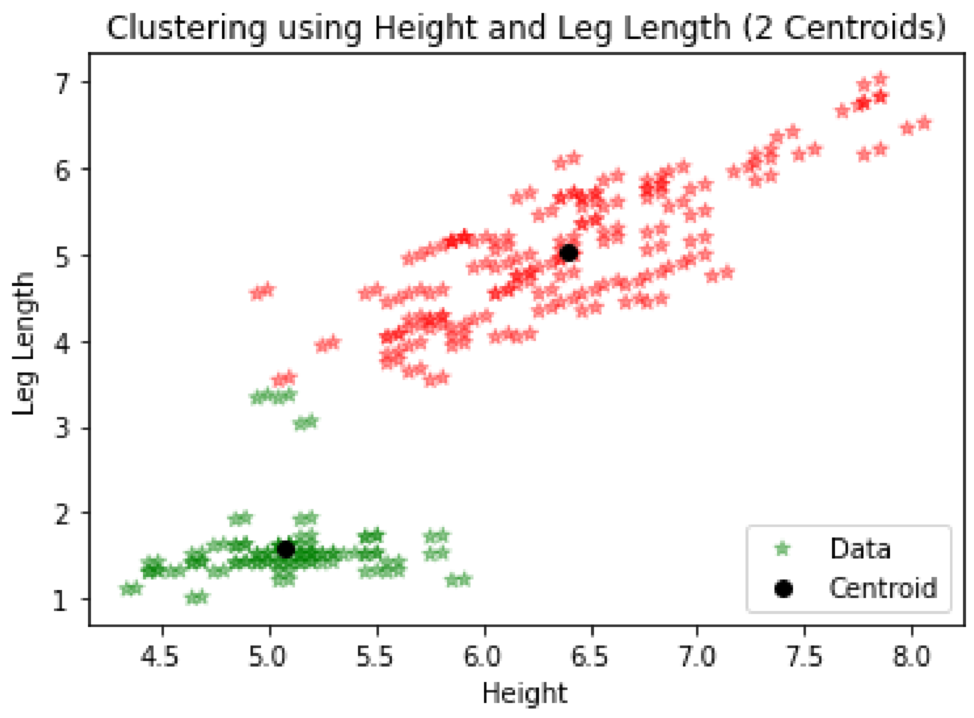


Figure 25 – K means clustering using 2 centroids and Height and Leg length as the axis (x, y)

