**Winter of Code 6.0**

**Building a Machine Learning Algorithm Library**

**Arman Bhattacharjee**

**IIT (ISM) Dhanbad**

**Objectives:**

The problem statement of WOC 6.0 asked us to implement Linear Regression, Polynomial Regression, Logistic Regression, K-Nearest Neighbour, N-layered neural Network and K-Means Clustering algorithms from scratch in Python without the use of any machine learning libraries. Only Numpy, Matplotlib and Pandas libraries were allowed.

**Linear Regression Implementation:**

The provided training data contained 50000 training examples, with 20 features. Using linear regression, we were required to find a linear function which would be able to predict the output values from just the input values. After defining all the required functions, I started by defining two zero arrays for the weights and bias values. The algorithm then iterated several times to change the values of weights and biases to get a minimum ‘mean squared error’. The mean squared error is a quadratic function, so it has only one minimum value. Several values of learning rate were tried and the value of the learning rate giving the fastest convergence without overshooting was chosen. The training set was split in an 80:20 ratio into the training and cross-validation sets, and the output of the cross-validation set was predicted with the weights and bias obtained from the training set. The R2-score of both the data sets were obtained, and the cost v/s iteration graph was also plotted. I also plotted actual value v/s predicted value scatter plots for each feature and the linear function obtained and I plotted a graph of predicted values v/s actual values for the training and cross-validation set.

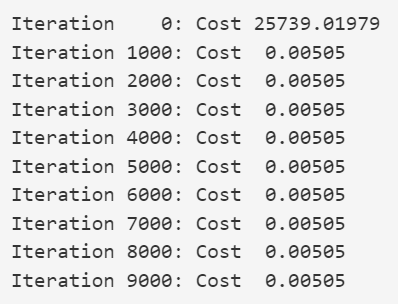
Initially, my code took a lot of time to run because I had used a lot of nested for loops, but after I vectorized my code, I was able to run the algorithm in seconds. I used the same approach in the other algorithms as well.

The values of learning rates I tried:

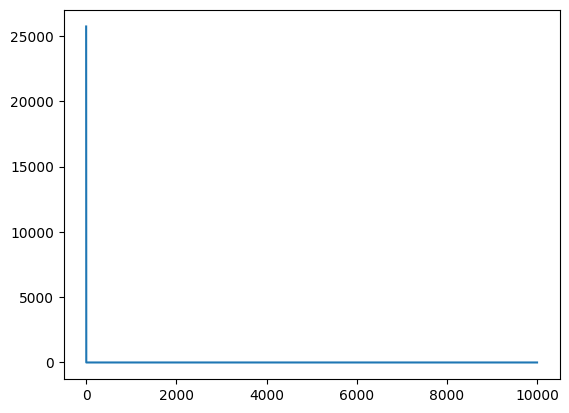
* 0.001: The cost constantly decreased but slowly.
* 0.01: The cost decreased quicker than before.
* 0.1: The cost decreased even quicker.
* 1.0: The cost reached a minimum after around 1000 iterations, same as for 0.1
* 2.0: Since the algorithm was able to handle a learning rate of 1.0, so I tried 2.0, but the cost started increasing instead of decreasing, i.e. the algorithm was ‘overshooting’.

After trying all these values of the learning rate, I decided to take 1.0 to be my learning rate for this algorithm.

