CMP3751M – Machine Learning

Assignment Item 1: Polynomial Regression & K-Means Clustering

Alex Howe – 15618835

Contents

Section 1: Polynomial Regression

1.1: Description

Polynomial Regression is a method of regression, and a variation on the Linear Regression model. This technique is used for fitting graphs which exhibit a polynomial or non-linear trend in their data, whereas Linear Regression is for use with linear trends.

As implied by its name, Linear Regression is one of the simplest forms of Machine Learning algorithm. It calculates, using a variety of different techniques, the weights for the equation $\hat{y} = b + wx$, where y is the predicted output value, b is the bias (also known as w_0 or y-intercept), w (also known as w_1) is the weight or gradient of the line.
For datasets with more than one feature variable, this equation can be extended to $\hat{y} = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$, where n is the total amount of features.
There are multiple methods for calculating these weights including using Gradient Descent and finding the Least Squares Estimation (LSE).
Gradient Descent is an iterative algorithm which calculates the total loss of all points from a line generated from random weight estimates. The derivative of the loss function is calculated, and the weights are iteratively changed in order the find the minimum point for this derivative.
However; Least Squares fitting is a much more mathematical approach. This uses matrix calculations to compute the values of the equation's weights. The general equation for Ordinary Least Squares, the most common formulation of LSE and the one which was used for this task, is $w = (X^TX)^{-1}X^Ty$ (Weisstein, n.d.), where y is a vector of all training output values and X is a two-dimensional Vandermonde matrix where each row consists of all features for a certain point, and each column contains the values for all data points for a given feature. X^T is simply the transpose of the X matrix, and w is the vector of weights applied to each feature. Applying the dot product with this weight vector and another Vandermonde matrix of feature values allows for the prediction of outputs. (Neutrium, n.d.)

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{bmatrix}$$

Figure 1: An example vandermonde Matrix (Weisstein, n.d.)

Although this algorithm is often useful in multivariate Linear Regression, it can also be applied to Polynomial Regression. Despite having quadratic, cubic or higher order polynomials in the output calculation, Polynomial Regression is considered a linear technique since the exponential component is based on the feature values, rather than the unknown values of the weights. The equation for calculating the outputs for a trained model using Polynomial Regression is $\hat{y} = w_0 + w_1x + w_2x^2 \cdots w_dx^d$. This still uses the same value of x for each term, but raises it to increasing powers, up to the degree d. For a degree of 1, the Polynomial Regression algorithm is identical to that of a Linear Regression model.

To evaluate the Polynomial Regression algorithm, a metric must be used to measure how accurate the model is to the actual dataset. For this assignment the Root Mean Squared Error was calculated. This method squares each error value, calculates the mean of these, and then takes the square root of this mean in order bring this value back in line with the initial error values. Alternative metrics include R-Squared and Mean Absolute Error.
In order for the error to be calculated, a separate dataset needs to be used to predict y values, that weren't used to train the model initially. Because of this, datasets are generally split into two sections for training and testing. The amount of testing data to use is usually between 15-30% of the entire dataset, and the rest is for training the model.

1.2: Implementation of Polynomial Regression

The "pol_regression" function takes arrays of both feature and output datapoints to train the model with, and finally the desired degree of the polynomial to be fitted. It generates a Vandermonde matrix using the feature values as the length and degree as the width, then uses this in $w = (X^T X)^{-1} X^T y$ to derive the vector of weights for the model. These are returned to the main section of code which allows points to be plotted.

```
def pol_regression(features_train, y_train, degree):
    xM = np.vander(features_train, degree+1, increasing=True) #creates a vandermonde matrix using x and the degree.
                                                              #the 'increasing' bool reverses the powers applied in the VM matrix

    xT = xM.transpose() #transposes the vandermonde matrix

    parameters = np.linalg.solve(xT.dot(xM), xT.dot(y_train)) #calculates the weights based on w=(xT*X)^-1*(xT*y)
                                                              #uses Linalg to calculate inverse as more stable
    return parameters
```