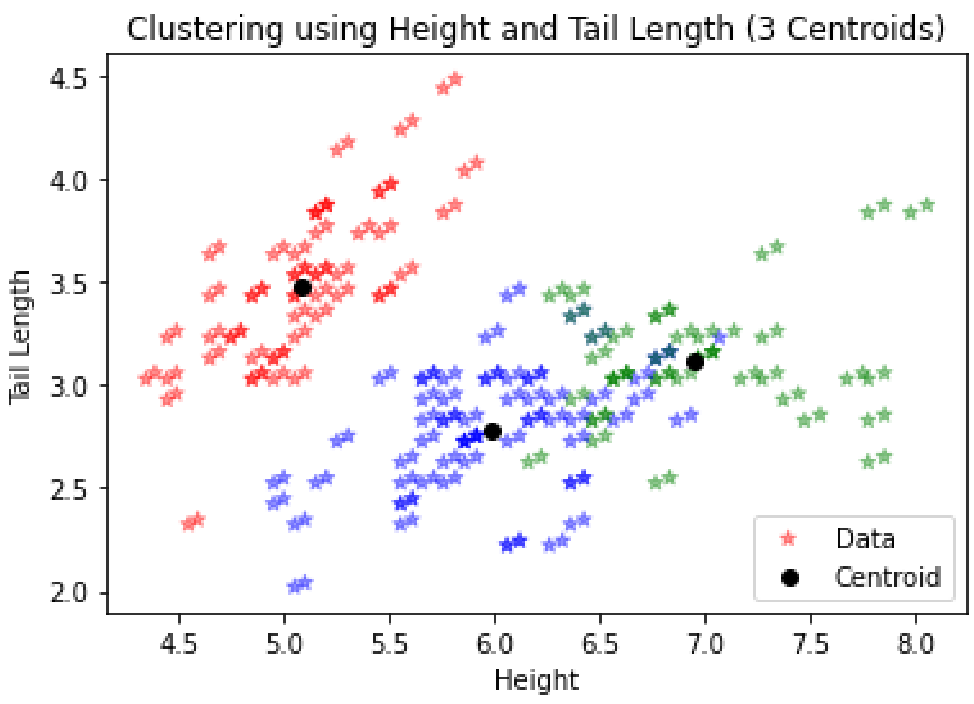


Figure 26 – K means clustering using 3 centroids and Height and Tail length as the axis (x, y)

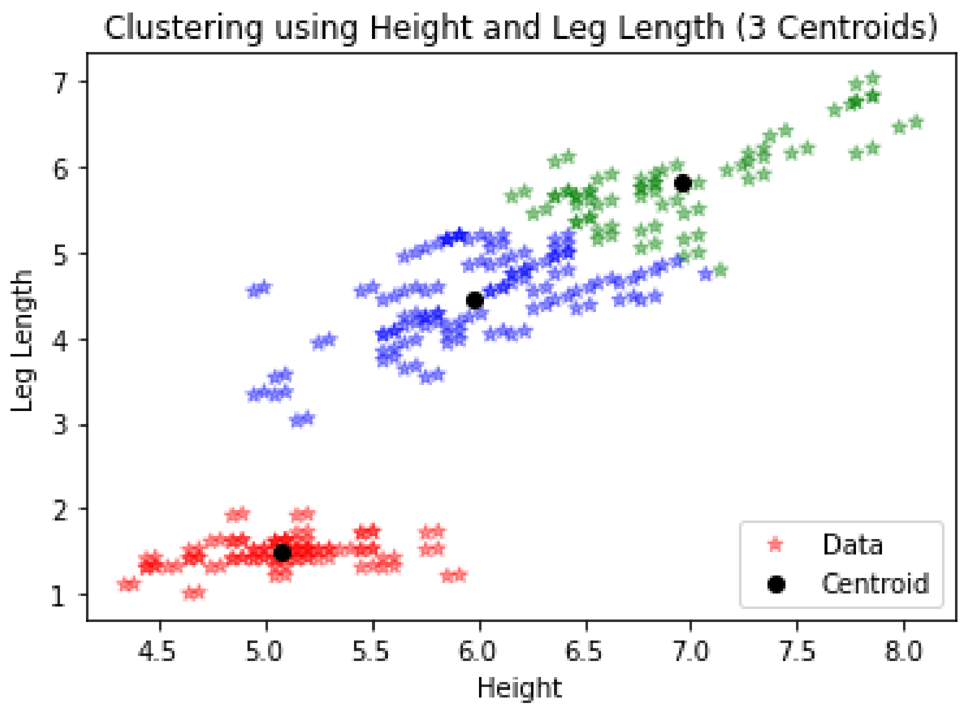


Figure 27 – K means clustering using 3 centroids and Height and Leg length as the axis (x, y)

