Machine Learning: Assessment 1

Task 1

Section 1.1 Description of Polynomial Regression

In the context of statistics, Linear Regression **"**quantifies the relationship between one or more predictor variables and one outcome variable" (Bock, 2019) For example, by plotting data on a scatter plot in x and y we could visually draw a straight line of best fit through our data points and then use this to either predict interpolated values within the range of our data set or extrapolate outside of the known range.

Linear regression models can be simple, where there is one predictor variable, or we may have multiple linear regression where more than one variable predicts an outcome (Murphy, 2012, 241).

Given a line drawn through some data points, if we want to know the success of that line in representing our data we firstly have to calculate the sum of residuals, or the "difference between the observed value of the dependant variable and the predicted value" (Stat Trek, 2019) If the relationship between x and y is linear, the most representative line is the one with the lowest sum of residuals. This is a simple example of an *error function*, which aims to evaluate how well the line represents the data. Taking this one step further, we could square the residuals (errors), to exaggerate for outliers, before summing them to produce the Sum of Squared Errors (SSE).

If we have a quantifiable way to test if a line represents the data, we can work backwards to discover from the data the best model. In linear regression, we are dealing with a straight-line equation of the form:

$$y = \beta x + \alpha$$

Where β is the gradient of the line and α the y-axis intercept. With the metric SSE we can determine whether our line is the 'best fit', by proving it has the lowest SSE of all possible lines, this is called the *Least Square Solution*. To find the intercept, α:

$$\alpha = \bar{y} - \hat{\beta}\bar{x}$$

(Rogers and Girolami, 2017, 5 - 7)

$\bar{y}$ is the mean of y and similarly, $\bar{x}$ is the mean of x. The gradient, β, can be found using:

$$S_x = \sum_{i=1}^{n} (x_i - \bar{x})$$

$$S_y = \sum_{i=1}^{n} (y_i - \bar{y})$$

$$\beta = \frac{S_{xy}}{S^2{}_x} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

S_x and S_y are the sum deviations from the mean of x and y respectively

Unfortunately, not all relationships between predictor and dependant variables are linear. A polynomial regression is similar but instead of a straight line, we aim to fit a polynomial of order k:

$$y = \beta_k x^k + \beta_{k-1} x^{k-1} + \cdots + \beta_1 x + \beta_0$$

We can express this equation in the form of the following matrices, where n is the sample size:

$$\begin{bmatrix} n & \sum {x_i}^1 \dots & \sum {x_i}^k \\ \sum {x_i}^1 & \ddots & \vdots \\ \sum {x_i}^k & \dots & \sum {x_i}^{2k} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_k \end{bmatrix} = \begin{bmatrix} \sum \alpha_i x^0 \\ \sum \alpha_i x^1 \\ \sum \alpha_i x^k \end{bmatrix}$$

By computing the values in the left- and right-hand matrices, we are left with k simultaneous equations which can be solved by any regular method to find the values of β_o, β_1, etc.

When creating a regression model, the usual method is to create the model using training data and to then test it with your testing data. Both datasets come from the same superset and should be randomly selected (Marsland, 2015, 21). The performance of the model for the testing and training should be similar if it is to work successfully. If it works better for the training data, then it is probably over trained/ over fitted.

Section 1.2: Implementation of Polynomial Regression

Below are three snippets which summarise the implementation; the code is commented for clarity. The full code can be found in the file Task1-2&1-3.py, which is supplied as supplementary material.

```
1.  # Takes vector input and sums elements to-
    gether, power can be used to square or cube elements before addi-
    tion but is 1 by default
2.  def Sum(data, power=1):
3.      total = 0
4.      for item in data:
5.          total = total + (item ** power)
6.      return total
7.
8.  # Takes two vector inputs and calculates dot product (a1*b1+a2*b2+…), op-
    tional power can be used to square or cube x component but is 1 by default
9.  def dotProduct(xData, yData, power=1):
10.     total = 0
11.     for x, y in zip(xData, yData):
12.         total = total + (y * (x ** power))
13.     return total
14.
15. # passed the weights (coefficents of x in an array) and the max order of the poly-
    nomial, this function returns the formula for the line as a string
16. def genFormula(weights, order):
17.     formula = '' # initialise as empty string
18.     for i in range(order, -1, -1): # runs from the order down to -1 in steps of -
    1 (e.g. if order were 5, values of i would be: 5,4,3,2,1,0)
19.         formula = formula + str(weights[i]) + '*x**' + str(i) + '+' # append for-
    mula with current coefficent next to x to the power of i
20.     formula = formula[:-1] # remove last char of formula which is an addi-
    tional '+' symbol
21.     return formula
```

```
1.  LHS Matrix Example:
2.  #   for data set x = [1, 2, 3, 4, 5], and a polynomial of order 3
3.  #   first row of LHS matrix:
4.  #       [n=5, sum(x) = 15, sum(x^2) = 55]
5.  #   second row:
6.  #       [sum(x)=15, sum(x^2) = 55, sum(x^3)]
7.  #   third row
8.  #       [sum(x^2), sum(x^3), sum(x^4)]
9.
10.
11. # calculates everything but the first value on the first row
12. def LHSFirstRow(xData, order):
13.     row = np.matrix([]) # initialise matrix
14.     for i in range(1, order + 1): # iterates i from 1 up to the order of polynomial
     in steps of 1
15.         row = np.hstack([row, np.matrix([Sum(xData, i)])]) # each value in row is s
    um of xData^ iterator i
16.     return row
17.
18. # calculates one row of the matrix
19. def LHSRow(xData, startPower, endPower):
20.     row = np.matrix([])
21.     for i in range(startPower, endPower + 1): # iterates from a starting point to a
     max order for that row
22.         row = np.hstack([row,  np.matrix([Sum(xData, i)])]) # each value in row is
    sum of xData ^ iterator i
23.     return row
24.
25.
26. def getLHS(xData, yData, order):
27.     myMatrix = np.matrix([xData.size]) # first value in matrix is size of sample (n
    umber of x values)
28.     myMatrix = np.hstack([myMatrix, LHSFirstRow(xData, order)]) # calculates remain
    der of first row, first value is sum of x values, next is sum of x values squared,
    then cubed etc. up to x^ the order of the polynomial
29.
30.     for i in range(1, order + 1): # iterates from 1 to order of polynomial in steps
     of 1
31.         # calculates and appends a row to matrix where starting power of x is one m
    ore than previous
32.         myMatrix = np.vstack([myMatrix, LHSRow(xData, i, i + order)])
33.     return myMatrix
34.
35.
36. def getRHS(xData, yData, order):
37.     myMatrix = np.array([Sum(yData)]) # initialise matrix and calculate first value
     as sum of y values * x^0
38.     #calculate next values in matrix as dot product of y values and x values ^i, wh
    ere i iterates from 1 to order of polynomial
39.     for i in range(order):
40.         myMatrix = np.hstack(
41.             (myMatrix, np.array([dotProduct(xData, yData, i + 1)])))
42.     return myMatrix
```

```
1.  def pol_regression(features_train, y_train, degree):
2.      RHS = getRHS(features_train, y_train, degree) # produce the matrix on the right
     hand side (RHS) of equation
3.      LHS = getLHS(features_train, y_train, degree) # priduce matrix on left hand sid
    e (LHS) of equation
4.      parameters = np.linalg.solve(LHS, RHS) # solve simultaneous equations, returns
    matrix of x coefficients
5.      return parameters
```

