

**winTER OF CODE – MACHINE LEARNING**

MACHINE LEARNING



January 22, 2024

IIT ISM DHANBAD

NAME: PRIYAM PRITAM PANDA

­­­

The problem statement is to implement the following Machine Learning Algorithms/models:

* Linear Regression
* Polynomial Regression
* Logistic Regression (or Classification)
* N-Layer Neural network
* KNN (K Nearest Neighbours)
* K-Means

**Implementation of Linear Regression from scratch:**

The training data given had 50,000 training examples and 21 columns, i.e. 20 features and 1 column for the “True labels”. I divided the training data into two sets:

1. Training Set, with 40,000 training examples
2. Cross-Validation Set, with 10,000 training examples

The goal is to fit a linear curve into the training data which would minimize the cost. For this, first I initialized the parameters. For the weights I created a 2-Dimensional numpy array of size (1,20) and set all the values to zeros. For the bias I just assigned a variable “b” to zero. The **Mean Squared Error** function was used as the cost function because it is a two-degree polynomial so will have only one minimum where we’d get the lowest cost.

Then I trained my Linear Regression model on my training set to get the final values of parameters which would minimize the cost. While training the model, I tried various values for the learning rate (alpha) and the number of iterations until I found the optimized values of alpha and no. of iterations for which the gradient descent will converge faster without overshooting at any point.

In the beginning, I used “for loops” to calculate the cost, gradients and the R2-score, due to which my code was running very slowly. But then I tried to use vectorization in my code which made it run much faster than before. Also, it made the code so much compact and easy to understand.

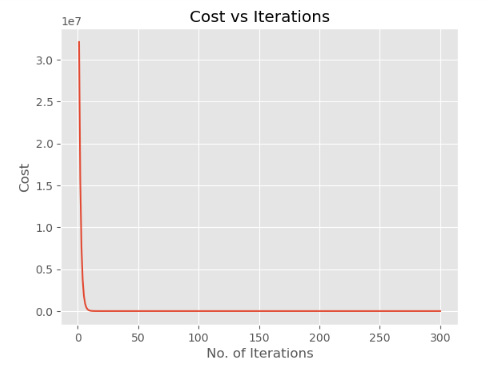
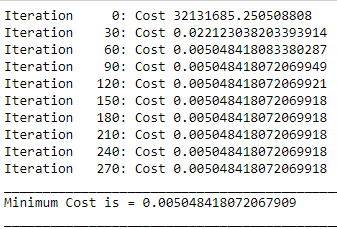
*Deciding the value of hyperparameters:*

During training, several values of learning rate were tried:

1. 0.001: With this value of learning rate, the cost continuously decreased but take a long time to run. So, I tried some higher values of learning rate.
2. 0.01: With this value of learning rate, the cost again decreased continuously and faster than in the case of 0.001.
3. 1: With this value of learning rate, the cost sometimes decreased and sometimes increased, which means that for this value of learning rate, the cost “overshooted”. So, I decided to try a smaller value of learning rate.
4. 0.1: In this case, the cost didn’t overshoot, and was even faster than in the case of 0.01.
5. 0.3: In this case, the cost decreased constantly and even faster than in the case of 0.1. So, I decided this value as my final value of learning rate.

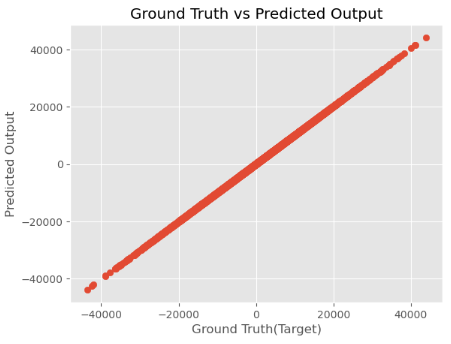
Also, at first, I took the no. of iterations = 400, and found that the cost was already converging at 160 iterations, so, I took a slight less no. of iterations, i.e. 300.