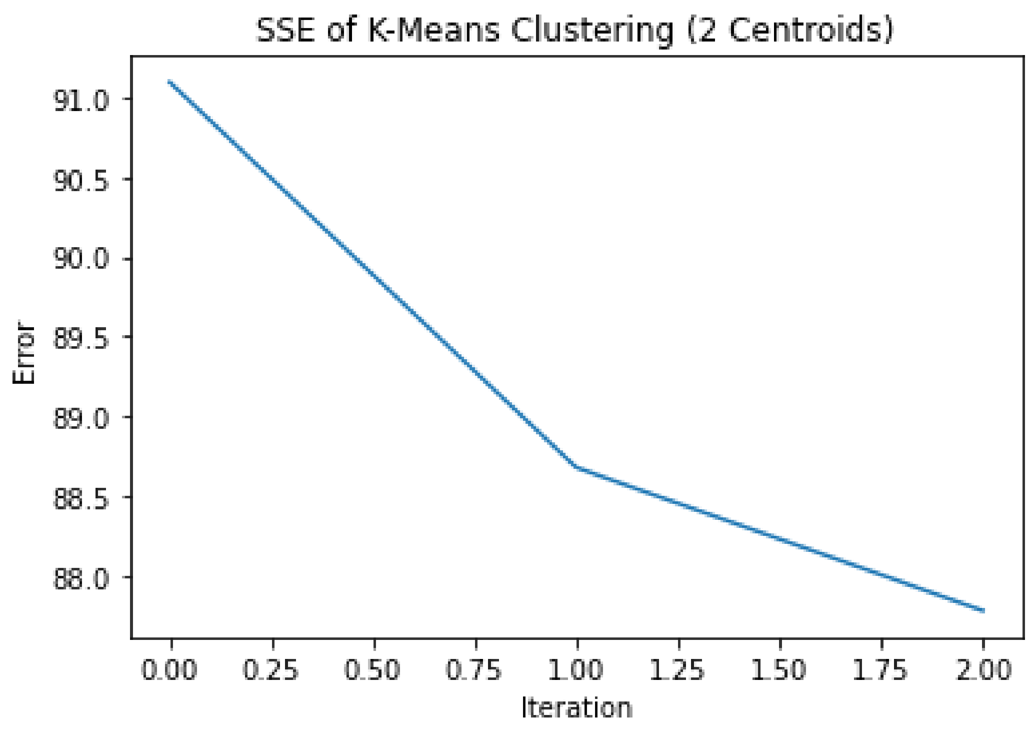


Figure 28 – Error of K-Means Clustering (2 Centroids)