The following figures show the output of the program for polynomials of differing degrees. As well as plotting the dataset (in blue), the line of regression is plotted (in orange). The equation of the line is also given.

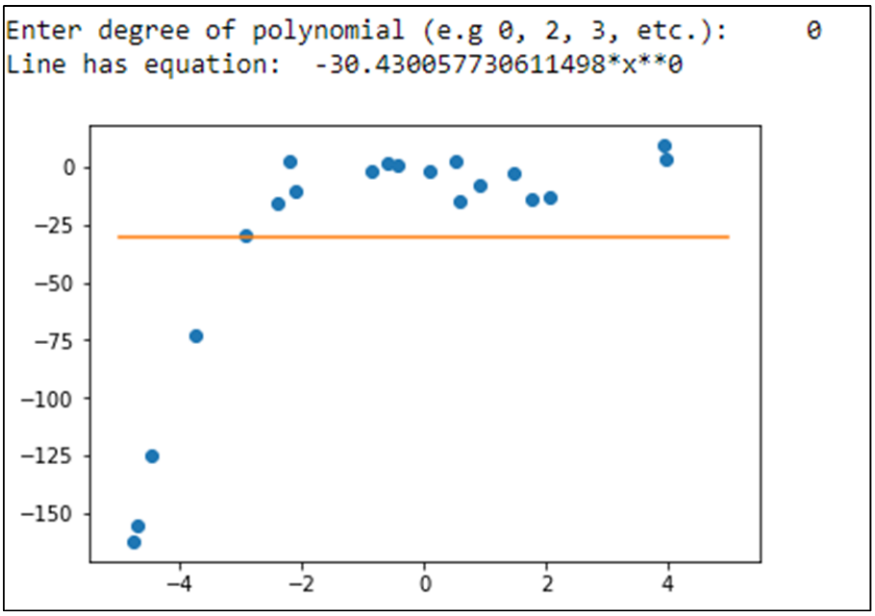

Figure 1: Polynomial of degree 0

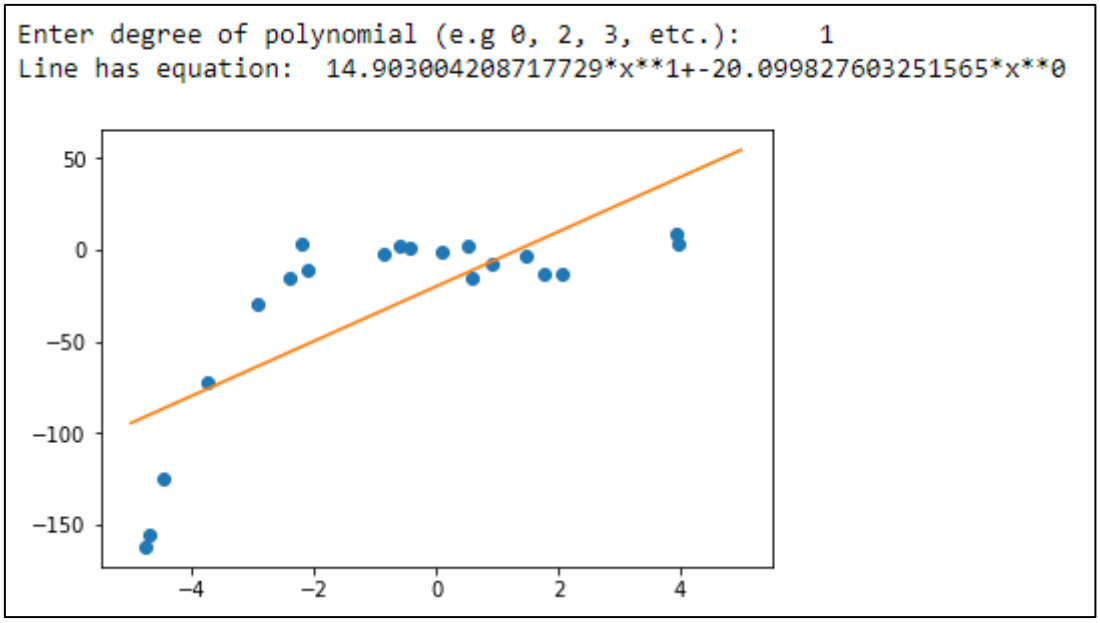

Figure 2: Linear (Degree = 1)

```
Enter degree of polynomial (e.g 0, 2, 3, etc.):     2
Line has equation:  -3.9823831749875955*x**2+9.900252504762516*x**1+5.514103999
5524554*x**0
```