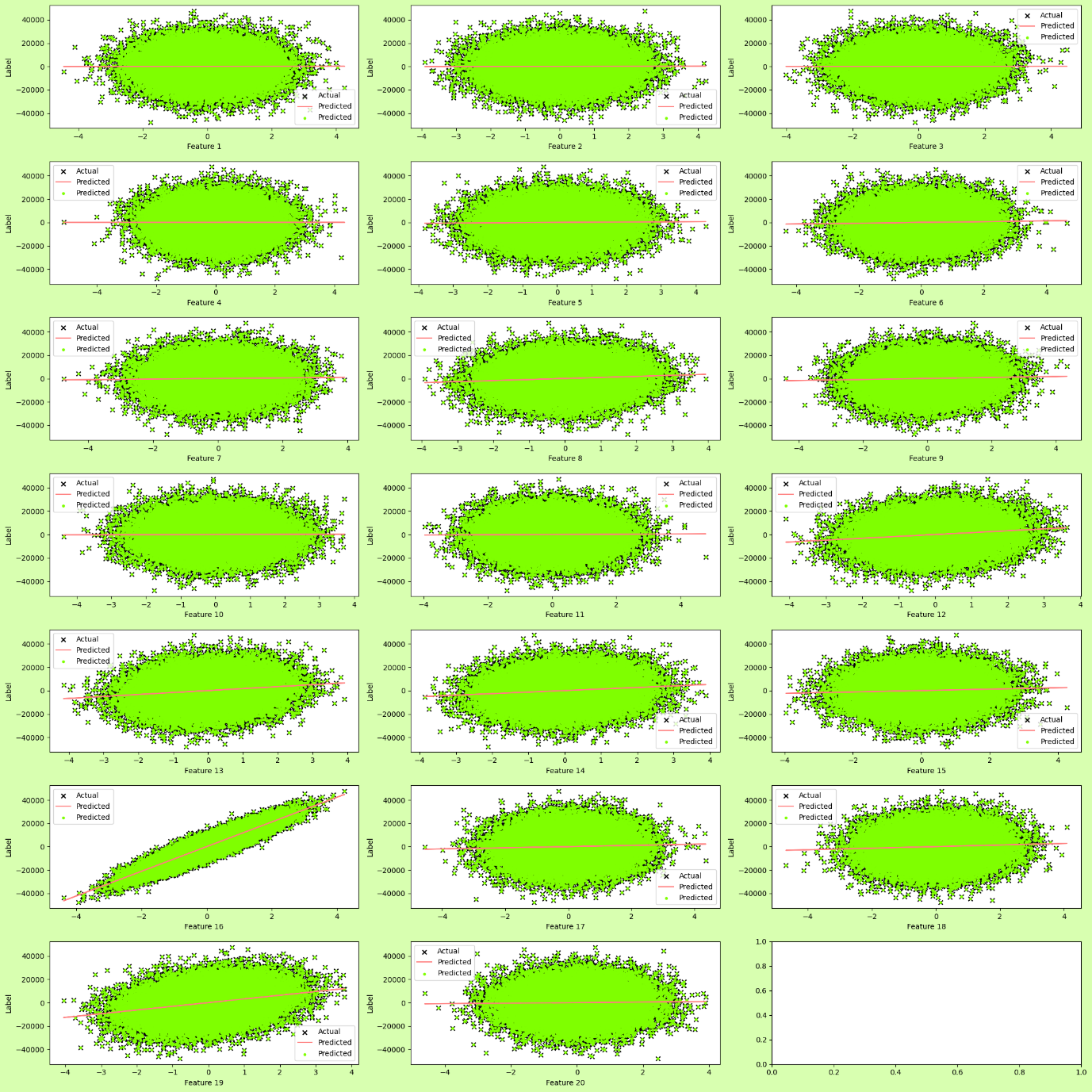
The cost decreased with the number of iterations in the following way for learning rate 1.0:



The cost v/s iteration graph was obtained as follows:

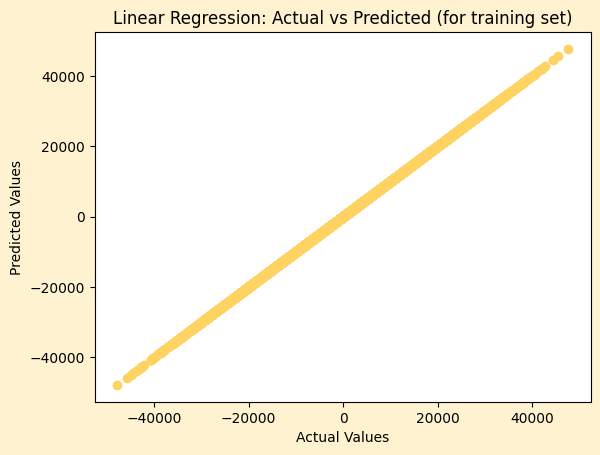


The predicted values v/s actual values scatter plot is as follows: (I have only shown plots of the first 6 features here)



The line shows the linear function for that particular feature.