Now, after setting learning rate = 0.3 and no. of iterations = 300, the cost decreased in the following way:



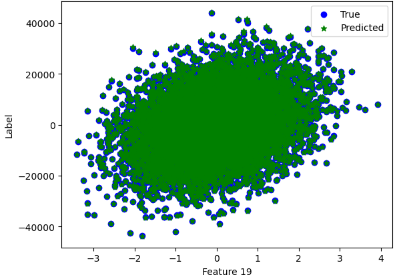
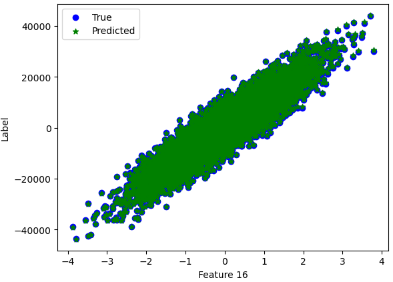
The R2- score on the **training set** was =

Then, I ran the model on the **cross-validation (CV) set**, and plotted a graph between the predictions made by the model and the true labels of the CV set:



The linearity of the graph shows that the predictions are almost the same as the true labels.

The R2- score on the **CV set** was = 

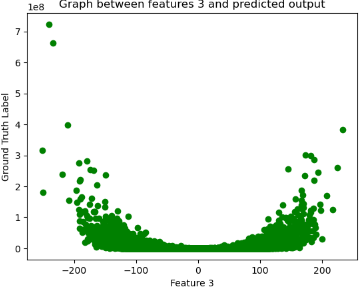
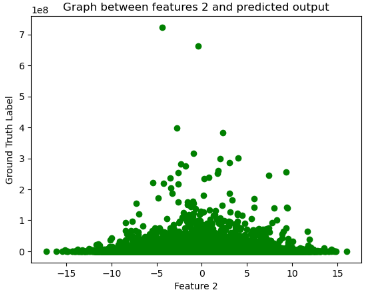
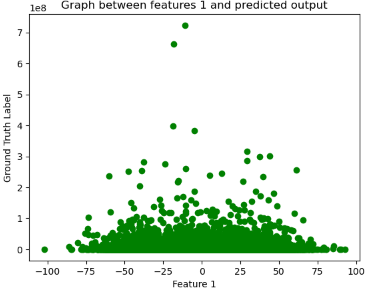
Then I plotted a scatter plot between two features of the CV set, i.e. 16 and 19, with the true labels and with the predictions of the model: 

Then I ran the model on the given test set to predict using the final parameters produced.

**Implementation of Polynomial Regression from scratch:**

The training data given had 50,000 training examples and 3 features. I divided Training set of 40,000 examples and Cross-Validation set of 10,000 examples.

In polynomial regression, we have to fit a n-degree polynomial into the training set for which the cost minimizes and then we have to run the model on the given test set to make predictions by using the parameters produced after training the model. In order to create a n-degree polynomial, I used feature engineering in which I created features of higher degrees by multiplying the given features with each other in different proportions. The **Mean Squared Error** function was used as the cost function because it is a two-degree polynomial so will have only one minimum where we’d get the lowest cost. Before running the gradient descent algorithm, first I initialized the parameters. For the weights, I created a row vector with no. of columns equal to the total no. of engineered features of that particular degree and assigned all of them to zero. Similarly, for the bias, I just created a variable ‘b’ equal to zero.

Also, I plotted graphs between all the 3 given features and the ground truth label to get an idea of how the data points are present in the space and which order polynomial should I use to fit the data properly and minimize the cost. The graphs are as follows: 

***The graph between parameter 3 and ground truth label shows that an even order polynomial will fit the data better****.*

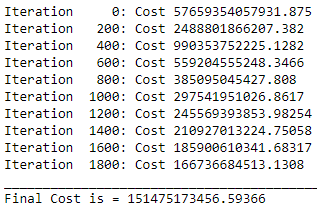
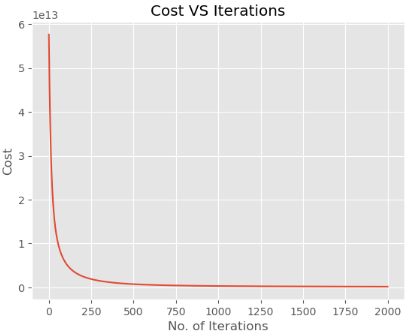
So, then I ran the gradient descent algorithm with even degree polynomials; such as 2, 4, 6, etc; and found that the saturated cost is minimum for degree 6. So, I decided to take a 6-degree polynomial as final.