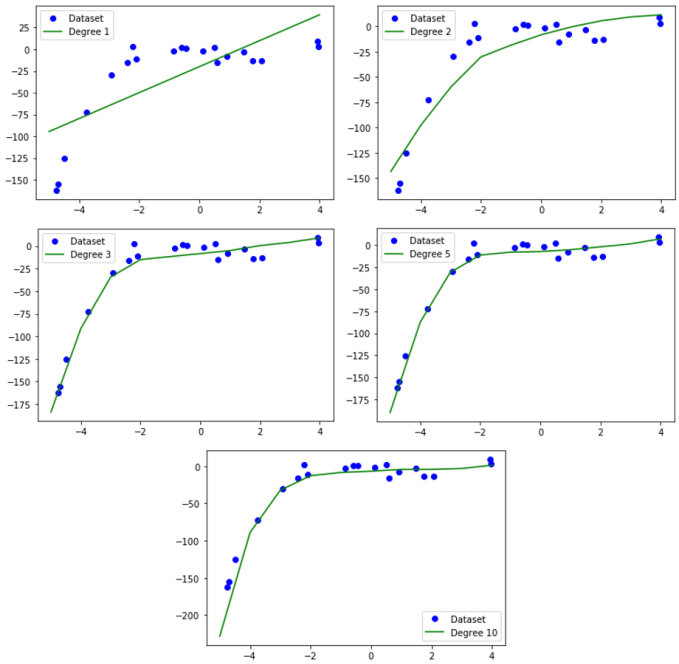
Figure 2: Implementation of the Polynomial Regression function

The next step is applying these weights to the training data in order to find the predicted output values, and to plot them against the actual dataset so that comparisons can be made.

```
x = np.vander(rangeArr, deg+1, increasing=True) # creates a VM matrix for the range of -5 to 5
y = x.dot(parameters) #gets the y values by multiplying the VM matrix  with the weights: y = Xa

plt.plot(features, output, 'bo') #plots dataset
plt.plot(rangeArr, sorted(y), 'g') #plots range of -5 to 5 with the respective y values
plt.legend(["Dataset", label])
plt.show()
```

Figure 3: Y values are predicted and plotted

After plotting graphs for several degree values and comparing them, I feel that a polynomial of degree 5 would be best suited for this dataset. The model created with a degree of one is linear – this is not an appropriate representation of the data as it clearly follows a nonlinear pattern. Degree 2 still looks heavily underfit for an estimation of the ground truth of the curve. The third degree is much better, however d=5 is the only graph that shows at a slight upwards curve towards the end of the graph, hinting that the data may in fact show a cubic curve. Degree 10 feels slightly overfit for the curve.



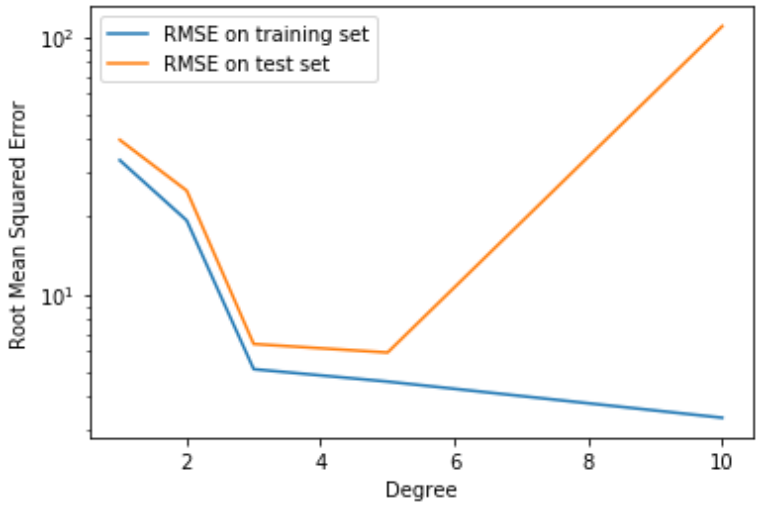
1.3: Evaluation

```
def eval_pol_regression(parameters, x, y, degree):
    xTrain = np.vander(x[0], degree+1, increasing=True) #creates two vandermonde matrices
    xTest = np.vander(x[1], degree+1, increasing=True)  #for both training and testing data

    rmseTrain = np.sqrt(np.mean((xTrain.dot(parameters) - y[0])**2)) #Calculates RMSE for the training data
    rmseTest = np.sqrt(np.mean((xTest.dot(parameters) - y[1])**2)) #RMSE for the testing data

    rmse = rmseTrain, rmseTest #combines both error arrays

    return rmse
```