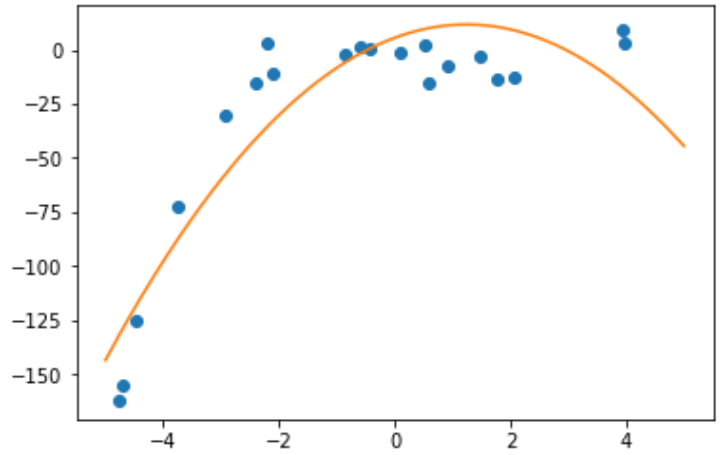

Figure 3: Quadratic (Degree = 2)

```
Enter degree of polynomial (e.g 0, 2, 3, etc.):     3
Line has equation:  1.2518352059012001*x**3+-2.601682458641687*x**2+-7.48952995
9228205*x**1+0.44240766994442465*x**0
```

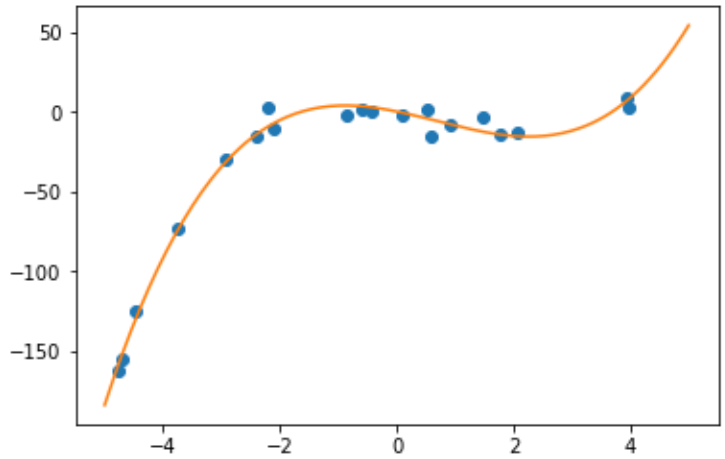

Figure 4: Cubic (Degree = 3)

```
Enter degree of polynomial (e.g 0, 2, 3, etc.):     5
Line has equation:  0.003224122149450766*x**5+-0.06919795280562006*x**4+1.04953
2243493842*x**3+-1.294209984416221*x**2+-5.749451240181667*x**1+-1.872020996151
3856*x**0
```

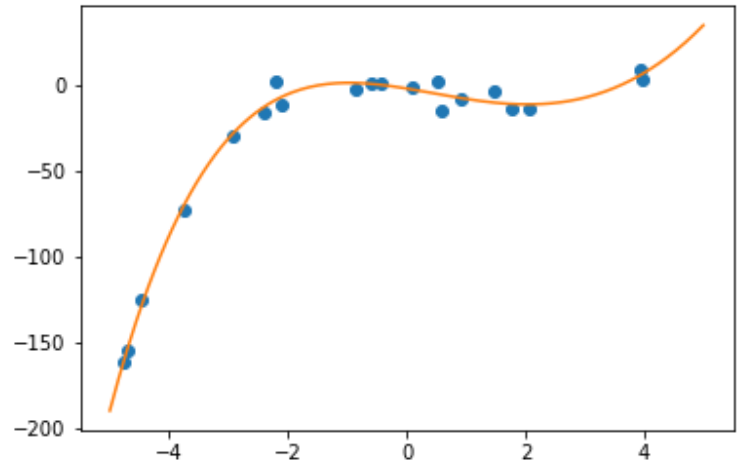

Figure 5: Polynomial of degree 5

```
Enter degree of polynomial (e.g 0, 2, 3, etc.):     10
Line has equation:  -0.0004131126370894956*x**10+-0.0024741095594911604*x**9+
0.008479577941898752*x**8+0.06775326709427026*x**7+0.0002580357506333465 4*x**6+
-0.5102998072123621*x**5+-0.720617308428018*x**4+1.9607042100269467*x**3+1.0712
053150227168*x**2+-5.496509753820017*x**1+-2.886915154354406*x**0
```

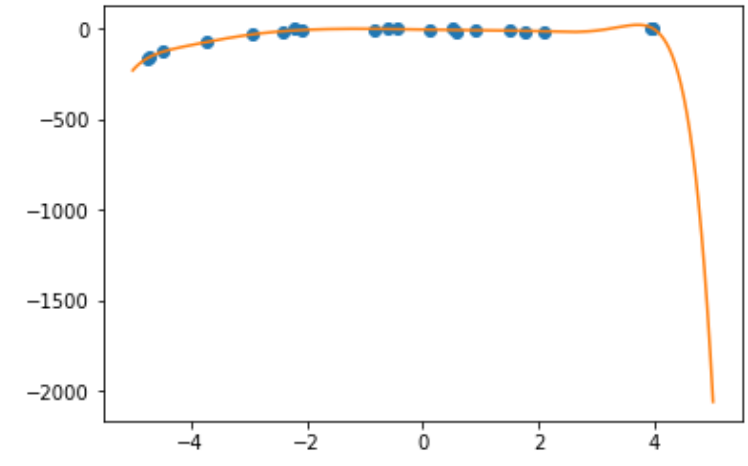

Figure 6: Polynomial of degree 10

Looking at figures 1 to 3, I would say they show signs of underfitting, the line does pass roughly through ‘busy’ areas with lots of data points, but it does not follow the curve of the points. On the other hand, Figure 6, with a polynomial of degree 10, is far overfitted and is jagged as it jumps to try and meet every data point. I think, were we to extrapolate beyond x=4 for this graph, we would not get accurate predictions.