This is the graph for the predicted values v/s the actual values (for the training set):



**Polynomial Regression Implementation:**

The given training data set had 50000 training examples and 3 features. With feature engineering, I created multiple new features. I defined a function which multiplied the features among themselves and raised them to the required powers to generate a multinomial of a given degree. I have also defined a function which generates a polynomial of a given degree, but since that was producing high cost and low accuracy, I have not used this for training my model. I divided this dataset into an 80:20 ratio as well. After training the model on the training data set, I used to train the model to predict the outputs of the cross-validation data set. The R2-score of both the data sets were obtained, and the cost v/s iteration graph was also plotted. I also plotted the actual values v/s predicted values scatter plot and graphs.

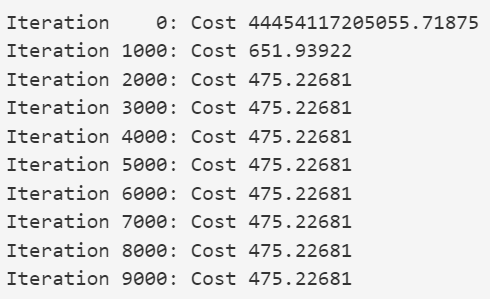
Initially, the features I generated were only the 3 features raised to a given degree, i.e. there were no interacting terms. After that, I generated features by multiplying only 2 of the 3 original features and raising them to the required powers. However, both of these methods led to very poor performance with high costs and low accuracies. Finally, with the multinomial expansion of the features, I was able to obtain a relatively low cost along with high accuracy (R2-score).

The values of learning rates I tried:

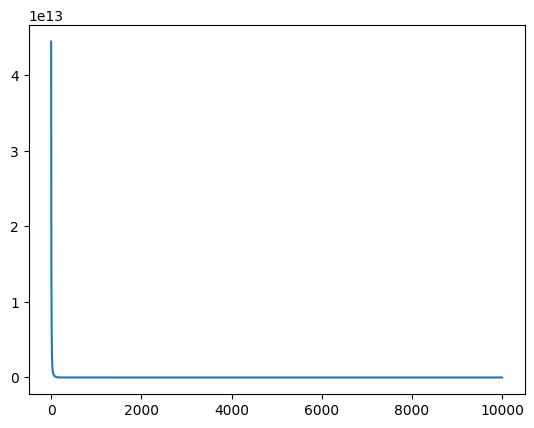
* 0.001: The cost decreased constantly but very slowly.
* 0.01: The cost decreased quicker than before but it was still quite slow.
* 0.1: The cost decreased at a desired rate and converged at around 2000 iterations.
* 1.0: I tried a higher learning but I encountered an overflow error in my cost function.

After trying all these values of the learning rate, I decided to take 0.1 to be the learning rate for this algorithm.