This function calculates and returns the Root Mean Squared Error for both training and testing data. This is then added to an array in the main section of code, where the RMSE values are plotted against their respective degrees.

```
degrees = [1, 2, 3, 5, 10] #specifies which degrees for which to plot training & testing RMSE
rmse = np.zeros([len(degrees), 2]) #initialises an array for the RMSE of training and testing data
for i in range(len(degrees)): #iterates for each value in degree array
    parameters = pol_regression(features[0], output[0], degrees[i]) #determines weights for the current degree
    rmse[i] = eval_pol_regression(parameters, features, output, degrees[i]) #gets training and testing RMSE

plt.semilogy(degrees, rmse[:, 0]) #plot RMSE logarithmically against each degree
plt.semilogy(degrees, rmse[:, 1])
plt.legend(('RMSE on training set', 'RMSE on test set'))
plt.show()
```

Because the data is shuffled every time the application is executed, the graph created can vary, however generally looks as displayed below. This illustrates that, as the error is high for both training and testing data, the model is underfit for degrees one and two, confirming the estimation made in Section 1.2. Additionally, the training error continues to decrease after degree 5, while the RMSE for testing data rapidly increases. This indicates an element of overfitting for the higher degree polynomials.



Section 2: K-Means

2.1: Description

K-Means Clustering is a form of unsupervised data classification. Its goal is to separate a given dataset into k clusters. This has a variety of uses – mostly to find patterns in data that can seem almost random to the human eye.

The K-Means algorithm is relatively efficient and involves iterating multiple times over the dataset. It employs the use of "centroids": created points of data which are used to compare which data is closest. Centroids are originally chosen randomly from the input data, and once they have been initialised the algorithm iterates over every point in the dataset and uses the 'objective function' to find the Euclidean distance from each point to the k centroids in the dataset. Euclidean distance is the direct distance from one point to the other – in this case the relevant centroid and data point. It's calculated using the formula $\sqrt{(p_1 - c_1)^2 + (p_2 - c_2)^2 + \cdots + (p_d - c_d)^2}$, where p and c are the point's and centroid's vector positions respectively and d is equal to the total dimensions of the dataset. (Mathonline, n.d.)

Once this has been calculated for each centroid, the point is classed as 'belonging to' whichever has the lesser Euclidean distance. This occurs for every point, and once finished each point has been assigned a centroid. After this the centroids are recalculated, using the mean of all points assigned to the respective centroids.

The process then iterates, until the recalculated centroids are effectively identical to the previous set, at which point they have become 'stable'. The algorithm is now complete, and the points belonging to each centroids are each their own class. These can be visually identified and labelled, for example there may be a dataset of various metrics about a variety of dogs (such as height, tail length, etc.), and each cluster of points would be likely to belong to each breed of dog.

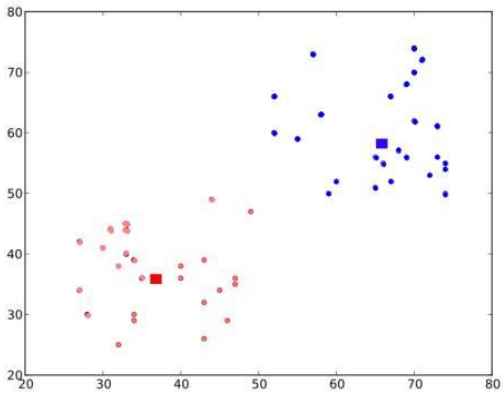
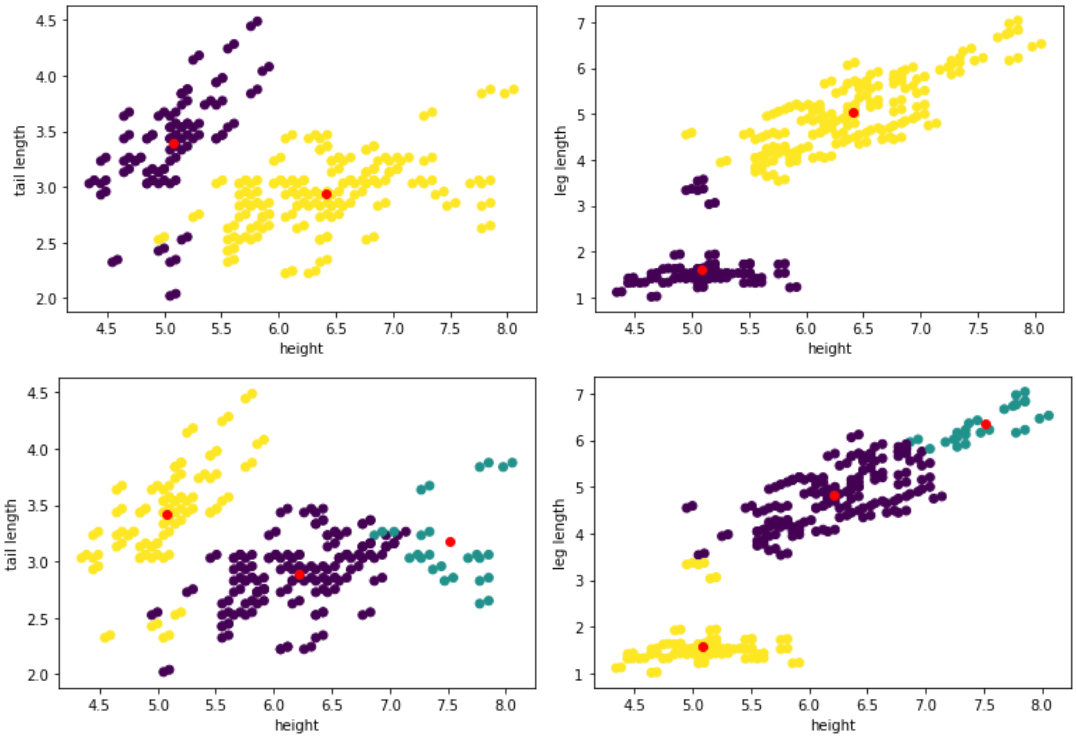


Figure 4: A small dataset clustered by K-Means (Mordvintsev, 2013)

One advantage of K-Means is that it's one of the simplest clustering algorithms to implement. As the new centroids are calculated based on the mean of all points, they have a guarantee of convergence. Another benefit of this algorithm is that it scales well with the length of the dataset – it's still relatively fast even when working with a large amount of points. However, a drawback of using this algorithm is that you have to choose the value of k manually in order to minimise error of classification. In addition, even with the best amount of clusters it may incorrectly categorise data if the pattern displayed does not rely on simply which points are closest, for example interlocking crescents (Kumar, 2018). It can also be very susceptible to outlier points.

2.2: Implementation of the K-Means Clustering

The following images show the given dataset with two and three clusters respectively. Where K = 2, the clusters are for the most part visually obvious, insofar as the majority of points in each cluster aren't near any points from other clusters. As this is a multivariate dataset with four feature values, the points that appear to be distant in the graphed axes may in fact be much closer in others, hence the categorization of these points.



For the plot of K = 3, the cluster marked in teal is much clearer and more separated. However, between the yellow and purple clusters there aren't many inherent visual clues to separate both of these clusters.

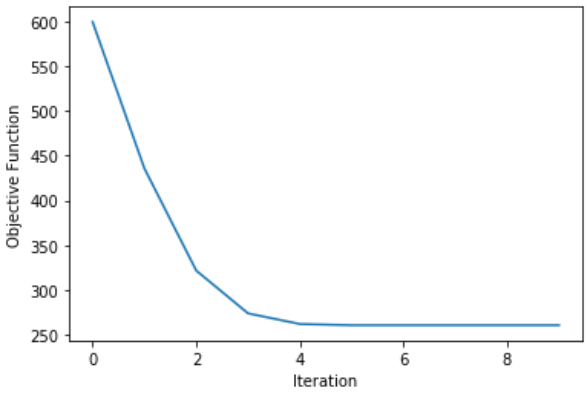


Figure 5: Graphing the objective function for each iteration of the K-Means algorithm

The objective function used was the sum of all distances from every point to its closest centroid. It's clear from the above image that as each iteration varies the location of each centroid this function decreases, up to a limit. This limit is reached when the centroids are stable.

Appendices

Appendix A: Full Code for Task 1

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

def pol_regression(features_train, y_train, degree):
    xM = np.vander(features_train, degree+1, increasing=True) #creates a vande
rmonde matrix using x and the degree. the 'increasing' bool reverses the power
s applied in the VM matrix

    xT = xM.transpose() #transposes the vandermonde matrix

    parameters = np.linalg.solve(xT.dot(xM), xT.dot(y_train)) #calculates the
weights based on w=(xT*X)^-
1*(xT*y). uses Linalg to calculate inverse as more stable
    return parameters

def eval_pol_regression(parameters, x, y, degree):
    xTrain = np.vander(x[0], degree+1, increasing=True) #creates two vandermon
de matrices
    xTest = np.vander(x[1], degree+1, increasing=True)  #for both training and
 testing data

    rmseTrain = np.sqrt(np.mean((xTrain.dot(parameters) - y[0])**2)) #gets Roo
t Mean Squared Error for the training model
    rmseTest = np.sqrt(np.mean((xTest.dot(parameters) - y[1])**2)) #gets RMSE
for the testing model

    rmse = rmseTrain, rmseTest #combines both error arrays
    return rmse

def main():
    data = pd.read_csv("Task1Data.csv").values #gets the values in the CSV fil
e into a 2D array
    np.random.shuffle(data) #shuffles data in order for the same test/train da
tanot to be chosen
    features = data[:, 0] #splits all feature data (x) into separate array
    output = data[:, 1] #splits all output data (y) into separate array

    #Section 1.2
    deg = 2 #**the degree for the polynomial. change this for various graphs**
    label = f"Degree {deg}" #the label for graphing the degree of the polynomi
al

    parameters = pol_regression(features, output, deg) #calls function which s
ets the weights for the features
```

```
    rangeArr = np.array(range(-
5, 5)) #initialises an array of integers between -5 and 5

    x = np.vander(rangeArr, deg+1, increasing=True) # creates a VM matrix for
the range of -5 to 5
    y = x.dot(parameters) #gets the y values by multiplying the VM matrix  wit
h the weights: y = Xa

    plt.plot(features, output, 'bo') #plots dataset
    plt.plot(rangeArr, sorted(y), 'g') #plots range with the respective y valu
es
    plt.legend(["Dataset", label])
    plt.show()

    #Section 1.3:
    #Split dataset into training & testing
    ratio = 0.7 #defines ratio of train:test data - 70% training as specified
in brief
    num = len(features) #gets the length of the dataset
    splitLine = round(num*ratio) #defines where to make the split for the test
 & train datasets

    features = features[:splitLine], features[splitLine:] #splits features int
o test & train data
    output = output[:splitLine], output[splitLine:] #splits outputs into test
& train data

    degrees = [1, 2, 3, 5, 10] #specifies which degrees for which to plot trai
ning & testing RMSE
    rmse = np.zeros([len(degrees), 2]) #initialises an array for the RMSE of t
raining and testing data
    for i in range(len(degrees)): #iterates for each value in degree array
        parameters = pol_regression(features[0], output[0], degrees[i]) #deter
mines weights for the current degree
        rmse[i] = eval_pol_regression(parameters, features, output, degrees[i]
) #gets training and testing RMSE

    plt.semilogy(degrees, rmse[:, 0]) #plot RMSE logarithmically against each
degree
    plt.semilogy(degrees, rmse[:, 1])
    plt.xlabel("Degree") #sets labels and legend
    plt.ylabel("Root Mean Squared Error")
    plt.legend(('RMSE on training set', 'RMSE on test set'))
    plt.show()

main()
```

Appendix B: Full Code for Task 2

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

def showPoints(dataset, centroids, cluster_assigned, x, y, labels):
    #scatter plot for dataset with points colour-coded to their cluster:
    plt.scatter(dataset[:, x], dataset[:, y], c=cluster_assigned)
    plt.scatter(centroids[:, x], centroids[:, y], c='r') #plot centroids in re
d
    plt.xlabel(labels[x]) #set labels for x & y axes
    plt.ylabel(labels[y])
    plt.show()

def compute_euclidean_distance(vec_1, vec_2):
    #calculate distance between two vectors
    diff = 0
    for i in range(len(vec_1)): #iterates over each dimension in vectors
        diff += (vec_1[i] - vec_2[i]) ** 2 #increments by the squared distance
 in current dimension
    distance = np.sqrt(diff) #takes square root to get final euclidean distanc
e

    return distance

def initialise_centroids(dataset, k):
    centroids = np.zeros([k, 4]) #specifies shape of centroid array
    for cent in range(k):
        randIndex = np.random.randint(0, len(dataset)) #generates a random ind
ex for the dataset array
        centroids[cent] = dataset[randIndex] #sets each centroid to the random
 index
    return centroids

def kmeans(dataset, k):
    centroids = initialise_centroids(dataset, k) #initialises centroids to ran
dom data vals
    #initialises necessary blank arrays:
    oldCent = sumCent = newCent = np.zeros([k, 4])
    cluster_assigned = np.zeros(len(dataset))
    dist = np.zeros(k)
    objectiveFunction = 0
    totObjFun = []

    for i in range(10):

    #while (abs(oldCent - centroids) != 0).all(): #loop until centroids are st
able
```

```
        for point in range(len(dataset)): #iterates over each point

            #calculate euclidean distance to each centroid and store each valu
e in array
            for cent in range(k):
                dist[cent] = compute_euclidean_distance(dataset[point], centro
ids[cent])

            #find closest centroid from distance array - finds index of minimu
m value of dist array
            closest = np.where(dist == np.min(dist))[0][0]

            objectiveFunction += dist[closest] #adds closest distance to objec
tive function sum

            #classify each point as its closest centroid
            cluster_assigned[point] = closest

            sumCent[closest] += dataset[point] #add the point's co-
ordinates to all other points in cluster

        #calculate average:
        for cent in range(k): #iterates centroids
            for dim in range(4): #iterates for each feature
                #finds average of each feature of each centroid, thus calculat
ing its next position
                newCent[cent][dim] = sumCent[cent][dim] / len(dataset[cluster_
assigned == cent])

        totObjFun.append(objectiveFunction)
        objectiveFunction = 0
        sumCent = np.zeros([k, 4]) #resets sum of all value positions
        oldCent = centroids #set values of previous centroids in order to comp
are in while loop
        centroids = newCent #assign new centroid values

    plt.plot(range(len(totObjFun)), totObjFun)
    plt.xlabel("Iteration")
    plt.ylabel("Objective Function")
    plt.show()

    return centroids, cluster_assigned #once centroids are stable, return cent
roid positions & each points' cluster

def main():
    csv = pd.read_csv("../Task2Data.csv") #initialises pandas DataFrame with t
he contents of relevant file
    dataset = csv.values #ignores headers
```

```
    labels = csv.columns #ignores data

    print("K = 2:")
    centroids, cluster_assigned = kmeans(dataset, k=2) #run k-means for k=2
    showPoints(dataset, centroids, cluster_assigned, 0, 1, labels)
    showPoints(dataset, centroids, cluster_assigned, 0, 2, labels)

    print("K = 3:")
    centroids, cluster_assigned = kmeans(dataset, k=3) #run k-means for k=3
    showPoints(dataset, centroids, cluster_assigned, 0, 1, labels)
    showPoints(dataset, centroids, cluster_assigned, 0, 2, labels)

main()
```

Bibliography

Kumar, A., 2018. *What are the weaknesses of the standard k-means algorithm (aka. Lloyd's algorithm)?.* [Online]
Available at: https://www.quora.com/What-are-the-weaknesses-of-the-standard-k-means-algorithm-aka-Lloyds-algorithm
[Accessed 26 November 2019].

Mathonline, n.d. *The Distance Between Two Vectors.* [Online]
Available at: http://mathonline.wikidot.com/the-distance-between-two-vectors
[Accessed 27 November 2019].

Mordvintsev, A. K. &. A., 2013. *Understanding K-Means Clustering.* [Online]
Available at: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_ml/py_kmeans/py_kmeans_understanding/py_kmeans_understanding.html
[Accessed 26 November 2019].

Weisstein, E. W., n.d. *"Least Squares Fitting--Polynomial." From MathWorld.* [Online]
Available at: http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html
[Accessed 27 November 2019].

Weisstein, E. W., n.d. *Vandermonde Matrix.* [Online]
Available at: http://mathworld.wolfram.com/VandermondeMatrix.html
[Accessed 26 November 2019].