Figures 4 and 5 show a good fit for the data, the line follows the data's general trend without jumping wildly to meet a single outlying point. Looking at the lines, I would say that either of these polynomials is an acceptable fit for the data.

Section 1.3: Evaluation

Below is the code used to evaluate the polynomial regression function described in section 1.2. The evaluation is done by calculating the Root Mean Square Error (RMSE). This is the square root of the mean of the errors. The errors, in this case, being the residuals, or the "difference between the observed value of the dependant variable and the predicted value" (Stat Trek, 2019). The equation for RMSE can be written as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2}$$

Where N is the sample size, Y the value calculated by the formula and $\hat{Y}$ the value of the actual data point (Murphy, 2012, 979). This will give a good indication of the ability for our line of regression to interpolate data points by evaluating the RMSE for the test data. We should also be able to see the RMSE reducing as the fit of our training data increases with the degree of the polynomial.

```
1.  # Below function calculates Root Mean Square Error (RMSE) - error = differ-
    ence in y from what line predicted to actual data point
2.  # yi = the sum of the y values as calculated by formula
3.  # to get Mean Squared Error (MSE), do (yi - actual y values)^2
4.  # and to get RMSE = sqrt(MSE)
5.  #
6.  # RMSE Example:
7.  #   formula = 2x + 4
8.  #   xData = [1, 2, 3, 4, 5]
9.  #   yData = [7, 8, 8, 15, 14]
10. #
11. #   yi, when i is 1 = (2(1) + 4) = 6
12. #   actual y at i is 1 = 7
13. #   (yi - y)^2 = (6 - 7)^2 = -1^2 = 1
14. #
15. #   Repeat above for i = 1,2,3,4,5 and add together the squared differences:
16. #       total diff = 114
17. #
18. #   calculate the MSE:
19. #       114 / number of data items (5) = 22.8
20. #   and sqrt to find final RMSE:
21. #       sqrt(22.8) = 4.77
22. def eval_pol_regression(weights, xData, yData, degree):
23.     mse = 0 # initilase mean square error as 0
24.     for x, y in zip(xData, yData): # for each item in data sets x and y
25.         yi = 0
26.         for i in range(degree, -1, -1): # runs from the order down to -
    1 in steps of -1 (e.g. if order were 5, values of i would be: 5,4,3,2,1,0)
27.             yi = yi + (weights[i] * (x ** i)) # sum up the weight * x^i to find the
     value of a given yi
28.         mse = mse + ((yi - y) ** 2) # get the difference between lines predic-
    tion (yi) and the actual data point (y), and square it to get squared error
29.     mse = mse / len(xData) # calcualte mean squared error by dividing to-
    tal squared error by number of data points
30.     rmse = math.sqrt(mse) # find RMSE from MSE by square rooting
31.     return rmse
```

```
1.  # Runs code for evaluation task
2.  def Task1_3():
3.      fig = plt.figure()
4.      dataset = pd.read_csv('Task1Data.csv') # read data from csv file
5.      #split data into x and y
6.      x_data = dataset['x'].as_matrix()
7.      y_data = dataset['y'].as_matrix()
8.
9.      # height of data = number of data items - needed to work out num items to creat
    e 70:30 split between training and test sets
10.     height = len(x_data)
11.     # training data is first 70% of total dataset
12.     x_dataTrain = x_data[0:math.ceil(height * 0.7)]
13.     y_dataTrain = y_data[0:math.ceil(height * 0.7)]
14.
15.     # Testing data is last 30% of total dataset
16.     x_dataTest = x_data[0:math.floor(height * 0.3)]
17.     y_dataTest = y_data[0:math.floor(height * 0.3)]
18.
19.     # loop just runs from 0 to 5 as we want to test polynomials with degrees 0 to 5
     as well as 10, which is below this loop
20.     for i in range(0, 5):
21.         #Run polynomial regression with training data
22.         weight = pol_regression(x_dataTrain, y_dataTrain, i)
23.
24.         #calculate the RMSE for the training data
25.         rmse = eval_pol_regression(weight, x_dataTrain, y_dataTrain, i)
26.         #plot RMSE in blue
27.         plotPoint(i, rmse, 'blue')
28.
29.         # calculate RMSE for test data
30.         rmse = eval_pol_regression(weight, x_dataTest, y_dataTest, i)
31.         # plot RMSE in red
32.         plotPoint(i, rmse, 'red')
33.
34.     #Run polynomial regression with training data
35.     weight = pol_regression(x_dataTrain, y_dataTrain, 10)
36.
37.     #calculate the RMSE for the training data
38.     rmse = eval_pol_regression(weight, x_dataTrain, y_dataTrain, 10)
39.     plotPoint(i, rmse, 'blue')
40.
41.     # calculate RMSE for test data
42.     rmse = eval_pol_regression(weight, x_dataTest, y_dataTest, 10)
43.     plotPoint(i, rmse, 'red')
44.
45.     # Put labels on axes & print label to id training and test data by data point c
    olour
46.     ax1 = fig.add_subplot()
47.     ax1.set_ylabel('RMSE')
48.     ax2 = fig.add_subplot()
49.     ax2.set_xlabel('Degree of Polynomial')
50.     print('\nRed points are the test set, blue are training set')
```

