Machine Learning Assignment 1 Benjamin Terrill

Table of Contents

Section 1.1 Description of Polynomial Regression (16%):

Regression is often used to predict the values of something in the future based on the data that is available. For example, it can be used to try and predict the stock market price based on current price and the conditions of the market, or predict the temperature based on weather conditions and the location.

Linear regression is the most used type of modelling. It determines the relationship between a dependant variable (Y) and independent variables (X) using a line of regression which is also known as a line of best fit. Simple linear regression uses just one independent variable and multiple linear regression will use multiple independent variables on the X-axis. It uses the equation Y = a + b*X, where Y is a dependant variable and X is the independent variable.

Finding the line of best fit for the values given can be done using different methods. First you need to find the Sum of Squared Errors (SSE), this is most commonly done using the Least Square Estimation Method. This method minimizes the sum of errors from the data points on the line. This is important because outliers have a big effect on the regression line and the predicted values. SSE follows the equation: $SSE = \sum_{i=1}^{n} (x_i - \bar{x})^2$

There are different types of regression models including; quadratic function models and cubic function models. Both of these are types of polynomial regression that turn a linear regression model into a curve. The reason for this is that the X value is squared or cubed rather than the coefficient.

The first model I will talk about here is the Quadratic Regression Model. This model follows the equation; y = ax^2 + bx + c . Quadratic regression requires more data points to show the full "U" shape that will be made with this model when compared to linear regression. Once the model is made, it can be tested to see if it is a good fit for the data by determining R-Squared. This tells you how much variation in y (dependant variable) can be explained by the relationship by x (independent variable). The closer to 0 means that it is not a good fit and y and x have little or no correlation. Whereas, if the number is closer to 1, this means that it is a good fit and there is a direct correlation between the two variables.

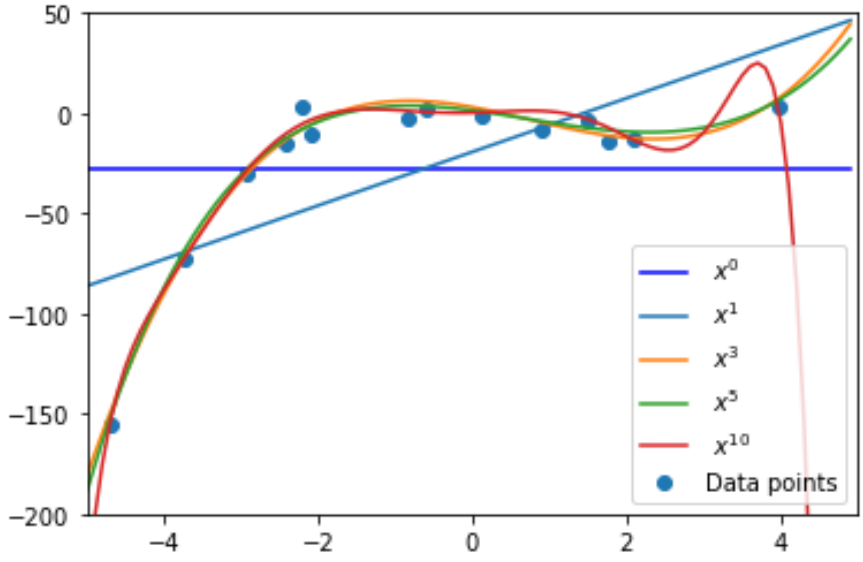
The next model to talk about it the Cubic Regression Model. The equation for this is; y = a + bx + cx^2 + dx^3 . Cubic regression is more useful if the data being used will have more deviation that causes the line of best fit the curve up and down.

Polynomial feature expansion is an important tool that can be used to get a better model for the data. This is done by creating a feature in which each variable in X is raised to the power of degrees - 1.

Polynomial regression is used when the power of the independent variable is >1. In this type of regression the line of best fit will curve rather than being a straight line. It fits the data points given more closely. It uses two datasets, Training and testing. Training dataset is the sample of data that is used to fit the model and test dataset is used to evaluate the final model fit on the training data. This is important in finding out the Sum of squared errors, Mean of squared errors and Root mean of squared errors. These are the metrics that are used to identify how accurate the regression model is, if the RMSE is smaller that means that the model is more accurate and a higher number means it is not accurate.