*Deciding the value of hyperparameters*:

For a 6-degree polynomial, I tried different values of learning rates:

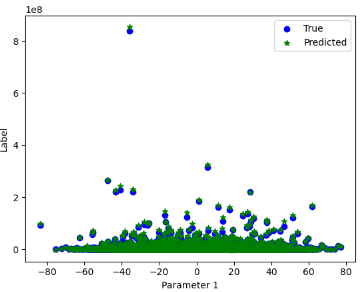
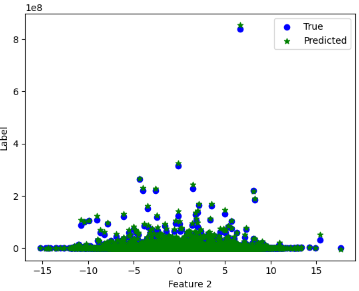
1. 0.001: The cost decreases constantly but converging process was slow which indicated that learning rate should be increased.
2. 1: Overshooting was observed which indicated that the learning rate is too high.
3. 0.01: The cost constantly decreases and was faster than in the case of 0.001.
4. 0.03: The cost decreases at a higher rate than in the case of 0.01 but it did overshoot at some points.

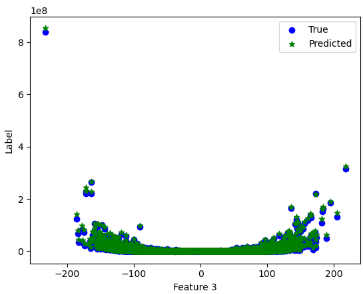
So, I finally concluded learning rate = 0.01, for which the cost decreased in the following way:

R2 score on the **training data set** =

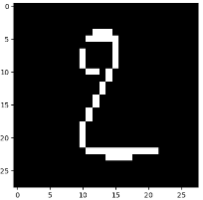
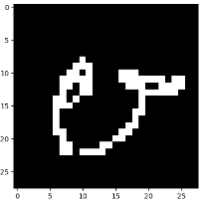
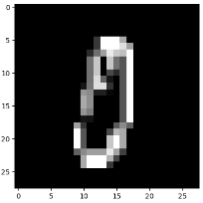
Then I ran the model on the **CV set** and R2 score on CV set is=

The following plots show how the predicted values fit the given true labels:  



We can see that the model fits the data well.

**Implementation of Logistic Regression from scratch:**

The training data given had 30,000 examples. Each training example was a set of 784 pixels (28x28) which means, we need to classify the pictures. To get an idea of what these images are of some of the images from the training data set are: 

So, the pictures given to us are of hand-written digits, means we need to classify handwritten digits.

Further, I divided the training data into:

1. Training set- 25,000 examples
2. Cross-Validation- 5,000 examples

Some important points about my logistic regression model:

* The Mean Squared Error can’t be used as the cost function because it has several local minima (because we are using the non-linear softmax function) which would make the convergence of cost difficult. So here, we use the Sparse Categorical Crossentropy Loss Function, and then sum over all the losses for every training example to get a convex cost function.
* The ground truth labels need to be **“One-hot encoded”.**

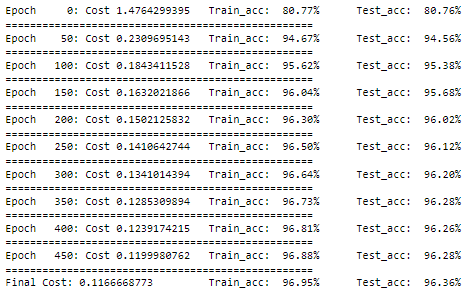
All the other things remain same as earlier- the parameters and the gradient descent.