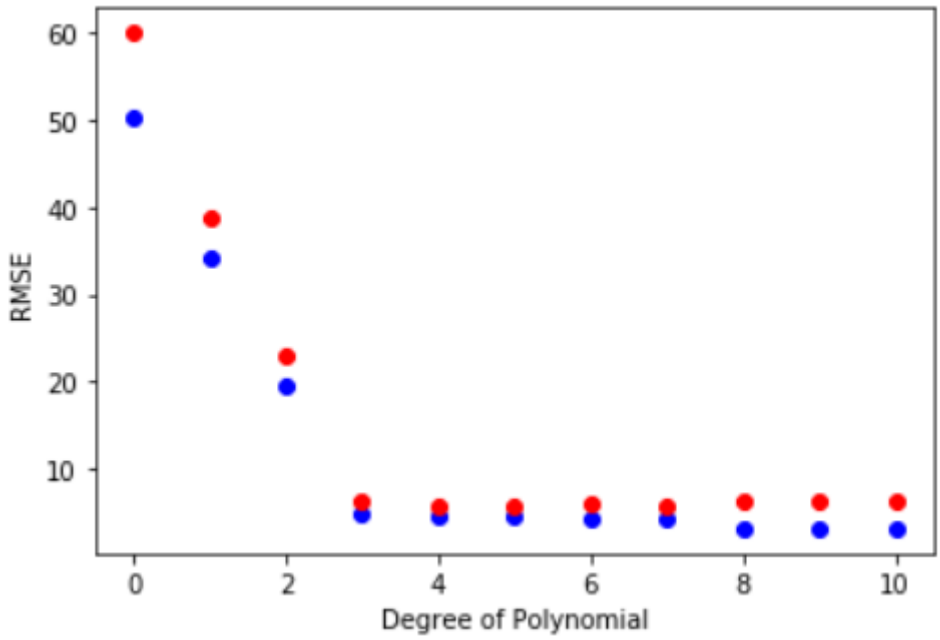

Figure 7: Plot of the RMSE for polynomials of degree 0 to 10 for test data (Red) and training (Blue)

The graph indicates that, for the training data, degree of polynomial and RMSE are inversely proportional, this is as expected due to the line being a better and better fit as the degree increases. It can also be noted that the improvement in fit (reduction in RMSE) between 0 and 3 is marked compared to the improvement between 3 and 10, suggesting a logarithmic relationship.
For the test data we also saw this inversely proportional relationship up until a degree of 4 or 5, at which point the RMSE begins to increase again. This is likely due to the polynomial becoming overfit at degrees greater than 5, meaning that despite it being better representative of the training data, it does not represent the super set of data.

Degree of Polynomial	Training RMSE	Test RMSE
0	50.271255356938084	60.04102228715721
1	34.05230128376551	38.83255102478253
2	19.38269937878986	23.05294460600515
3	4.819672477328903	6.323133163897483
4	4.409928496762635	5.622521712104138
5	4.391935190131512	5.710688141353777
6	4.230807071482502	6.101161771774809
7	4.094627419051673	5.791058161927237
8	3.187860542362442	6.188716361210541
9	3.168691867088409	6.171749765485259
10	3.1007675294997026	6.1369271740768045

Table 1: RMSE values for test and training data for polynomials with degrees 0 to 10

Looking at the specific values in Table 1, the test RMSE is lowest at a degree of 4, meaning that line is probably best.

Task 2

Section 2.1: Description of the K-Means Clustering

"The aim of cluster analysis is to create a grouping of objects such that objects within a group are similar and objects in different groups are not similar" (Rogers and Girolami, 2017, 205). For a given set of data, if we are to plot it in Euclidean space, we can see which points are 'similar' by their visual proximity. Mathematically, we could find this using Pythagoras in two dimensions, or in a multi-dimensional space we use:

$$\Delta(x_{ij}, x_{i'j}) = (x_{ij} - x_{i'j})^2$$

(Murphy, 2012, 876)

K-Means clustering is an unsupervised method and as such we need to formally describe a cluster as a "representative point" or centroid, which is the "mean of the objects that are assigned to the cluster" (Rogers and Girolami, 2017, 207). Each object is assigned as belonging to the cluster with which the Euclidean distance to the centroid is least. The objective function is to minimize the sum distances to the centroids, this can be done by iteratively running cycles were the centroids are recalculated to the mean position of each cluster at the end of a cycle.

The question is, for a completely unsupervised system, how can the centroids be defined? In this case they are initially created with random positions within the range of the dataset. All that needs to be supplied, therefore, is the number of centroids.

Given the number of centroids, k, the algorithm is as follows:

1. Create k centroids with random positions
2. For each object, find the closest centroid and assign that object to the centroid's corresponding cluster
3. For each cluster, re-calculate the centroid as the mean of the object's locations
4. If the centroid's positions have not changed, halt. If they have changed, return to 2

Choosing a value for K is a "common problem in cluster analysis, (…) as K increases, large clusters will be broken down into smaller and smaller parts" (Rogers and Girolami, 2017, 208) the smaller the clusters become the closer to the centroids and so the less useful the clustering is to tell us useful information.

K-means does sometimes fail to cluster objects in ways that would seem obvious to us. If the objects do not "conform to our current notion of similarity (distance)" (Rogers and Girolami, 2017, 205) it will fail to cluster correctly. For example, a dataset consisting of two consecutive rings of data, given a value of k = 2, will have both centroids placed inside the inner ring of data. This result ignores the obvious pattern we see in the data because it is simply designed to place the centroids at the mean location. This can sometimes be resolved by altering the view of the data to a different dimension.