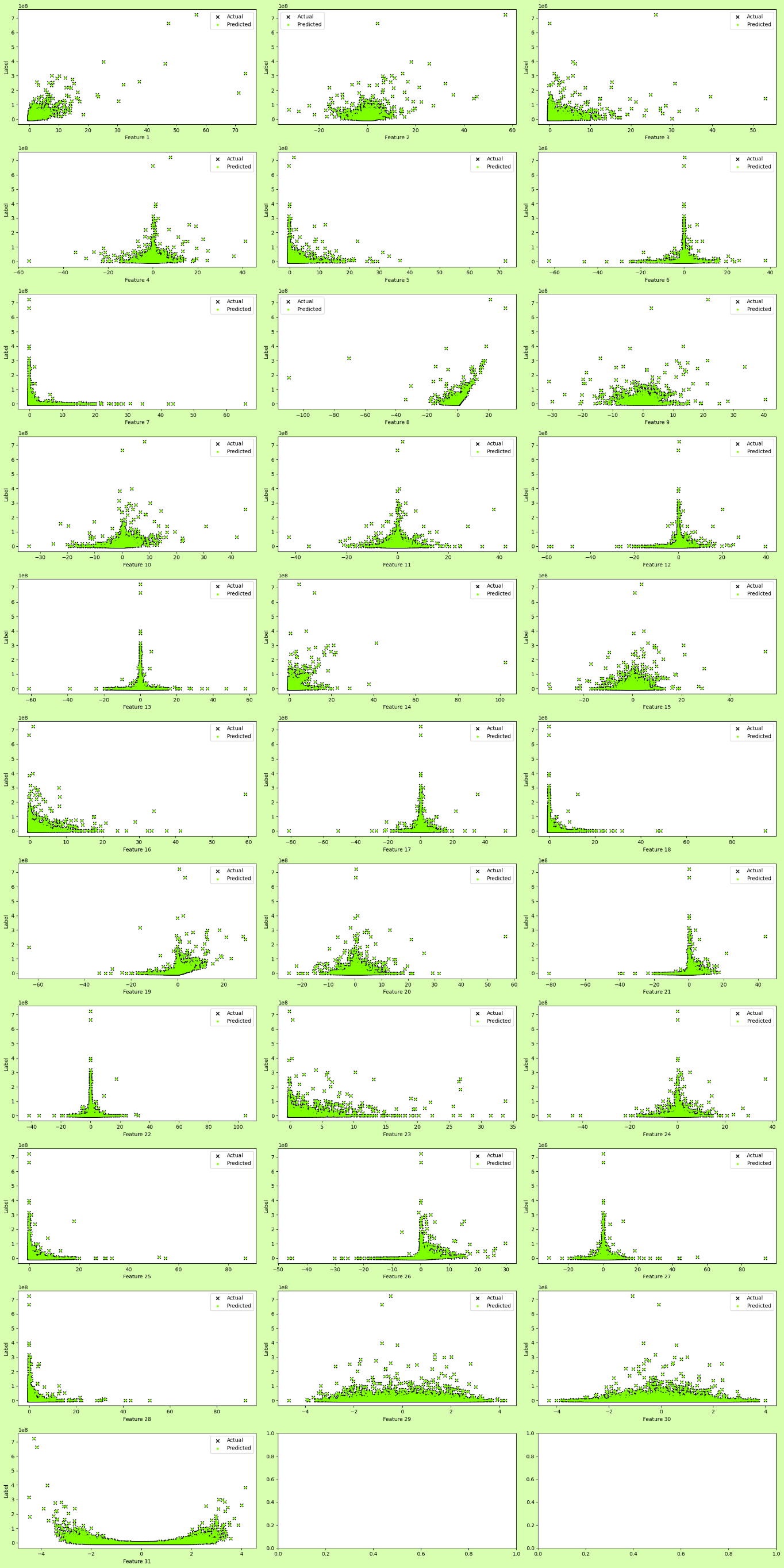
The cost decreased with the number of iterations in the following way for a learning rate of 0.1:



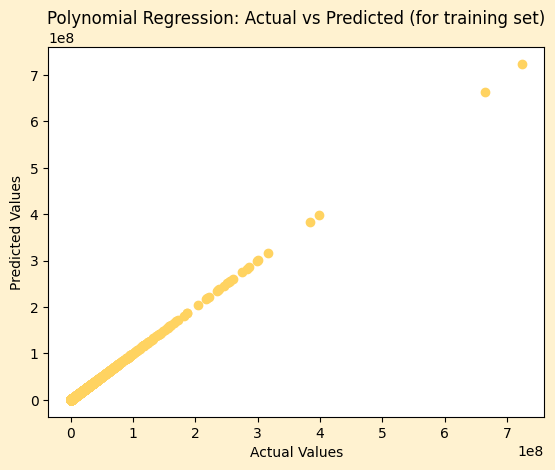
The cost v/s iteration graph was obtained as follows:



The predicted values v/s the actual values scatter plot is as follows: (I have shown plots of the first 6 features here):