Section 1.2 Implementation of Polynomial Regression (20%):

Firstly, below is a graph of the data plotted on a graph and clear line for polynomial regression using degrees 0,1,2,3,5,10.



From the graph I can clearly see some under and over fitting. Using degree 0 and 1 creates a line that is extremely under fitted and they are almost useless for predicting results.

Degree 3 is much better and the line shows the regression well, this is shown especially toward the end where it continues upwards after the final data points and after the dip. This is important because it allows for better predictions of future data.

Degrees 5 is the best of the choices here because it is neither under or over fitted and follows the correct trend after reaching the final data points. Because of this it is much more useful for predicting the future data.

Degree 10 show signs of overfitting as the regression lines goes off drastically towards the end because it is being massively over fitted, this makes the predictions useless because they drop straight down.

Because of the overfitting when using higher degrees, x^5 is the most suitable out of the ones on the graph because it continues on a path that would be more accurate for predictions.

The first step to getting the polynomial regression output is to create a variable that contains the data from the dataset. In this case “dataset_train”.

From there I created two variablea that contain training and test data. Split up 70% and 30%. Then I plot the data on the graph and assign variables to hold “X” and “y” respectively for each dataset.

```
def main():

    dataset_train = pd.read_csv('task1_dataset.csv')
    #dataset_test = pd.read_csv('tast1_dataset.csv')
    #print(dataset_train)
    #print(dataset_test)

    dataset_train_copy = dataset_train.copy()

    dataset_train2 = dataset_train_copy.sample(frac = 0.70)
    # train dataset gets 70% of the data.
    print("test train", dataset_train2)
    dataset_test2 = dataset_train_copy.drop(dataset_train2.index)
    # test dataset will get the remaining 30%.
    print("test test", dataset_test2)

    train_pct_index = int(0.7 * len(dataset_train['x']))
    train_x, test_x = dataset_train['x'][:train_pct_index], dataset_train['x'][train_pct_index:]
    train_y, test_y = dataset_train['y'][:train_pct_index], dataset_train['y'][train_pct_index:]
    print("test", train_x)

    plt.clf()
    plt.scatter(dataset_train2['x'], dataset_train2['y'])
    # Plot data on graph
```

To the right is a screenshot of my pol_regression function. I pass it test x,y and train x,y as well as the degree that is being used.

I perform numPy operations om the data to get both the weights and a stacked matrix that can be used.

This need to be done so that the shape of the matrix is usable alongside the dataset matrix to be plotted.

The returned value (parameters) contains the values needed to get the Y-axis for the regression model based on the degree given.

```
def pol_regression(train_x, train_y, test_x, test_y, degree):
    #print("train_x pre steps:", train_x)
    X = np.ones(train_x.shape)
    #print("np ones of train_x: ", X)
    for i in range(1, degree + 1):
        X = np.column_stack((X, train_x ** i))
    #print("np column stack: ", X)

    XX = X.transpose().dot(X)
    #print("XX", XX)

    w = np.linalg.solve(XX, X.transpose().dot(train_y))
    # Get the weights for polynomial fit.
    #print("w: ", w )

    Xtest1 = np.ones(test_x.shape)
    for i in range(1, degree+1):
        Xtest1 = np.column_stack((Xtest1, test_x ** i))
    #print("Xtest1: ", Xtest1)

    parameters = Xtest1.dot(w)
    #print("Ytest1: ". Ytest1)
    return parameters
```

Finally, this section of code is where the pol_regression function is called and the regression model is plotted on the graph.

The first part of the code gets an x value that allows me to plot the line of best fit along every part of the x axis rather than just plotting on data points.

I then get an average value for one. This is to substitute as the degree 0 line that cannot be done using the function because it is less than 1.