Section 2.2: Implementation of the K-Means Clustering

```
# calculates euclidean distance between vectors 1 and 2 which can have any number of dimmensions (as long as they both have same number)
def compute_euclidean_distance(vec_1, vec_2):
    distance = 0 # initialise distance between points as 0
    for a, b in zip(vec_1, vec_2): # for each element of vector 1 and 2
        distance = distance + ((a - b) ** 2) # difference squared added to total distance
    distance = math.sqrt(distance) # root of distance gives final answer
    return distance
```

```
# creates k points with coordinates in the range of the data set passed
def initialise_centroids(dataset, k=2):
    rangeMin = math.floor(np.amin(dataset)) # finds the lowest value in the data (and rounds down to int)
    rangeMax = math.ceil(np.amax(dataset)) # finds max val in data (and rounds up to int)
    centroids = np.array([random.randint(rangeMin, rangeMax), random.randint(
        rangeMin, rangeMax), random.randint(rangeMin, rangeMax), random.randint(rangeMin, rangeMax)]) # initialise at least one centroid as a 4D vector, with random values for each in the range established above
    # loop to create and append as many other centroids are needed to match k
    for i in range(k - 1):
        centroids = np.vstack([centroids, np.array([random.randint(rangeMin, rangeMax), random.randint(
            rangeMin, rangeMax), random.randint(rangeMin, rangeMax), random.randint(rangeMin, rangeMax)])])# create centroid as a 4D vector, with random values for each in the range established above and append to matrix
    return centroids
```

```
1.  def kmeans(dataset, k=2):
2.      sumDistances = [] # keeps track of the objective and is appended each cy-
    cle with the current sum distances from centroids
3.      centroids = initialise_centroids(dataset, k) # creates k centroids in space oc-
    upied by the dataset
4.      assignmentsOld = np.empty(dataset.shape[0], dtype=int) # initialise array
5.      Exit = False # while loop will continue to loop until this flag is true
6.      while Exit == False:
7.          assignmentsNew = runCycle(dataset, centroids) # runs a cycle and as-
    signs all of the points to a centroid
8.          if np.array_equal(assignmentsOld, assign-
    mentsNew): # if no points have been assigned to a different centroid then the clus-
    tering is complete and the program can exit
9.              Exit = True
10.         else:
11.             assignmentsOld = assignmentsNew # if not, store the new assign-
    ments to check against after the next iteration
12.         for item in centroids: # for each centroid
13.            if np.count_nonzero(item) == 0: #if the centroid = [0, 0, 0, 0]
14.                 return kmeans(dataset, k) # re-run the process be-
    cause there has been an error in the initial assignments
15.         centroids = reCalcCentroids(dataset, assignmentsNew, k) #  re-calcu-
    late the centroids to be the mean of their cluster
16.         sumDistances.append(objFunction(dataset, centroids, assignmentsOld)) # ap-
    pend the array with the sum of the distance of all points to their assigned cen-
    troid - this is graphed later to show the progress of the clustering
17.     cluster_assigned = assignmentsNew # return the current assignments as the fi-
    nal clusters
18.     plotLine(sumDistances) # make a plot of the objective function over each itera-
    tion
19.     return [centroids, cluster_assigned]
```