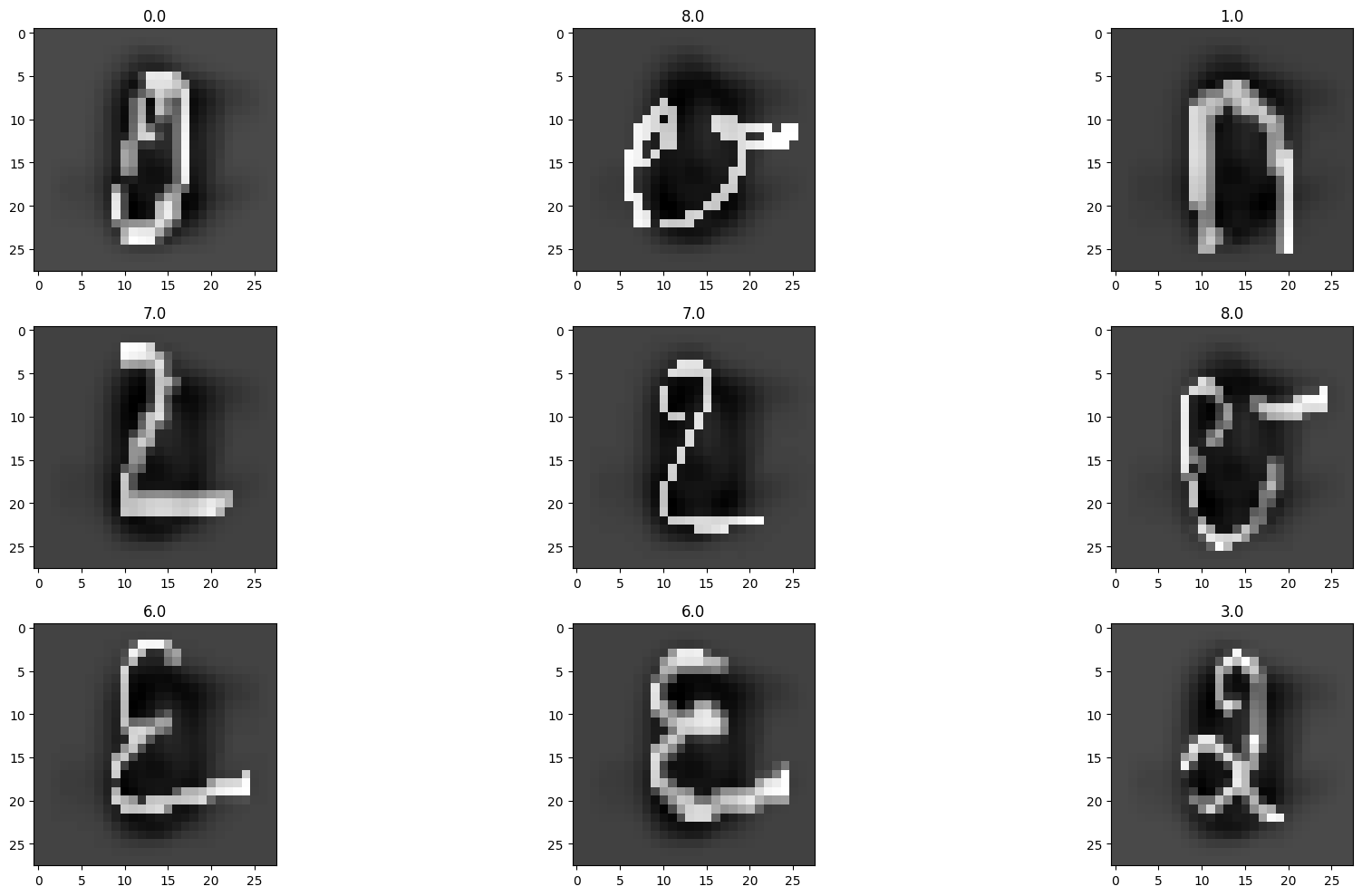


This is the graph for the predicted values v/s the actual values (for the training set):



**Logistic Regression Implementation:**

For the classification algorithms, the given dataset contained 30000 training examples and 784 features, where each feature corresponded to pixels of a 28 pixels \* 28 pixels image. The first 9 examples look like this on plotting:



We were required to recognize the digits in the given picture by training a logistic regression model. Like in the other models, I split the dataset into an 80:20 ratio into the training and cross-validation sets. In this model, instead of the mean squared error, binary cross entropy loss was used. Since it was a multiclass classification, the ‘one v/s all’ technique was used, in which one class is taken as the positive class, while all the remaining classes are taken as negative classes. The accuracies of both datasets were obtained. I plotted the cost v/s iteration graph for each digit. Lastly, I picked a random example from the cross-validation and test dataset, visualized it using Matplotlib, and printed the predicted and actual value of the digit.