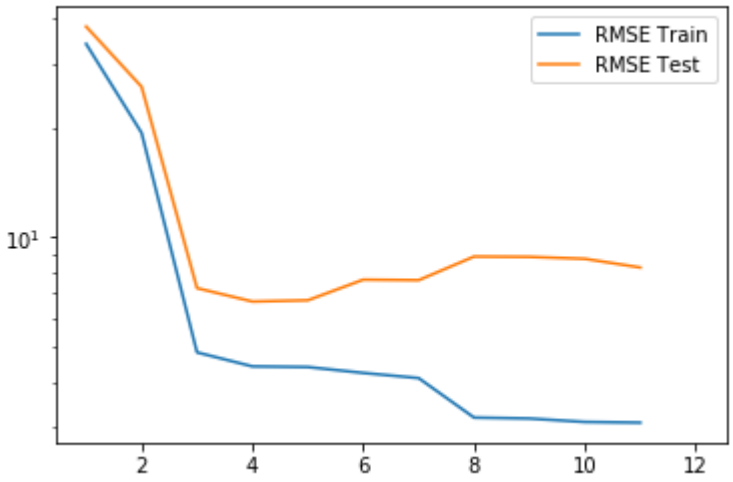
I then get a y value for each degree that is being used from the pol_regression function and plot it on the graph.

```
x = np.arange(-5, 5, 0.1)
print("Matrix: " , x)
print("Matrix-x: " , test_x)
#This is used to plot along the entire x axis.
#Stops the graph from plotting only at training X asix points.

avg = np.average(train_y) # Get average value of y
print("Average: ", avg)
y = []
for i in range(len(x)):
    y.append(avg) # creating a y value for degree 0
plt.plot(x, y, 'blue')


y_value1= pol_regression(train_x, train_y, x, test_y, 1)
plt.plot(x, y_value1)
#plot the line for degree 1

y_value3= pol_regression(train_x, train_y, x, test_y, 3)
plt.plot(x, y_value3)

y_value5= pol_regression(train_x, train_y, x, test_y, 5)
plt.plot(x, y_value5)

y_value10 = pol_regression(train_x, train_y, x, test_y, 10)
plt.plot(x, y_value10)

plt.ylim((-200, 50))
# limit the graph to -200 and 50 along y axis
plt.xlim((-5, 5))
# limit graph to -5,5 along the x axis
plt.legend((' $x^{0}$', ' $x^{1}$', ' $x^{3}$', ' $x^{5}$',' $x^{10}$', 'Data points')
```

At the bottom you can see that I limited the Y axis to -200 to 50 and limited the X axis to -5 and 5. This is here to stop the polynomial regression model exceeding the limits and distorting the data.

Section 1.3 Evaluation (14%):

Below if the graph showing the RMSE of both the training data and the test data as well as the arrays containing the values.




```
rmsetrain:
[[ 34.05230128]
 [ 19.38269938]
 [  4.81967248]
 [  4.4099285 ]
 [  4.39193519]
 [  4.23080707]
 [  4.09462742]
 [  3.18786054]
 [  3.16869187]
 [  3.10076753]
 [  3.08703108]
 [  0.        ]]
rmsetest:
[[ 38.02275151]
 [ 25.96717203]
 [  7.24215974]
 [  6.65264951]
 [  6.70538099]
 [  7.64206759]
 [  7.61723478]
 [  8.84816736]
 [  8.83151736]
 [  8.72259011]
 [  8.25468508]
 [  0.        ]]
```

From the graph and the data we can see that the training error decreases as the polynomial degree changes. The test data also decreases but starts to go up again as the polynomial degree increases. This is typical from data that is over fitted.

The optimal model should be chosen based on the test data and not the training data. This is because the training data is simply remembered by the algorithm and not learnt how to generate new samples.

It should be noted that the test data has a sharp drop of rather than a gradual one. This could be that there is an erroneous data point that is throwing off the algorithm at the end.

To the right side is a screenshot of the function used to evaluate the polynomial regression. It uses training and test data to get the rmse of both.

It does this by using matrix functions to create usable arrays for the x and y coordinates.

I also use w to get a weight. The same way I did in previous task.