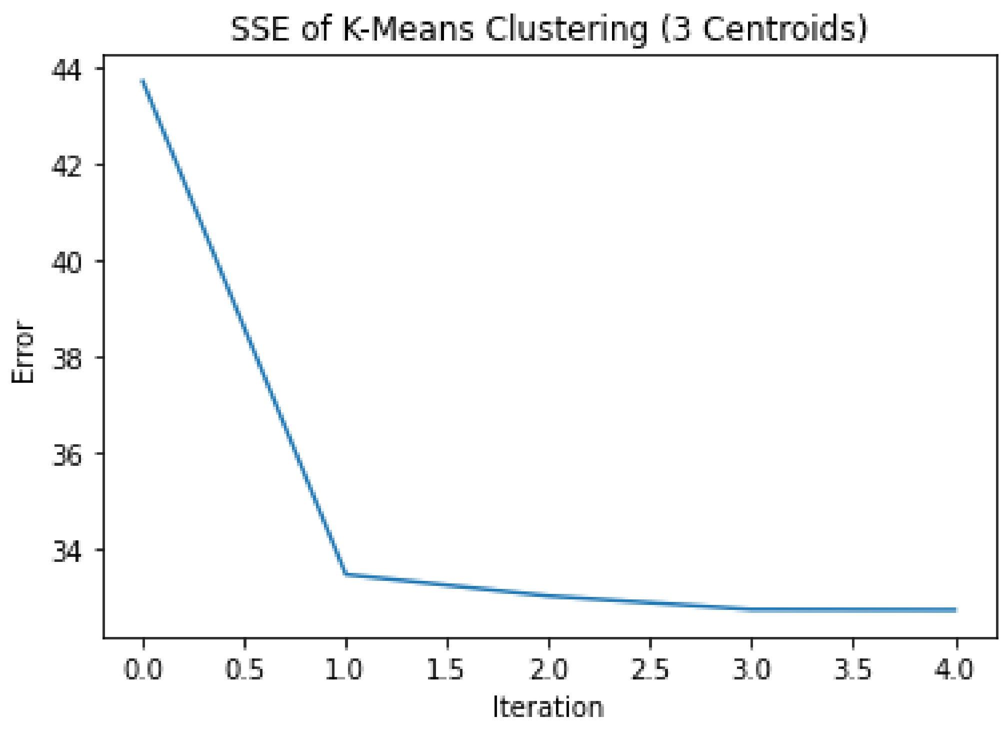
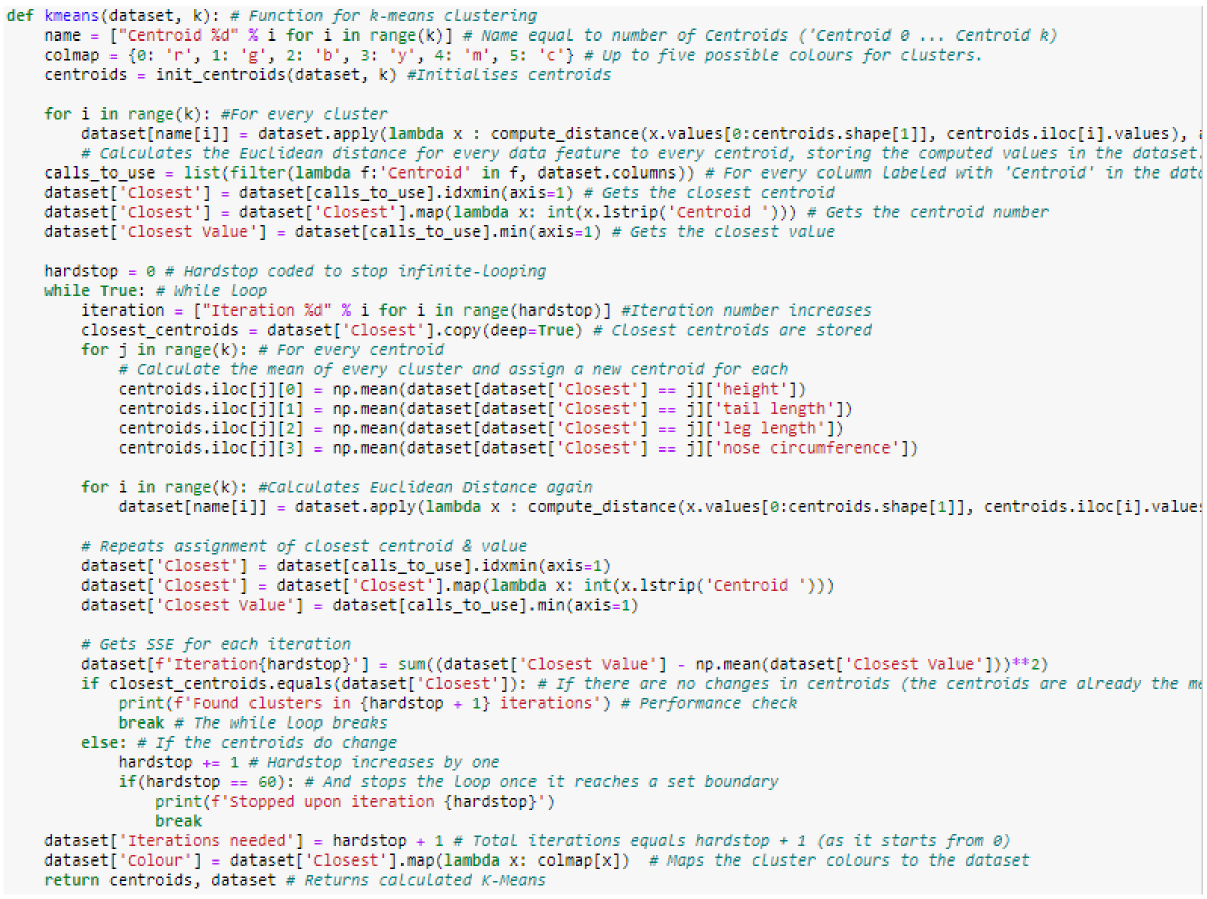


Figure 29 – Error of K-Means Clustering (3 Centroids)

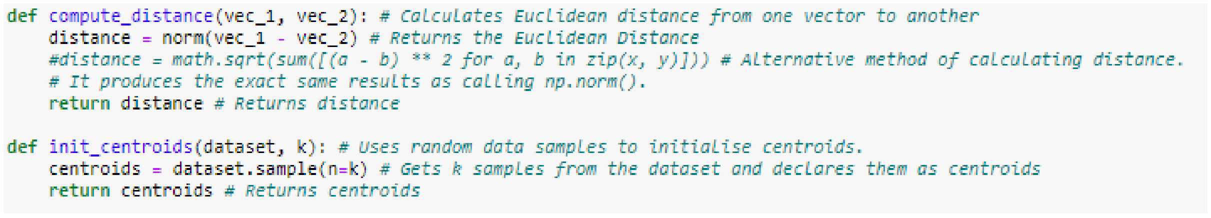
Each graph was constructed based off the values formulated from this function definition (figure 30):



```
def kmeans(dataset, k): # Function for k-means clustering
    name = ["Centroid %d" % i for i in range(k)] # Name equal to number of Centroids ('Centroid 0 ... Centroid k)
    colmap = {0: 'r', 1: 'g', 2: 'b', 3: 'y', 4: 'm', 5: 'c'} # Up to five possible colours for clusters.
    centroids = init_centroids(dataset, k) #Initialises centroids

    for i in range(k): #For every cluster
        dataset[name[i]] = dataset.apply(lambda x : compute_distance(x.values[0:centroids.shape[1]], centroids.iloc[i].values),
        # Calculates the Euclidean distance for every data feature to every centroid, storing the computed values in the dataset.
    calls_to_use = list(filter(lambda f:'Centroid' in f, dataset.columns)) # For every column labeled with 'Centroid' in the dat
    dataset['Closest'] = dataset[calls_to_use].idxmin(axis=1) # Gets the closest centroid
    dataset['Closest'] = dataset['Closest'].map(lambda x: int(x.lstrip('Centroid '))) # Gets the centroid number
    dataset['Closest Value'] = dataset[calls_to_use].min(axis=1) # Gets the closest value

    hardstop = 0 # Hardstop coded to stop infinite-looping
    while True: # While loop
        iteration = ["Iteration %d" % i for i in range(hardstop)] #Iteration number increases
        closest_centroids = dataset['Closest'].copy(deep=True) # Closest centroids are stored
        for j in range(k): # For every centroid
            # Calculate the mean of every cluster and assign a new centroid for each
            centroids.iloc[j][0] = np.mean(dataset[dataset['Closest'] == j]['height'])
            centroids.iloc[j][1] = np.mean(dataset[dataset['Closest'] == j]['tail length'])
            centroids.iloc[j][2] = np.mean(dataset[dataset['Closest'] == j]['leg length'])
            centroids.iloc[j][3] = np.mean(dataset[dataset['Closest'] == j]['nose circumference'])

        for i in range(k): #Calculates Euclidean Distance again
            dataset[name[i]] = dataset.apply(lambda x : compute_distance(x.values[0:centroids.shape[1]], centroids.iloc[i].value

        # Repeats assignment of closest centroid & value
        dataset['Closest'] = dataset[calls_to_use].idxmin(axis=1)
        dataset['Closest'] = dataset['Closest'].map(lambda x: int(x.lstrip('Centroid ')))
        dataset['Closest Value'] = dataset[calls_to_use].min(axis=1)

        # Gets SSE for each iteration
        dataset[f'Iteration{hardstop}'] = sum((dataset['Closest Value'] - np.mean(dataset['Closest Value']))**2)
        if closest_centroids.equals(dataset['Closest']): # If there are no changes in centroids (the centroids are already the m
            print(f'Found clusters in {hardstop + 1} iterations') # Performance check
            break # The while loop breaks
        else: # If the centroids do change
            hardstop += 1 # Hardstop increases by one
            if(hardstop == 60): # And stops the loop once it reaches a set boundary
                print(f'Stopped upon iteration {hardstop}')
                break
    dataset['Iterations needed'] = hardstop + 1 # Total iterations equals hardstop + 1 (as it starts from 0)
    dataset['Colour'] = dataset['Closest'].map(lambda x: colmap[x])  # Maps the cluster colours to the dataset
    return centroids, dataset # Returns calculated K-Means
```

Figure 30 – K-Means code

Using Panda Dataframes to increase computational speed, this implementation of K-means manages to find the same clusters in approximately 3 to 10 iterations. The two functions below are indicative of the two functions called in the one prior – ‘calculate_distance’ and ‘init_centroids’:



```
def compute_distance(vec_1, vec_2): # Calculates Euclidean distance from one vector to another
    distance = norm(vec_1 - vec_2) # Returns the Euclidean Distance
    #distance = math.sqrt(sum([(a - b) ** 2 for a, b in zip(x, y)])) # Alternative method of calculating distance.
    # It produces the exact same results as calling np.norm().
    return distance # Returns distance

def init_centroids(dataset, k): # Uses random data samples to initialise centroids.
    centroids = dataset.sample(n=k) # Gets k samples from the dataset and declares them as centroids
    return centroids # Returns centroids
```

Figure 31 – Euclidean Distance and centroid initialisation

From figures x to x it is clear to see that whilst this implementation of the K-means clustering algorithm is successful, it does lack when configuring clusters in more randomly spread out data. Once again, this may likely be due to the dimensions calculation required. Despite this however, the two graphs showing the overall error function both show a decreasing trend, proving the overall effectiveness of the algorithm.

However, this is not to say that the implementation of (simple) K-means is perfect; the function that initialises the centroids takes random samples from the input dataset, a method known as the Forgy method of initialisation. There is discourse as to whether or not this method – the random initialisation of centroids – is a truly effective methods, with some studies suggesting that there are far more effective methods of centroid initialisation (Bradley et al, 1998). Despite this debate however, the Forgy initialisation has proven to be suitable for this case scenario, regardless of its potential ranking amongst other methods.

Regarding the data presented on the graphs itself, the following information can be extrapolated from the clustered data:

Clustering the height and tail length tends to provide clusters in columns, suggesting that tail length is roughly linked to height and vice versa. Alongside this however, I do think that height and tail length is an inappropriate guideline for the x and y axis of a graph representing this data. As the centroid count increases, the clusters become more disjointed, spreading into one-another, indicating that despite whatever correlation they may have, height and tail length do not have a strong enough correlation to have a verifiable impact on producing quantifiable clusters of high quality.

Height and leg length are clearly linked to one-another – each cluster is tightly bound and separate from every other cluster. This is especially prevalent when calculating this with three centroids, as it still separates the smaller cluster towards the bottom left of the graph away from the other two centroids.

With the prior data and the error graphs presented, I believe the most suitable number of centroids – and thus clusters – for the required data is three. However, I would also estimate that a more suitable number, outside the bounds of this assignment, would be four. The increasing trend of efficiency from two to three centroids is easy to see on the error graphs, and four centroids would match the dimensions of the data (one for each feature).

References

Bradley, P.S. and Fayyad, U.M., 1998, July. Refining initial points for k-means clustering. In *ICML* (Vol. 98, pp. 91-99).

De la Garza, A., 1954. Spacing of information in polynomial regression. *The Annals of Mathematical Statistics*, pp.123-130.

Demiriz, A., Bennett, K.P. and Bradley, P.S., 2008. Using assignment constraints to avoid empty clusters in k-means clustering. *Constrained clustering: advances in algorithms, theory, and applications*, p.201.

Hartigan, J.A. and Wong, M.A., 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, *28*(1), pp.100-108.

"polynomial." *Merriam-Webster.com*. 2011. https://www.merriam-webster.com (22 November 2020).

Miller, S.J., 2006. The method of least squares. *Mathematics Department Brown University*, *8*, pp.1-7.

Oflazer, K., 1988. Partitioning in parallel processing of production systems.

Ostertagová, E., 2012. Modelling using polynomial regression. *Procedia Engineering*, *48*, pp.500-506.

Roy, P.P., Leonard, J.T. and Roy, K., 2008. Exploring the impact of size of training sets for the development of predictive QSAR models. *Chemometrics and Intelligent Laboratory Systems*, *90*(1), pp.31-42. __

Watson, G.S., 1967. Linear least squares regression. *The Annals of Mathematical Statistics*, *38*(6), pp.1679-1699. __

Yoshinaga, N. and Kitsuregawa, M., 2009, August. Polynomial to linear: Efficient classification with conjunctive features. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing* (pp. 1542-1551).