The values of learning rates I tried:

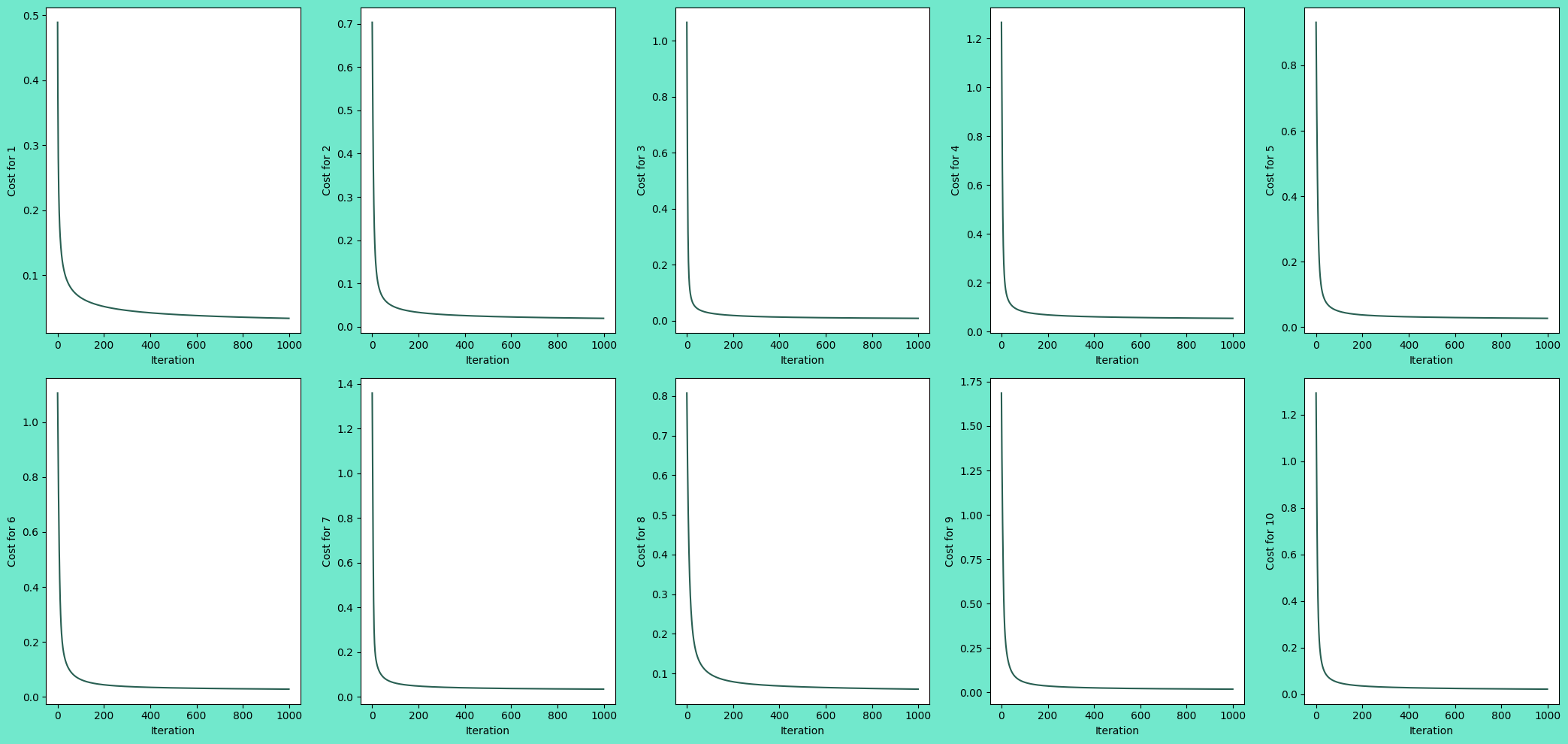
* 0.001: The cost decreased but very slowly.
* 0.01: Cost decreased relatively quickly but was still slow.
* 1.0: Cost decreased quickly without overshooting.