Then I get the mean difference between the training points squared. This is repeated for the test data.

```
def eval_pol_regression(parameters, trainx, trainy, testx, testy, degree):
    #SSEtrain = np.zeros((11,1))
    #SSEtest = np.zeros((11,1))

    trainX = np.ones(trainx.shape)
    for i in range(1, degree + 1):
        trainX = np.column_stack((trainX, trainx ** i))

    testX = np.ones(testy.shape)
    for i in range(1, degree + 1):
        testX = np.column_stack((testX, testx ** i))


    X = np.ones(trainx.shape)
    for i in range(1, degree + 1):
        X = np.column_stack((X, trainx ** i))

    XX = X.transpose().dot(X)
    w = np.linalg.solve(XX, X.transpose().dot(trainy))
    #for i in range (1, 10):
    SSEtrain = np.mean((trainX.dot(w) - trainy)**2)
    SSEtest = np.mean((testX.dot(w) - testy)**2)
    rmseTrain = math.sqrt(SSEtrain)
    rmseTest = math.sqrt(SSEtest)
    return rmseTrain, rmseTest
```

I then use the math.sqrt function to get the RMSE of each and return them.

```
rmsetrain = np.zeros((14,1))
rmsetest = np.zeros((14,1))

for i in range(1, 14):
        rmsetrain[i - 1], rmsetest[i - 1] = eval_pol_regression(y_value1, train_x, train_y,

print("rmsetrain: ")
print(rmsetrain)
print("rmsetest: ")
print(rmsetest)

plt.figure();
plt.semilogy(range(-5,9), rmsetrain)
plt.semilogy(range(-5,9), rmsetest)
plt.legend(('RMSE Train', 'RMSE Test'), loc = 'lower right')
```

I then create to numPy arrays called rmsetrain and rmsetest. These arrays will store the rmse value for each polynomial degree.

I then use a for loop to call the function for each degree, and store it in the array.

Final step is to plot the graph with the rmse data to create a useful visual of the test and training data.

Section 2.1 Description of the K-Means Clustering (20%):

K-means clustering is a type of unsupervised machine learning algorithm. It is one of the most popular of these and is a relatively simple algorithm, designed to split up a given dataset into a fixed number of clusters. Below are the steps needed to complete K-means clustering.

Steps:

- First plot the data on a scatter graph.
- Generate Centroids with random positions that go onto the graph.
- Then calculate the Euclidean distance between each data point and each centroid.
- Assign each data point to its nearest centroid.
- Each centroid is given a cluster of its nearest data points.
- Get the mean distance from each cluster and update to create a new centroid at that location.
- Repeat until the centroids have stabilized (i.e. no change in centroid values because the clustering has been successful) or the defined number of iterations has been achieved.

Centroids are the first part of the algorithm. A centroid is a data point placed into the original cluster (normally at random). K number of centroids are made and need to be unique.

Once the centroids are made and placed into the cluster, the Euclidean distance (or straight line distance) need to be calculated between each data point and each cluster. This is done to find out which centroid is the nearest to each data point in the dataset. Once you have the Euclidean distance for each data point, you need to assign the points to its nearest centroid. This then creates a cluster of data points assigned to the centroid. If there are 3 centroids then the first assignment will create 3 clusters.

The next step is the update step, this is where the mean location of the data points in a cluster are calculated and a new centroid is placed there. From here the assignment step is then repeated assigning each data point to its new closest centroid.

This then repeats until the centroids have stabilized and cannot be changed again using the algorithm, or if the pre-defined number of iterations have been completed.

Advantages

- K-means is a relatively simple algorithm to implement.
- It is able to scale well with large data sets.
- Can choose the starting position of centroids if chosen.
- Guarantees convergence

Disadvantages

- The algorithm is dependent on the initial values for the centroids, this can be a problem for large data sets. This can be mitigated by choosing specific values rather than randomising.
- K-means doesn’t work well when data is of different sizes.

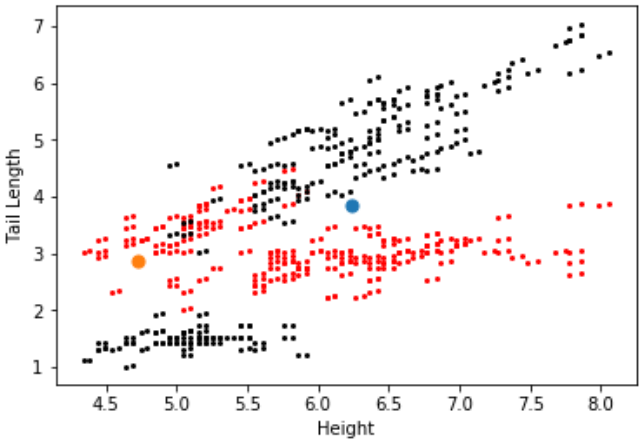
- Outliers in the dataset can affect the centroids and drag them out towards the edge of the dataset, or they might get their own cluster if they are particularly far away. Because of this it is better if their data points are removed before running the algorithm.

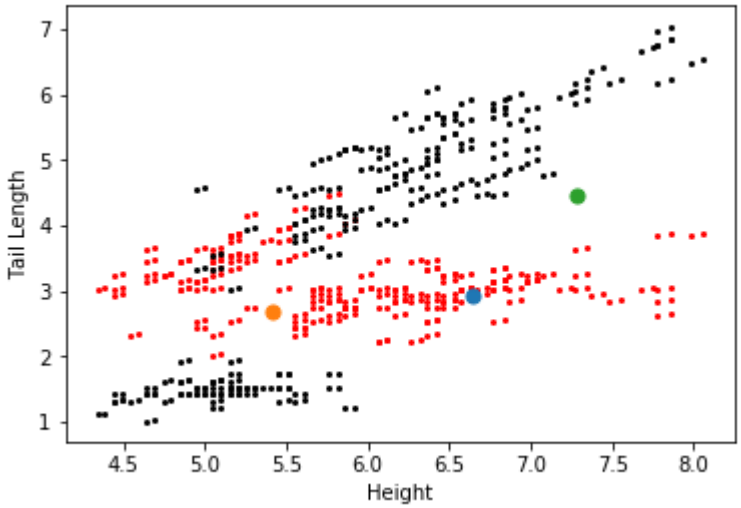
Section 2.2 Implementation of the K-Means Clustering (30%):

Shown below is two scatter graphs showing the subplot for “height”, “tail length” and “height”, “leg length”. The first shown in red and the latter in black.

In the first, K=2 so there are two randomly placed centroids coloured orange and blue.

In the second scatter graph, K=3 so there are three randomly placed centroids coloured orange blue and green.






```
Centroids:  [(5.091766908850456, 2.2515506635580333), (4.769994041494647, 4.073111376860515), (6.834715912868285,
2.9967737752004284)]
Final Centroids:  [(5.60282278481, 2.6395443038), (5.04014141414, 3.4674040404), (6.701584, 3.033288)]
```

The screenshot above shows the final centroids after running the K-means algorithm with the given starting centroids displayed just above.

I was able to complete the algorithm and get valid results. However, I was unable to plot the updated centroids on the scatter graph without some errors. Therefore, I cannot display a visual representation of the k-Means working.

K-Means Code:

Here I will go through my code for K-Means Clustering and how it works.

First thing I did was create a class called 'Point'. Here I created some functions, one to calculate the Euclidean distance and I created x and y which will hold positions of data points.

Dist will be used to calculate the distance between each data point and the centroid

```
26 class Point:
27
28     def __init__(self, x, y):
29         self.x = x
30         self.y = y
31         self.cid = -1
32     def dist(self, point2):
33         return np.sqrt(np.power(self.x-point2.x,2)+np.power(self.y-point2.y,2))
34     def compute_euclidean_distance(vec_1, vec_2):
35         distance = np.sqrt((vec_1 - vec_2) **2 + (vec_1 - vec_2)  **2)
36         return distance
37     def __repr__(self):
38         return "(" + str(self.x) + ", " + str(self.y) + ")"
39     def __str__(self):
40         return "(" + self.x + ", " + self.y + ")"
41
```

```
47 def initialise_centroids(pointset, k):
48     #minX = min(pointset,key=operator.attrgetter('x')).x
49     #minY = min(pointset,key=operator.attrgetter('y')).y
50     #maxX = max(pointset,key=operator.attrgetter('x')).x
51     #maxY = max(pointset,key=operator.attrgetter('y')).y
52
53     centroids = []
54     for i in range(k):
55         centroids.append(Point(np.random.uniform(4.5, 8), np.random.uniform(1, 7)))
56     return centroids
57
```

Next I created an Initialise centroids function which creates K amount of centroids and places them in random positions within the bounds of the dataset given.

Below is the getClusterAssignments function. This is used to get alongside compute_euclidean_distance to get the distance between each point in the pointset and the randomly placed centroids.

Once this is done each point has an ID that correlates to the cluster that it is being assigned to.

```
58 def getClusterAssignments(pointset, centroids):
59     k = len(centroids)
60
61     """make an empty array of arrays (a list of lists of points, each list of points are the points for a certain cluster)"""
62     cluster_assigned = []
63     for i in range(k):
64         cluster_assigned.append([]);
65
66     """for every point find its nearest centroid, then add to that cluster list"""
67     for p in pointset:
68         best = p.dist(centroids[0])
69         #best = compute_euclidean_distance(p, centroids[i])
70         bestId = 0
71         for i in range(k):
72             currentDistance = compute_euclidean_distance(p, centroids[i])
73             if currentDistance <= best:
74                 best = currentDistance
75                 bestId = i
76         cluster_assigned[bestId].append(p)
77     return cluster_assigned
78
```

K-Means is the most important function for this algorithm.

It takes in the parameters of pointset and K and returns the new centroids locations and each assigned cluster.

```
80 def kmeans(pointset, k):
81     if k<1:
82         return
83     changed = True
84     centroids = initialise_centroids(pointset, k)
85     print("Centroids: ",centroids)
86     while(changed):
87         changed = False
88         """cluster assignment"""
89         cluster_assigned = getClusterAssignments(pointset, centroids)
90 
91         for i in range(k):
92             avg = Point(0,0)
93             count = 1
94             for p in cluster_assigned[i]:
95                 avg.x += p.x
96                 avg.y += p.y
97                 count+=1
98             avg.x/=count
99             avg.y/=count
100            if compute_euclidean_distance(avg, centroids[i])>0:
101                changed=True
102            centroids[i]=avg
103    return centroids, cluster_assigned
104
```

This is where centroids are initialised and the update part of the program runs.

While changed = true, cluster assigned will call the function getClusterAssignments so we know which point is in which cluster.

Then I get the average of all those distances for each cluster in range K (2 or 3). Centroids is assigned the average and this is where the new centroids will be placed.

Finally, I have my main function which imports the dataset, sets K and assigns the pointset the values from the given dataset.

I also plot the data here.

```
105 def main():
106     dataset = pd.read_csv('task2_dataset.csv')
107     k = 3
108     pointset2 = [Point(2,3),Point(5,6),Point(2,9)]
109     pointset = []
110     for i in range (len(dataset.height-1)):
111         #print("i", i)
112         pointset.append(Point(dataset.height[i], dataset.tail_length[i]))
113 
114     #print("P2", pointset2)
115     #print("p1", pointset)
116     centroids, cluster_assigned = kmeans(pointset, k)
117     plt.scatter(dataset.height, dataset.tail_length, color = 'orange', s = 4)
118     plt.scatter(dataset.height, dataset.leg_length, color = 'blue', s = 4)
119     plt.xlabel('Height')
120     plt.ylabel('Tail Length')
121 
122     #print("centroids", centroids)
123     #for i in range (k):
124     #    plt.scatter(centroids[i],centroids[i] , s = 50)
125 
126 
127 
128     #print("Pointset: ", pointset)
129     print("Final Centroids: " ,centroids)
130     #print("Clustered Points: " , cluster_assigned)
131 
132 
133 if __name__ == "__main__":
134     main()
135 
```

Unfortunately, I ran into errors when plotting the centroids so while the positions can be displayed they do not appear on the scatter graph as intended.

However, from testing the results the outputs given for the final location of the centroids and the cluster assignments is valid.