*Deciding the values of the hyperparameters*:

During training, several values of learning rate were tried:

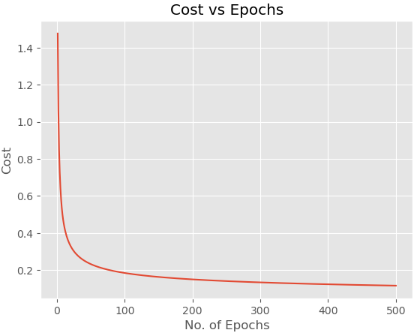
1. 1e-3: Overshooting was observed which indicated that learning rate is too big.
2. 1e-5: Cost decreased constantly but at a slow rate which indicate which indicated to increase the learning rate.
3. 5e-5: Cost decreased constantly and faster than in case of 1e-5.

So, I finally decided to take learning rate = 5e-5. The cost decreased in the following way:

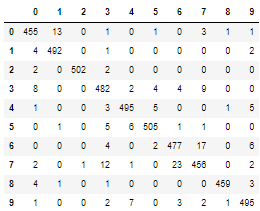


Here, the final accuracy on the training set and the cross-validation set can be seen.

The graph of cost vs no. of iterations is:

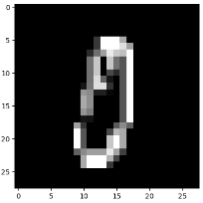
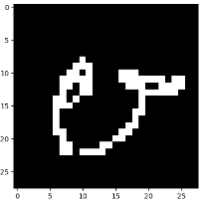
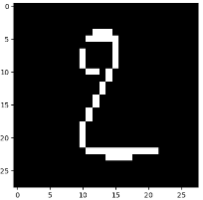


Then I ran the model on the Cross-Validation data set and calculated **f1 score** using the **confusion matrix** as follows:

 F1 score on the cross-validation set is =

Now, after getting the final values of all the weights and biases, we use these values to make predictions on the given test set.

**Implementation of n-Layer Neural Network from scratch:**

The training data given had 30,000 examples. Each training example was a set of 784 pixels (28x28) which means, we need to classify the pictures. To get an idea of what these images are of some of the images from the training data set are:   

So, the pictures given to us are of hand-written digits, means we need to classify handwritten digits.

Further, I divided the training data into:

1. Training set- 25,000 examples
2. Cross-Validation- 5,000 examples

Some important points about my neural network:

* This network is used for classification.
* Softmax function was used in the final output layer.
* The cost function used here was Sparse Categorical Crossentropy function.
* The ground truth labels were one hot encoded.
* To find the gradients of the parameters, I used back propagation.

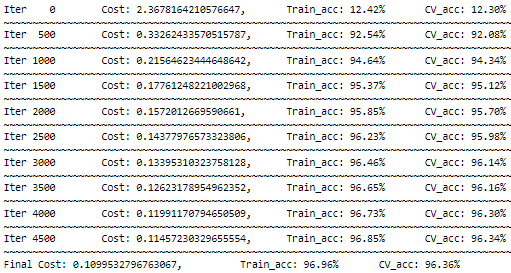
*Deciding the hyperparameters*:

During training, several values of learning rate were tried:

0.001: The cost decreased constantly but at a slow rate which indicated that the learning rate is too small.

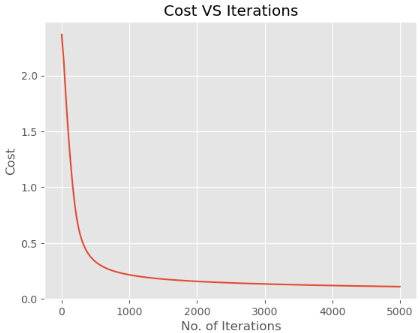
1. 0.1: Overshooting was observed which indicated that the learning rate was too large.
2. 0.01: The cost decreased constantly and faster than in the case of 0.001.
3. 0.03: The cost decreased constantly and even faster than in the case of 0.1.

So, finally I set learning rate = 0.03. The cost decreased in the following way:



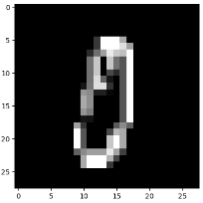
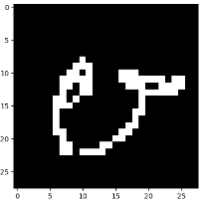
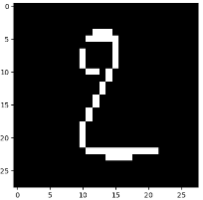
Here, the final accuracy on the training set and the cross-validation set can be seen.

The graph of cost vs no. of iterations is:



Now, after getting the final values of all the weights and biases, we use these values to make predictions on the given test set.

**Implementation of KNN (K-Nearest Neighbours) from scratch:**

The training data given had 30,000 examples. Each training example was a set of 784 pixels (28x28) which means, we need to classify the pictures. To get an idea of what these images are of some of the images from the training data set are:   

So, the pictures given to us are of hand-written digits, means we need to classify handwritten digits.

Further, I divided the training data into:

1. Training set- 25,000 examples
2. Cross-Validation- 5,000 examples

The KNN algorithm is a lazy algorithm. No training of the model done is done here. It just takes a bunch of data points (from the training data) and when a new data point is added (from the cross-validation set), then it computes the distances of the point from all the pre-loaded points and observes the nearest “K” nearest points. And by looking at their true labels, it observes that which ground truth is the most repeated in those “K” nearest points (or neighbours) and then the algorithm predicts that the new data should be of the same class that is most repeated. Since, there is no training of the model, so there are no parameters needed.

When I ran the model on the training set and the cross-validation set, the accuracy came to be:



Then, I ran the model on the training and the test set, where the training set was the pre-loaded bunch of points and the model made predictions for each point from the test set.

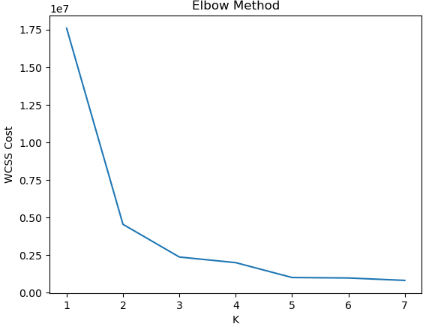
**Implementation of K-Means from scratch:**

The training data given had 178 training examples and 13 columns. Means each training example represented a point in a 13-dimensional space. K-Means algorithm comes under unsupervised learning where no ground truth label is specified. In K-Means the model just makes clusters of the data points depending on their Euclidean distances from one another.

The K-Means algorithm basically consists of the following steps:

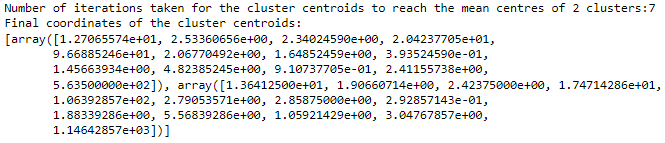
* Chooses any “K” random points from the training examples as the cluster centroids of “K” clusters.
* Measure the Euclidean distance of each point from all the “K” cluster centroids and assigning each point to the cluster to which it is closest to.
* Once all the points are assigned to their respective clusters, then the average coordinate of all the points in a cluster is calculated, and the cluster centroid of that cluster is then moved to that coordinate. This is done for all the “K” clusters.

The 1st step is done once in the beginning, and then the 2nd and 3rd steps are repeated till the cluster centroids don’t move any more or they have already reached the mean coordinate position of the clusters.



Now, to choose the appropriate number of cluster- “K”, I used the “Elbow Method”, and I got ***K = 2***.

Then I ran the K-Means algorithm on my training set to train it and get the final locations of the cluster centroids.



Now, after getting the final positions of the cluster centroids, we use these values to make predictions on the given test set.