So, I decided to take 1.0 to be the learning rate for this algorithm.

The costs decreased with the number of iterations in the following way for learning rate 1.0:

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

The cost v/s iteration graph was obtained as follows:

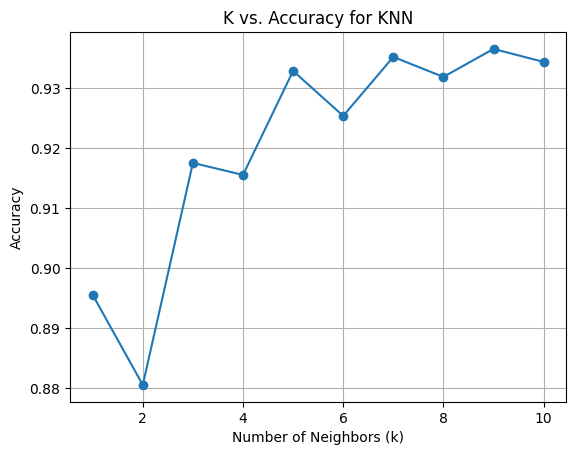


The values of weights and bias found by the model were used to predict the outputs of the test data set.

**K-Nearest Neighbour Implementation:**

For the KNN algorithm as well, the given dataset contained 30000 training examples and 784 features, where each feature corresponded to pixels of a 28 pixels \* 28 pixels image. We were required to recognize the digits with the help of the K-Nearest Neighbour Algorithm. The working principle behind this algorithm is that it considers an example to be of the same kind as that of the majority among its ‘K’ nearest neighbours. The given dataset was split into two parts in the ratio of 80:20: one part was used to train the KNN model and the outputs of the second part were predicted and the accuracy was calculated.

In a previous run of the code, I used K values from 1 to 10 and calculated the accuracy for each K. Since this takes a lot of time to run (running the algorithm for only K=9 took about 30 minutes to run, I have commented out this part of the code, but have kept the output. I have also plotted the accuracy v/s K graph:



From the graph, it can be concluded that K=9 gives the maximum accuracy. So using this value of K, I predicted the outputs of the test dataset provided.

**N-Layered Neural Network Implementation:**

For the neural network as well, the same dataset with 30000 training examples and 784 features, each corresponding to the pixels of a 28 pixels \* 28 pixels image was provided. We were required to recognize the digits with the help of a neural network. Although I have used only one hidden layer for this digit recognition task, the algorithm can be expanded to have any number of hidden layers. For the input and hidden layer, I have used the ReLU activation function while for the output layer, I have used the softmax activation function. Since the output activation function is softmax, I have used the cross-entropy function as the cost function.

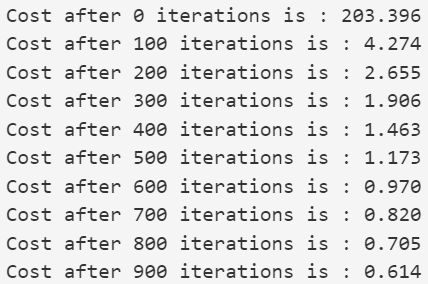
The algorithm was initialised with the weights and bias randomly chosen and appropriately scaled. The labels for the given data are also ‘one-hot encoded’. The outputs for each iteration were calculated through ‘forward propagation’ and the weights and biases were updated through ‘backward propagation’. In backward propagation, the gradients are calculated by the chain rule of differentiation.