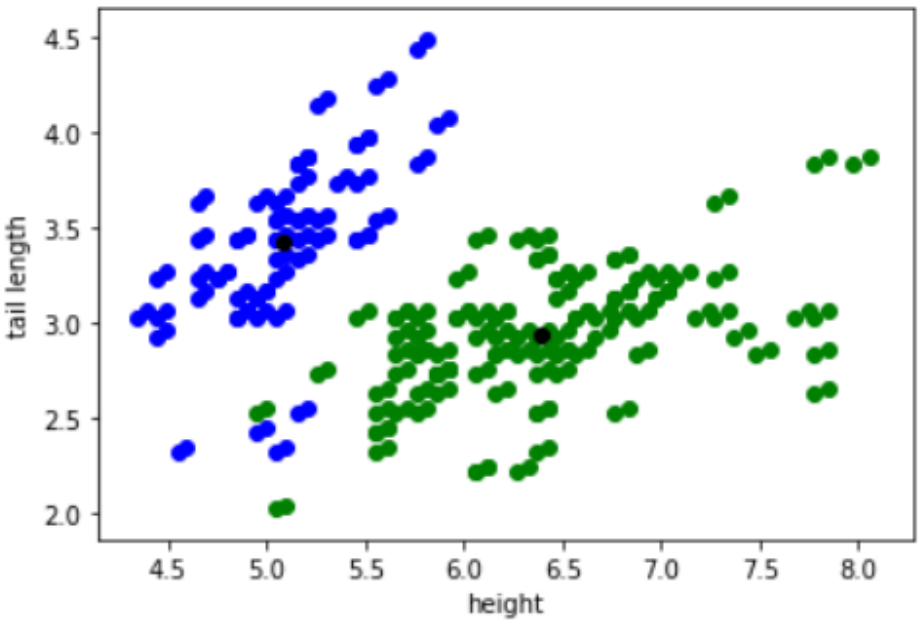

Figure 8: K=2, centroids are marked in black

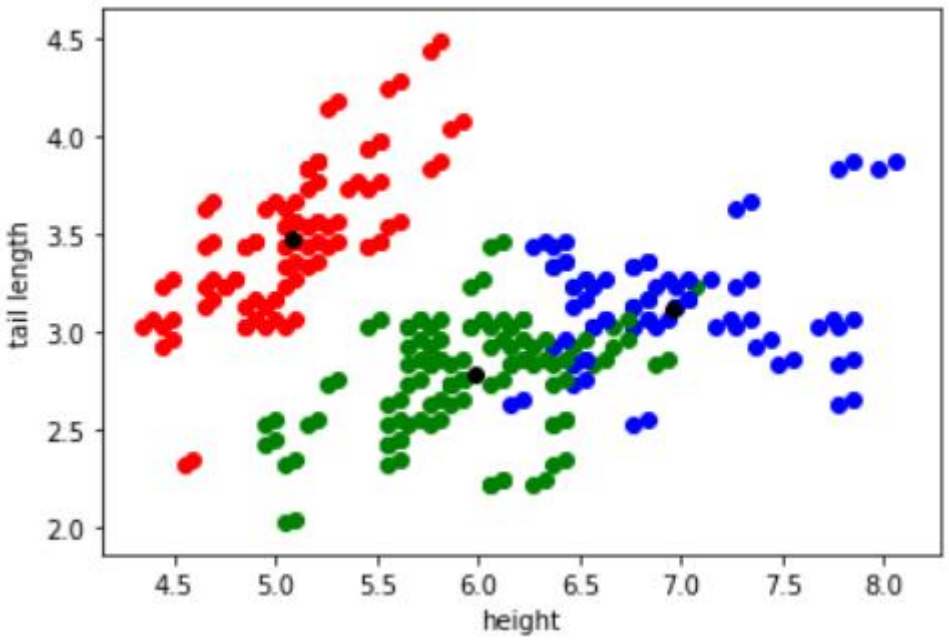

Figure 9: K=3, centroids are marked in black

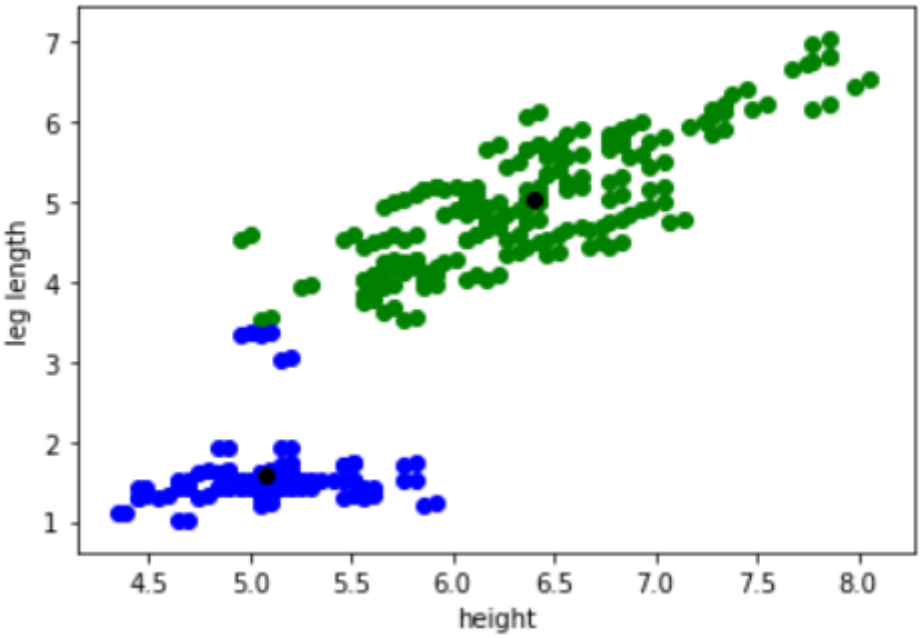

Figure 10: K=2, centroids are marked in black

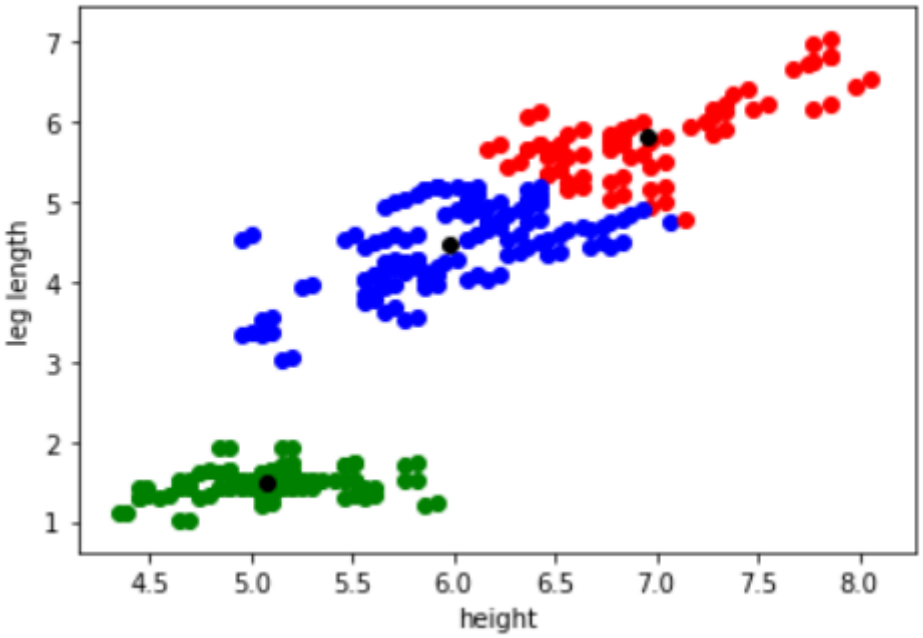

Figure 11: K=3, centroids are marked in black

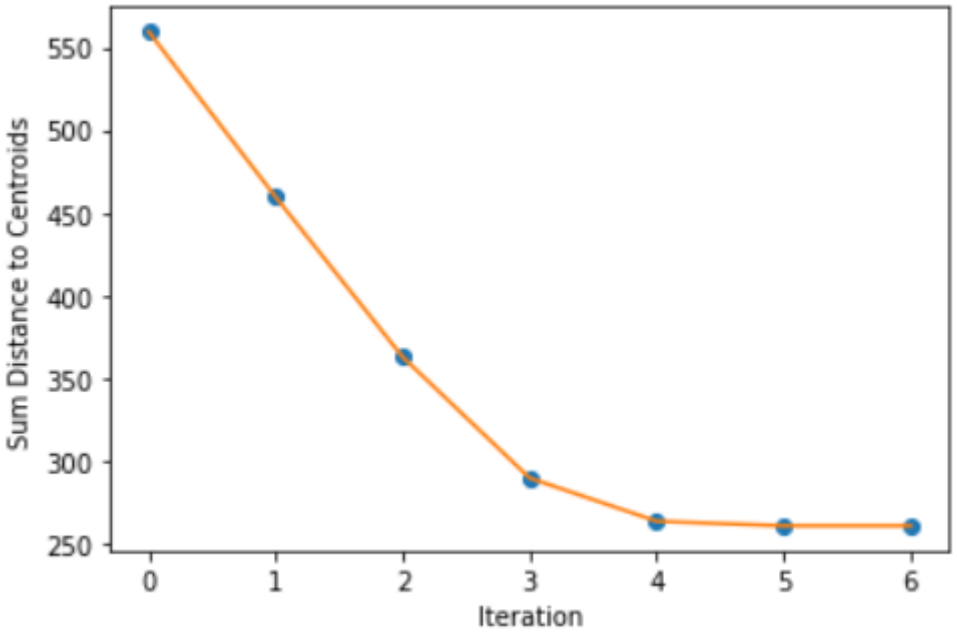

Figure 12: Iteration plotted against objective function for K = 2

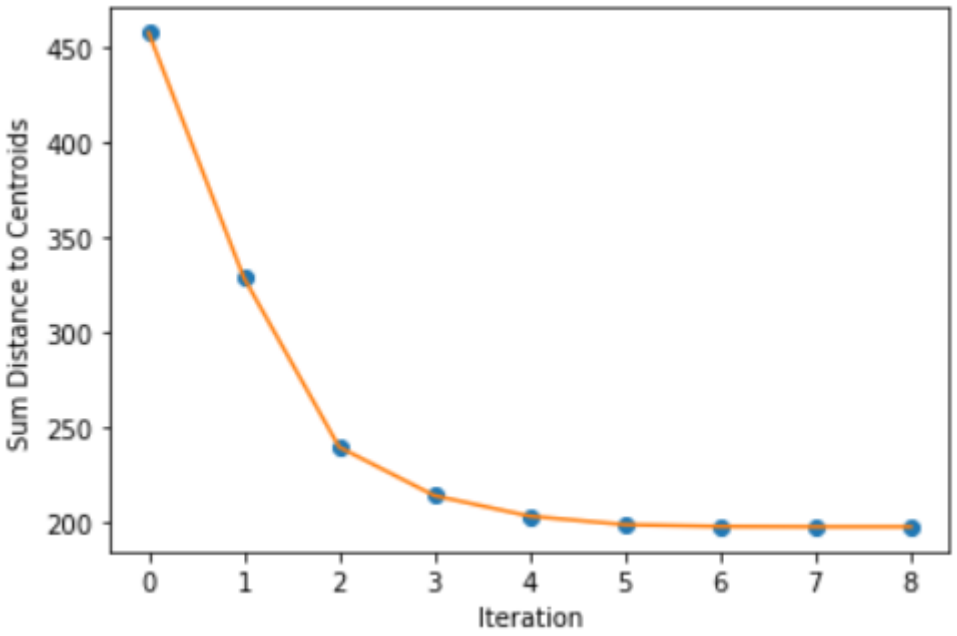

Figure 13: Iteration plotted against objective function for K = 2

Figures 8 – 11 of the program seems to show that it runs successfully and categorises the data into k clusters. It also seems, objectively speaking, that the clusters are made up of geographically neighbouring data points. This is not true of all the points, but because we have a 2D view of a higher dimensional data set it isn't possible to visually tell from these figures if the algorithm is successful.

Figure 12 and 13 seems to indicate that the code is successful as the objective function decreases in a logarithmic fashion and the convergence for K = 3 is lower than for K = 2. This would be the expected output as for each iteration the clusters are improved, and the objective function reduced. The lower convergence for k = 3 is due to there being more, smaller clusters and "the smaller each cluster is, the closer each point will get to its cluster mean" (Rogers. S and Girolami. M, 2017, 208). Figure 14 is an example of a very high K value producing tangled clusters. It's recommended to consider the context of the data in selecting an appropriate value for K, to make the clustering of most use.

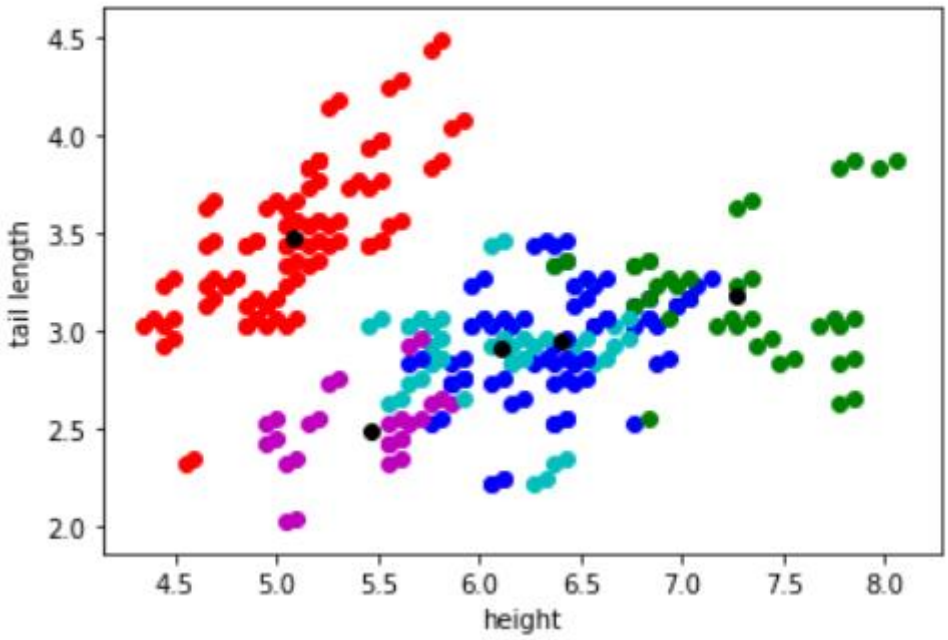

Figure 14: Example of too high a K value (K=5)

References

1. Rogers, S. Girolami, M. (2017) *A First Course in Machine Learning*. USA: Taylor & Francis Group
2. Murphy, K. (2012). *Machine Learning: A Probabilistic Perspective*. USA: MIT Press
3. Stat Web (2000) *rms Error.* USA: Stanford University. Available from http://statweb.stanford.edu/~susan/courses/s60/split/node60.html [Accessed 22 November 2019]
4. Stat Trek (2019) *Residual: Definition*. Available from https://stattrek.com/statistics/dictionary.aspx?definition=residual [Accessed 23 November]
5. Bock, T. *What is Linear Regression.* Displayr.com. Available from https://www.displayr.com/what-is-linear-regression/*)* [Accessed 23 November]
6. Marsland, S. (2015) *Machine Learning: An Algorithmic Perspective, 2nd Edition*. Florida, USA: Taylor & Francis Group, LLC