I tested my neural network algorithm with 1250 neurons in the hidden layer and the following values of learning rate:

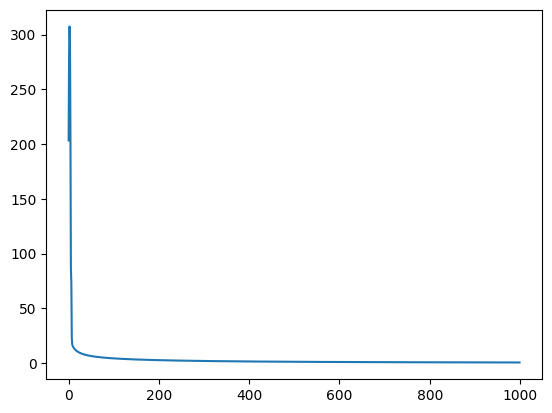
* 0.0001: The cost decreases quickly.
* 0.0009: The cost decreases even faster. No overshooting was observed.
* 0.001: This raises a runtime warning because of an invalid value obtained in the denominator during the backpropagation. This means the value of the learning rate is too big.

So, I chose 0.0009 as the learning rate for the neural network algorithm.

The costs decreased with the number of iterations in the following way for learning rate 0.0009:



The cost v/s iteration graph was obtained as follows:



After this, I randomly chose a number from the training set, visualised it, and printed the predicted value.

**K-Means Implementation:**

The dataset provided for the K-Means algorithm contained 178 training examples and 13 features. Using the K-Means Algorithm, we were required to cluster these training examples into ‘K’ clusters. The algorithm starts by randomly initializing ‘K’ centroids. Then the distance is measured between each point and centroid and the points are assigned to their nearest centroid. The centroids are updated by taking the mean of all the points assigned to them, and points are again assigned to the new centroids. The loop breaks when the position of centroids remains the same in two consecutive iterations.

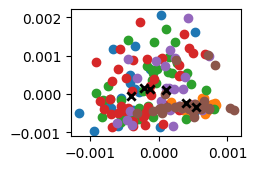
The number of clusters, i.e. K, can be chosen by two methods, the elbow method and the silhouette score. In the elbow method, the sum of squared errors (SSE), i.e. the squared Euclidean distance between each data point and centroid, is plotted against K. The ‘elbow’ point is the point where the rate of decrease of SSE slows down. The K value at the elbow point is chosen as the optimal value for K.

In the silhouette method, the average silhouette score is calculated for different values of K. The value of the silhouette score varies from 1 to -1, where 1 is the ideal case and -1 is the worst possible case. The silhouette score measures how similar a datapoint is to its own cluster compared to the other clusters. A high value of silhouette score means better-defined clusters, so the value of K with the highest value of silhouette score is chosen.

For the given dataset, I used the elbow method to plot SSE values for K=1 to K=10, and the graph seems to flatten after K=4. Using the silhouette method, I calculated the silhouette scores for the same range of K, and I got the highest value of 0.6578 for K=2 and the second highest value of 0.6001 for K=6.

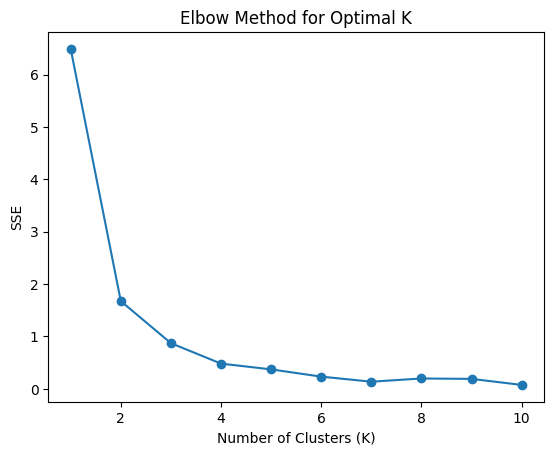
From these two values of K, I decided to use K=6 for my K-Means algorithm. I did not use K=2 even though it had the highest silhouette score because the SSE v/s K graph does not flat out at 2, and from the silhouette score v/s data point graph, it seems like for K=2, a greater number of data points having very low silhouette scores are present.

The centroids converged after 17 iterations and this is the final scatter plot obtained between feature 0 and feature 1:



The black crosses represent the centroids of the clusters obtained from the K-Means algorithm.

This is the graph obtained for the elbow method:



The silhouette score v/s data point graph for K=6 was obtained as follows:

