Machine Learning

ML Bootcamp

Rohan Garg

23JE0822

IIT (ISM) Dhanbad

This file contains the report of the algorithms that I made during my Winter Of Code 6.0 Hackathon for the Machine Learning Division. This file contains all the implementations that I did into my code during the code period. It also contains some mistakes which were done previously but were rectified in the due course of time.

I used the following libraries in my code:

a) google.colab -> I used this library to import my train and test datasets inside my google colab notebook.
b) numpy -> I used this library to handle my data and used it to perform different operations on the dataset in order to get outputs. I also used it as an alternative to the loops as vectorisation techniques to make my code time efficient.
c) pandas -> I used this library to handle the imported csv files.
d) matplotlib -> I used this library to get the plots of different parameters inside my code.

Linear Regression

- The linear regression refers to a supervised Machine Learning model which takes some data as input, learns or trains itself on a linear function with the help of training data and then produces an output on the basis of the function made by it while training.

- Overfitting- Overfitting refers to a condition that the model fits the training data completely well but it underperforms in the test data. Hence to prevent overfitting I divided my dataset of 50,000 training examples to 40,000 as training set and 10,000 as cross validation set. The r2 score of the training set and cross validation set gave me a good picture that my model was performing well and doesn't overfit.

- Z-score Normalization- Z-score normalization, also known as standardization or z-score scaling, is a method used to standardize the scale of a feature by transforming it into a standard normal distribution. It efficiently helps in decreasing the run time of the code and makes the machine learning model time efficient. It results it making the data with mean=0 and standard deviation=1

$$z = \frac{x-\mu}{\sigma}$$

- Shape of x- (40000,20)
  Shape of y- (40000,)
  Shape of x_cv- (10000,20)
  Shape of y_cv- (10000,)
  Function- f_wb(x[i]) = w.x[i] + b
  Shape of w- (20,)

- Cost Function-

$$J(w,b) = \frac{1}{2m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

$$f_{w,b}(x^{(i)}) = wx^{(i)} + b$$

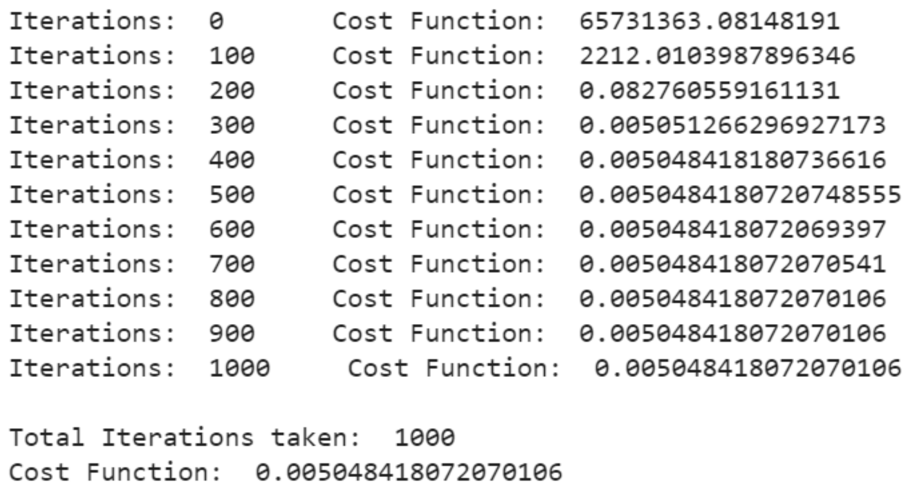
- Gradient and Gradient Descent-

$$w = w - \alpha\frac{\partial J(w,b)}{\partial w}$$
$$b = b - \alpha\frac{\partial J(w,b)}{\partial b}$$

$$\frac{\partial J(w,b)}{\partial w} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{w,b}(x^{(i)}) - y^{(i)})$$

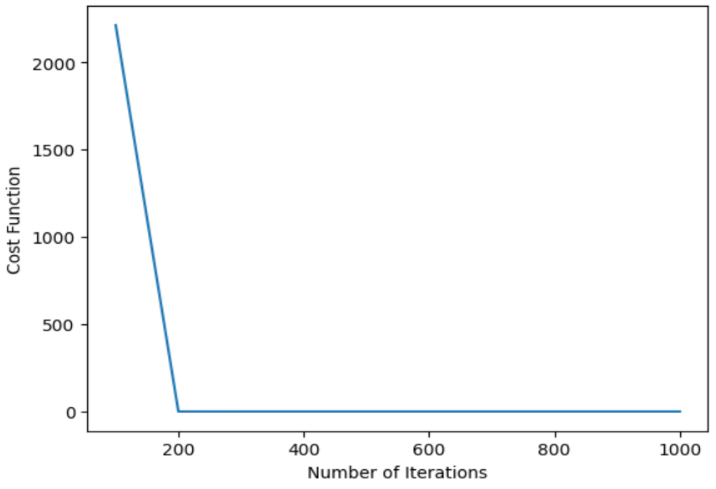
- On taking alpha= 0.05 and total number of iterations= 1000, I got the following results:



```
Iterations:  0       Cost Function:  65731363.08148191
Iterations:  100     Cost Function:  2212.0103987896346
Iterations:  200     Cost Function:  0.082760559161131
Iterations:  300     Cost Function:  0.005051266296927173
Iterations:  400     Cost Function:  0.005048418180736616
Iterations:  500     Cost Function:  0.0050484180720748555
Iterations:  600     Cost Function:  0.005048418072069397
Iterations:  700     Cost Function:  0.005048418072070541
Iterations:  800     Cost Function:  0.005048418072070106
Iterations:  900     Cost Function:  0.005048418072070106
Iterations:  1000     Cost Function:  0.005048418072070106

Total Iterations taken:  1000
Cost Function:  0.005048418072070106
```

- Plot of cost vs iterations-



- R2 value- The R-squared value, also known as the coefficient of determination, is a statistical measure that represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s). It is often used to evaluate the goodness of fit of a regression model.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

My r2 value for training dataset is:

```
0.9999999999232193
```

- Now I used the values of w and b that I got after gradient descent for determining the labels for the cross validation set according to my machine learning model in order to check that the model is correct or overfitting the data.

  My r2 value for cross validation dataset is:

```
0.9999999999216167
```

- As I can clearly see that the r2 value for the training dataset is pretty good and the r2 value for the cross validation set is also very good and almost equal to the training r2 value. Hence, I can conclude that my machine learning linear regression model is doing pretty well on the training data and with the help of cross validation I can clearly see that my model is not overfitting on the data.

- Then I used the test data and predicted the output labels for the test dataset and stored it in a new csv file.

[mistakes made and rectified:

- Firstly, I didn't use vectorisation in my code which made the code a lot of time taking. After that I used some vectorisation techniques like np.dot(), np.sum() etc which made the code very time efficient.
- Firstly, I didn't use the feature scaling which took a lot of time for my code to run then I used z-score normalization which decreased a lot of run time of the code.]

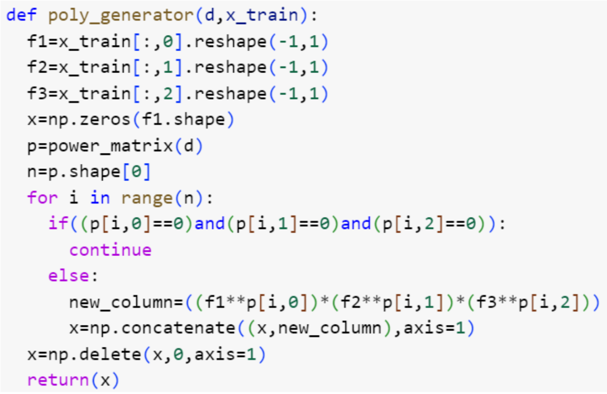
Polynomial Regression

- The Polynomial regression refers to a supervised Machine Learning model which takes some data as input, learns or trains itself on a polynomial function with the help of training data and then produces an output on the basis of the function made by it while training.

- Overfitting- To address the problem of overfitting I divided my dataset of 50,000 training examples to 45,000 as training set and 5,000 as cross validation set. The r2 score of the training set and cross validation set gave me a good picture that my model was performing well and doesn't overfit.

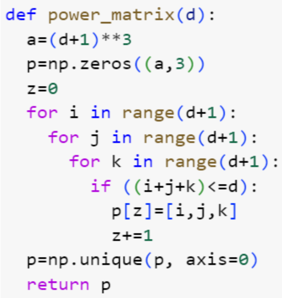
- Shape of x- (45000,3)
  Shape of y- (45000,)
  Shape of x_cv- (5000,3)
  Shape of y_cv- (5000,)

- A function which gives a generalised polynomial- I constructed a function which gave me a polynomial generated of any degree.



```
def poly_generator(d,x_train):
  f1=x_train[:,0].reshape(-1,1)
  f2=x_train[:,1].reshape(-1,1)
  f3=x_train[:,2].reshape(-1,1)
  x=np.zeros(f1.shape)
  p=power_matrix(d)
  n=p.shape[0]
  for i in range(n):
    if((p[i,0]==0)and(p[i,1]==0)and(p[i,2]==0)):
      continue
    else:
      new_column=((f1**p[i,0])*(f2**p[i,1])*(f3**p[i,2]))
      x=np.concatenate((x,new_column),axis=1)
  x=np.delete(x,0,axis=1)
  return(x)
```

Here I got f1, f2 and f3 as the three different features present in x and reshaped it in the shape (-1,1) which gives me a column matrix of all the elements of that particular matrix. Then I used power matrix function in order to get the powers that I would be needing in the polynomial for each term which is to be generated.



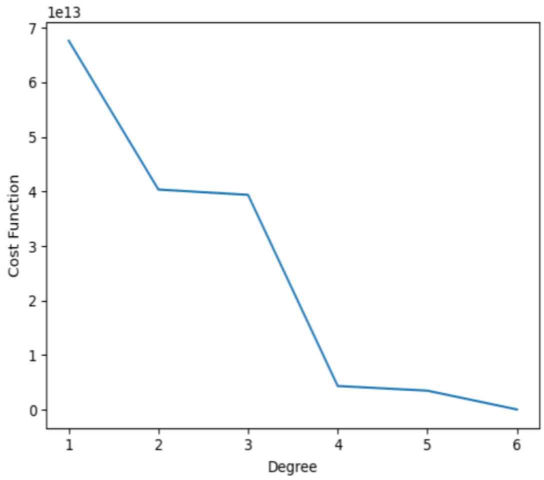
```
def power_matrix(d):
  a=(d+1)**3
  p=np.zeros((a,3))
  z=0
  for i in range(d+1):
    for j in range(d+1):
      for k in range(d+1):
        if ((i+j+k)<=d):
          p[z]=[i,j,k]
          z+=1
  p=np.unique(p, axis=0)
  return p
```

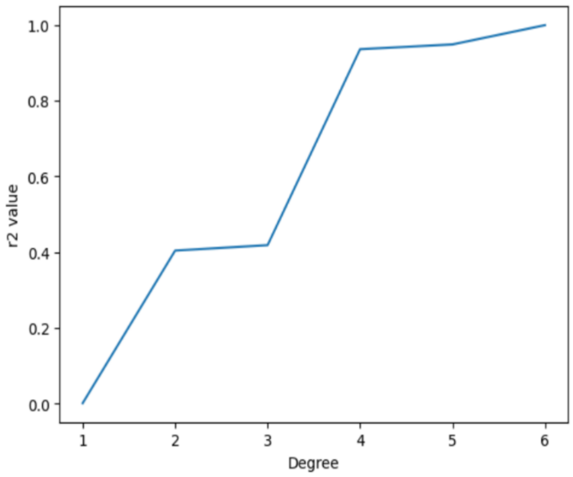
After getting the power matrix I just use these powers and then concatenate the resulting column matrix into my original training dataset x.

- Then all the same operations are done for that newly generated x with the help of poly_generator function like I did in linear regression namely cost function, gradients and gradient descent.

- I checked for different values of degrees of the polynomial and got the following results.

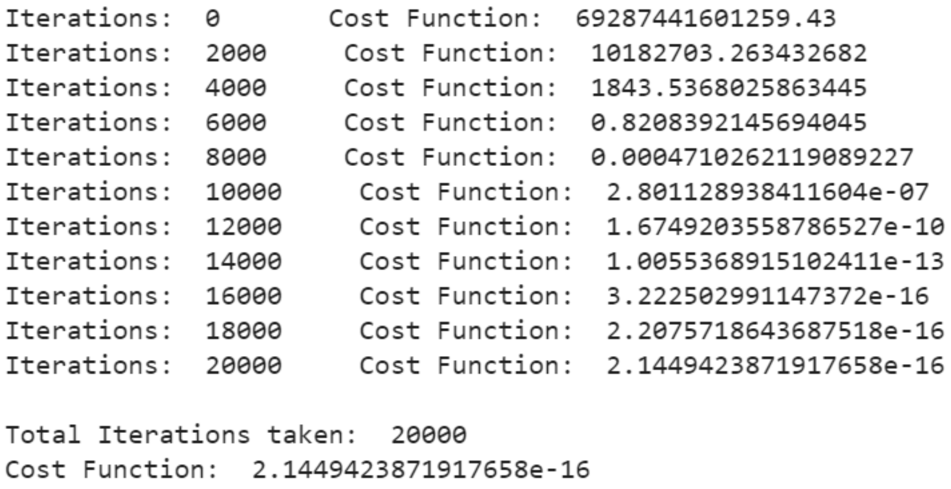
S no	Degree	Iterations	Cost	R2 value
1	1	1000	67653274576799.69	0.0006632774025594479
2	2	1000	40349282514248.08	0.4039827340334746
3	3	1000	39376926654409.16	0.4183458365477124
4	4	1000	4317098100258.3105	0.9362302165914195
5	5	1000	3460468490441.4487	0.9488838750005557
6	**6**	**1000**	**12176699060.245138**	**0.9998201325418904**





- On the basis of my several attempts I found the degree= 6 as the best for my Polynomial Machine Learning model as it was giving a pretty good r2 value and was also doing well on the cross validation set.

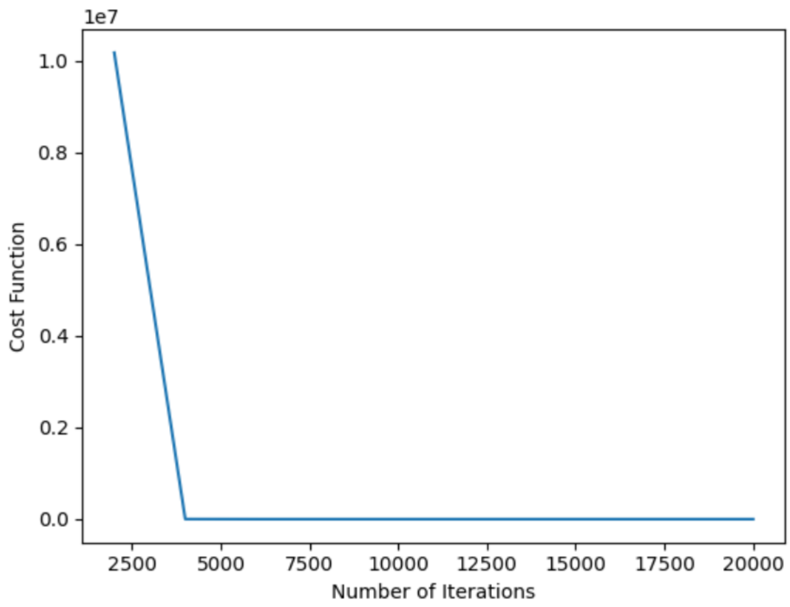
- On taking alpha= 0.2 and total number of iterations= 20000, I got the following results:



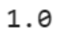
```
Iterations:  0       Cost Function:  69287441601259.43
Iterations:  2000     Cost Function:  10182703.263432682
Iterations:  4000     Cost Function:  1843.5368025863445
Iterations:  6000     Cost Function:  0.8208392145694045
Iterations:  8000     Cost Function:  0.0004710262119089227
Iterations:  10000     Cost Function:  2.801128938411604e-07
Iterations:  12000     Cost Function:  1.6749203558786527e-10
Iterations:  14000     Cost Function:  1.0055368915102411e-13
Iterations:  16000     Cost Function:  3.222502991147372e-16
Iterations:  18000     Cost Function:  2.2075718643687518e-16
Iterations:  20000     Cost Function:  2.1449423871917658e-16

Total Iterations taken:  20000
Cost Function:  2.1449423871917658e-16
```

- Plot of cost vs iterations-



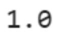
- My r2 value for training dataset is:



```
1.0
```

- Now I used the values of w and b that I got after gradient descent for determining the labels for the cross validation set according to my machine learning model in order to check that the model is correct or overfitting the data.

  My r2 value for cross validation dataset is:



```
1.0
```

- As I can clearly see that the r2 value for the training dataset is pretty good and the r2 value for the cross validation set is also very good and almost equal to the training r2 value. Hence, I can conclude that my machine learning linear regression model is doing pretty well on the training data and with the help of cross validation I can clearly see that my model is not overfitting on the data.

- Then I used the test data and predicted the output labels for the test dataset and stored it in a new csv file.

[<u>mistake made and rectified</u>:

- I firstly didn’t use the generalised model for the polynomial and it is not practically possible to manually write the code for bigger degree of polynomial. Hence, I formed the generalised polynomial function for better accuracy. The use of vectorization also helped me to decrease the runtime of the code.]

Logistic Regression

- The logistic regression refers to a supervised Machine Learning model which takes some data as input, learns or trains itself and given an output as a label being a probability between 0 and 1.

- Overfitting- To address the problem of overfitting I divided my dataset of 30,000 training examples to 25,000 as training set and 5,000 as cross validation set. The accuracy of the training set and cross validation set gave me a good picture that my model was performing well and doesn't overfit.

- Normalization/Scaling the features- I just here divided my complete dataset by the maximum value of the data (here 255) which scaled my data between 0 and 1 and made my code time efficient.

- Shape of x- (784,25000)
  Shape of y- (1,25000)
  Shape of x_cv- (784,5000)
  Shape of y_cv- (1,5000)

- Sigmoid function- It is the function that is used in the logistic regression which gives us the result in the range of 0 to 1 only.

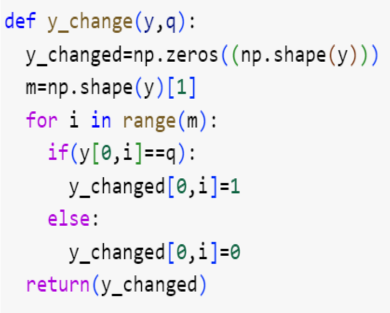
$$g(z) = \frac{1}{1+e^{-z}}$$

- Cost Function-

$$f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$$

$$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = (-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)$$

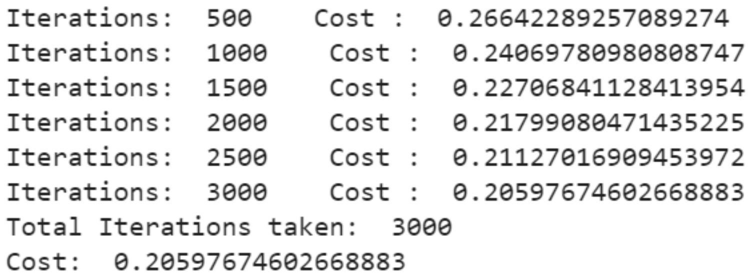
- Here my idea was to firstly change the y into 0's and 1's which signify 'not 0' and '0' respectively. Then I used this new y in the logistic regression for seeing that the label should be 0 or not. I did this for the other 9 labels as well.



```
def y_change(y,q):
  y_changed=np.zeros((np.shape(y)))
  m=np.shape(y)[1]
  for i in range(m):
    if(y[0,i]==q):
      y_changed[0,i]=1
    else:
      y_changed[0,i]=0
  return(y_changed)
```

- Now for doing this algorithm for 10 times would be very time consuming hence I used the idea of parallel computation in which I made a w of 10 different rows and did the gradient descent on all the 10 labels parallelly which made my code time efficient.

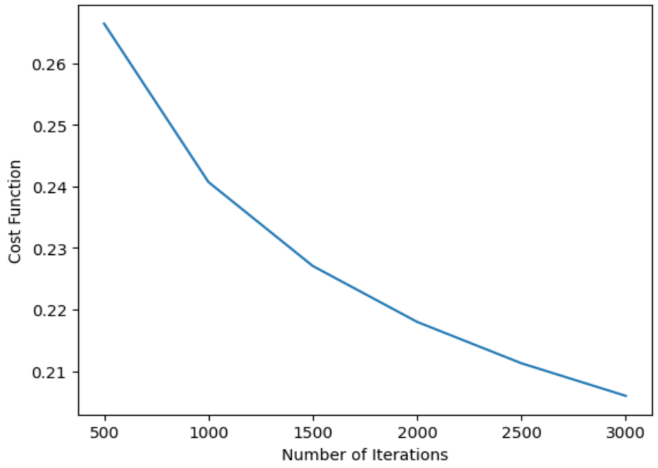
- On taking alpha= 5 and total number of iterations= 3000, I got the following results:



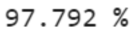
```
Iterations:  500    Cost :  0.26642289257089274
Iterations:  1000    Cost :  0.24069780980808747
Iterations:  1500    Cost :  0.22706841128413954
Iterations:  2000    Cost :  0.21799080471435225
Iterations:  2500    Cost :  0.21127016909453972
Iterations:  3000    Cost :  0.20597674602668883
Total Iterations taken:  3000
Cost:  0.20597674602668883
```

- On doing the gradient descent I calculated the value of y predicted by the model. I just calculated the probabilities for all the 10 labels and then saw which label has max probability and hence assigned that value of label to y predicted.

- Plot of cost vs iterations-



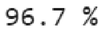
- My accuracy for training dataset is:



```
97.792 %
```

- Now I used the values of w and b that I got after gradient descent for determining the labels for the cross validation set according to my machine learning model in order to check that the model is correct or overfitting the data.

  My accuracy value for cross validation dataset is:



```
96.7 %
```

- As I can clearly see that the accuracy for the training dataset is pretty good and the accuracy for the cross validation set is also very good and almost equal to the training accuracy. Hence, I can conclude that my Logistic regression Machine Learning model is doing pretty well on the training data and with the help of cross validation I can clearly see that my model is not overfitting on the data.

- Then I used the test data and predicted the output labels for the test dataset and stored it in a new csv file.

[mistake made then rectified:

- Earlier, I tried to perform the gradient descent for each individual label which took a lot of time for each label. Then I tried the concept of the parallel computation hence then I calculated the w and b parallelly and hence the run time of the cost decreased a lot.]

K- Nearest Neighbours (KNN)

- The K- Nearest Neighbours (KNN) refers to a supervised Machine Learning model which takes some data as input, and on the testing set it sees for its nearest k- neighbours for each test data with respect to the training data and takes that label as output with most frequency in k nearest neighbours.

- I divided my dataset of 30,000 training examples to 25,000 as training set and 5,000 as cross validation set. The accuracy of the cross validation set gave me a good picture that my model was performing well.

- Shape of x- (25000,784)
  Shape of y- (25000,)
  Shape of x_cv- (5000,784)
  Shape of y_cv- (5000,)
  K = 3

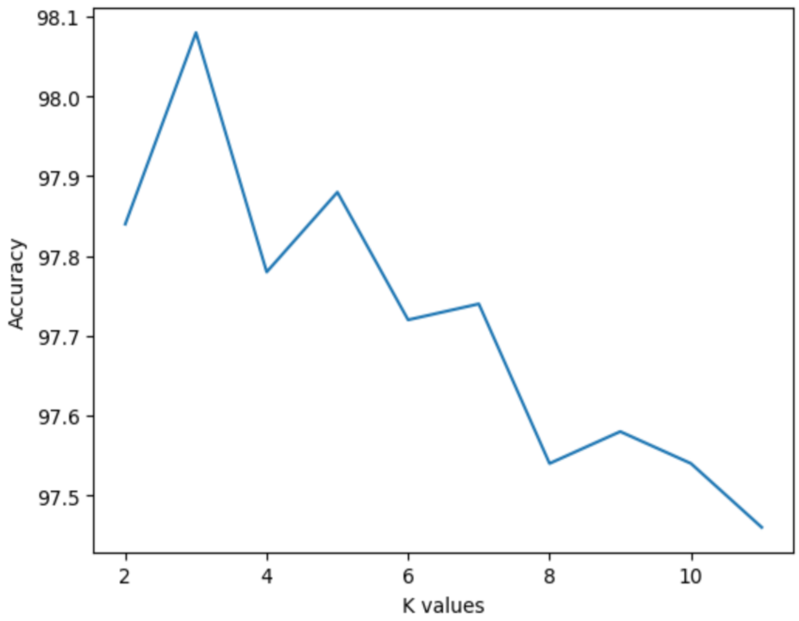
- Euclidean Distance- It is the type of distance that is used to calculate the distance between two points.

$$d(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

- For each point in the cross validation set (training point) I calculated the distance between each point of the cross validation set with all the points of the training set. I see for the nearest k neighbours of that point and the label with maximum frequency is selected as the output label for that particular data point.

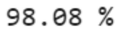
- For choosing the best K value for the KNN, I used different values of k and then calculated the respective accuracies. The value of K with most accuracy is chosen.

S no	Value of K	Accuracy
1	2	97.84 %
2	**3**	**98.08 %**
3	4	97.78 %
4	5	97.88 %
5	6	97.72 %
6	7	97.74 %
7	8	97.54 %
8	9	97.58 %
9	10	97.54 %
10	11	97.46 %



- According to the graph I take k = 3 as the bast value of k for my KNN model.

- My accuracy for cross validation dataset is:



```
98.08 %
```

- As I can clearly see that the accuracy for the cross validation set is also pretty good. Hence, I can conclude that my KNN Machine Learning model is doing pretty well and I can finalise the model.

- Then I used the test data and predicted the output labels for the test dataset and stored it in a new csv file.

[<u>method which I used earlier but changed it which helped to decrease my code runtime</u>:

- Earlier, I tried to implement KNN by computing the distances point to point which take a lot of time due to a large dataset (it calculated 25000 X 5000 distances).
- But then I vectorised the code and calculated the distance along axis = 1 or along the column which resulted in the decrease of the runtime of the code by 50%.]

Artificial Neural Network

- The Artificial Neural Network refers to a supervised Machine Learning model which takes some data as input, learns or trains itself by the help of the neural network model including the input layer, some hidden layers and the output layer.

- Overfitting- To address the problem of overfitting I divided my dataset of 30,000 training examples to 25,000 as training set and 5,000 as cross validation set. The accuracy of the training set and cross validation set gave me a good picture that my model was performing well and doesn't overfit.

- Normalization/Scaling the features- I just here divided my complete dataset by the maximum value of the data (here 255) which scaled my data between 0 and 1 and made my code time efficient.

- Shape of x- (784,25000)
  Shape of y- (1,25000)
  Shape of x_cv- (784,5000)
  Shape of y_cv- (1,5000)
  Shape of W1- (256,784)
  Shape of W2- (128,256)
  Shape of W3- (64,128)
  Shape of b1- (256,1)
  Shape of b2- (128,1)
  Shape of b3- (64,1)

- One hot encoding- It is the type of changing the y such that for each data of y will have 10 columns. The label of y is given the value of corresponding value as '1' and the other columns as '0'.

- Rectified Linear Unit (ReLU)- It is the function that is used in the hidden layers inside the neural network.

  $$f(x) = \max(0, x)$$

- Softmax activation- It is the layer that is specially used in the output layer when there are more than 2 possible outputs.

  $$p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- Initialise Parameters- Here I get the list of the number of the hidden layers and the number of the neurons present in each layer and randomly set the parameters of the layers namely W1, W2, W3, b1, b2, b3 inside a dictionary 'parameters'.

- Forward propagation- It is the function used to find the forward outputs of the layers and get the w and a as a form of the dictionary called ‘w_a_values.

  $$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$$
  $$a^{(l)} = f(z^{(l)})$$

  Where f = ‘ReLU’ activation for hidden layers and f = ‘sigmoid’ for output layer.

- Cost Function-

$$J(\mathbf{w}, b) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{N} 1\left\{y^{(i)} == j\right\} \log \frac{e^{z_j^{(i)}}}{\sum_{k=1}^{N} e^{z_k^{(i)}}}\right]$$

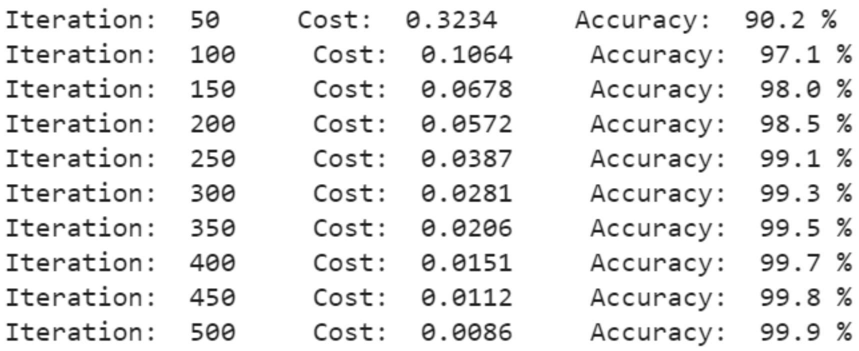
- Backward Propagation- It is the type of the method used to calculate the gradients (dW1, dW2, dW3, db1, db2, db3)

  $$\delta^{(3)} = a^{(3)} - y$$
  $$\frac{\partial J}{\partial W^{(3)}} = a^{(2)} \cdot \delta^{(3)}$$
  $$\frac{\partial J}{\partial b^{(3)}} = \delta^{(3)}$$
  $$\delta^{(2)} = (W^{(3)})^T \cdot \delta^{(3)} \cdot f^{(2)'}(z^{(2)})$$
  $$\frac{\partial J}{\partial W^{(2)}} = a^{(1)} \cdot \delta^{(2)}$$
  $$\frac{\partial J}{\partial b^{(2)}} = \delta^{(2)}$$
  $$\delta^{(1)} = (W^{(2)})^T \cdot \delta^{(2)} \cdot f^{(1)'}(z^{(1)})$$
  $$\frac{\partial J}{\partial W^{(1)}} = X \cdot \delta^{(1)}$$
  $$\frac{\partial J}{\partial b^{(1)}} = \delta^{(1)}$$

- Update the parameters- I updated all the parameters of W’s and b’s with respect to the dW’s and db’s obtained in the backward propagation.

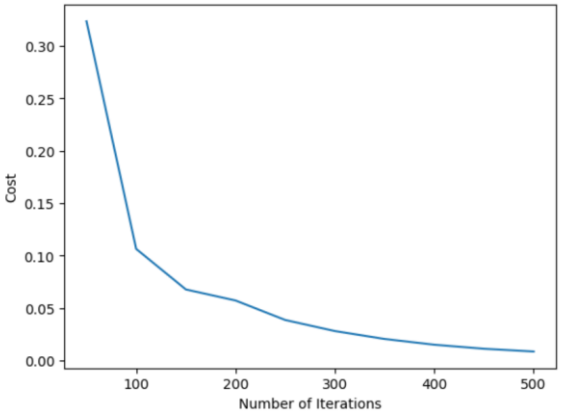
  $$w = w - \alpha \frac{\partial J(w, b)}{\partial w}$$
  $$b = b - \alpha \frac{\partial J(w, b)}{\partial b}$$

- On taking alpha= 1 and total number of iterations= 500, I got the following results:

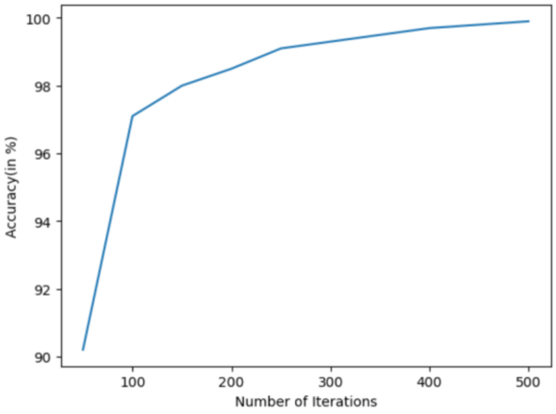


```
Iteration:  50     Cost:  0.3234     Accuracy:  90.2 %
Iteration:  100     Cost:  0.1064     Accuracy:  97.1 %
Iteration:  150     Cost:  0.0678     Accuracy:  98.0 %
Iteration:  200     Cost:  0.0572     Accuracy:  98.5 %
Iteration:  250     Cost:  0.0387     Accuracy:  99.1 %
Iteration:  300     Cost:  0.0281     Accuracy:  99.3 %
Iteration:  350     Cost:  0.0206     Accuracy:  99.5 %
Iteration:  400     Cost:  0.0151     Accuracy:  99.7 %
Iteration:  450     Cost:  0.0112     Accuracy:  99.8 %
Iteration:  500     Cost:  0.0086     Accuracy:  99.9 %
```

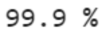
- Plot of cost vs iterations-



- Plot of cost vs accuracy-



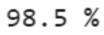
- My accuracy for training dataset is:

  ```
  99.9 %
  ```

- Now I used the values of W’s and b’s that I got after training for determining the labels for the cross validation set according to my machine learning model in order to check that the model is correct or overfitting the data.

  My accuracy value for cross validation dataset is:

  ```
  98.5 %
  ```

- As I can clearly see that the accuracy for the training dataset is pretty good and the accuracy for the cross validation set is also very good and almost equal to the training accuracy. Hence, I can conclude that my Artificial Neural Network Machine Learning model is doing pretty well on the training data and with the help of cross validation I can clearly see that my model is not overfitting on the data.

- Then I used the test data and predicted the output labels for the test dataset and stored it in a new csv file.

[mistakes:

- Firstly, I didn’t use the generalised model of the neural network but then I used the generalised model and could easily able to use 3 hidden layers and get a very good accuracy.
- In the construction of the generalised neural network firstly I tried to use the list for a and w but I couldn’t correctly complete the code then I used the dictionary for collecting the a and w values which made the code to run easily.]

K means Clustering

- The K means Clustering refers to an unsupervised Machine Learning model which takes some data as input and then makes k number of clusters out of that dataset.

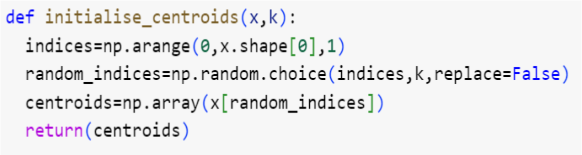
- Shape of x- (178,13)

- Euclidean Distance- It is the type of distance that is used to calculate the distance between two points.

$$d(P,Q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

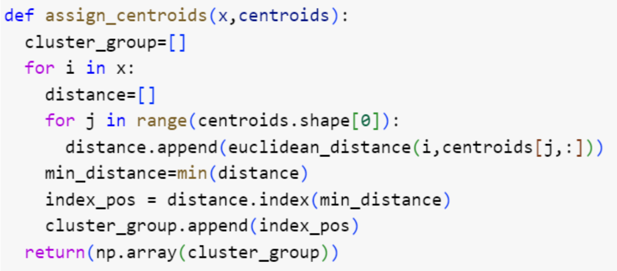
- Cost Function- The cost function in k means algorithm is defined as the sum of all the square of the distances of the points with their respective centroids.

- In the k means clustering algorithm we need to follow the respective steps->
  - Firstly, we have to assign some random 'k' number of centroids.



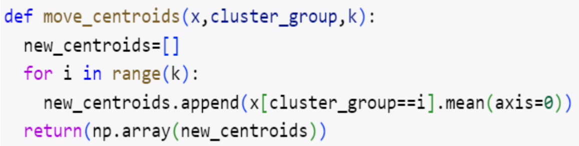
```
def initialise_centroids(x,k):
  indices=np.arange(0,x.shape[0],1)
  random_indices=np.random.choice(indices,k,replace=False)
  centroids=np.array(x[random_indices])
  return(centroids)
```

  - Then the points are assigned the centroids which is nearest to them.



```
def assign_centroids(x,centroids):
  cluster_group=[]
  for i in x:
    distance=[]
    for j in range(centroids.shape[0]):
      distance.append(euclidean_distance(i,centroids[j,:]))
    min_distance=min(distance)
    index_pos = distance.index(min_distance)
    cluster_group.append(index_pos)
  return(np.array(cluster_group))
```

  - After the assign of centroids, for each centroid we take the mean of the points related to that centroid and move the centroid to than mean position.



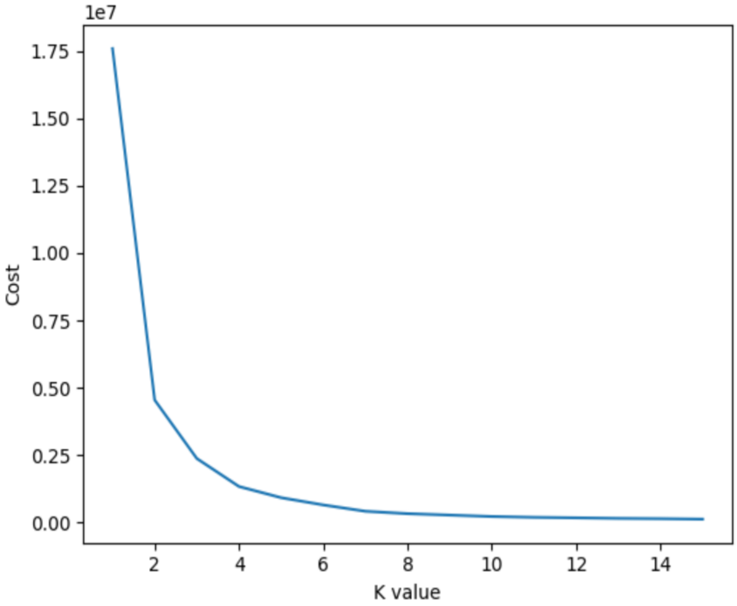
```
def move_centroids(x,cluster_group,k):
  new_centroids=[]
  for i in range(k):
    new_centroids.append(x[cluster_group==i].mean(axis=0))
  return(np.array(new_centroids))
```

  - We need to repeat the above two steps until the centroids gets equal to the old centroids of the previous iteration.

- There can be a case that when we assign the centroids initially and run the k means algorithm, the point sticks in a local minima and hence the clustering can't be done with a good accuracy. Hence, we need to run the k means for that same k some number of times (may be 100 or 200 times) and then choose the one with the least cost.

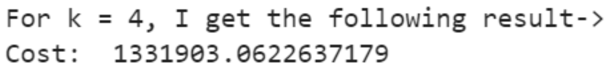
- To choose the best k for the k means algorithm we need to use the elbow method in which we run the k means algorithm for some values of k and then plot the graph of cost v/s k and then see for some point where we see that after that point the cost decreases comparatively at a less rate on the increasing k value.

- Plot of cost vs k value-



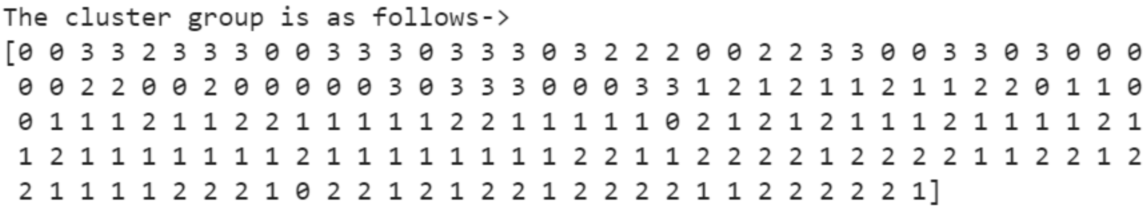
- From the graph I concluded to choose k = 4 as the best value of k for my k means clustering algorithm.

- So, after choosing the value of k as 4 and running the k means algorithm for 500 times (to get the best and the least cost) I get->



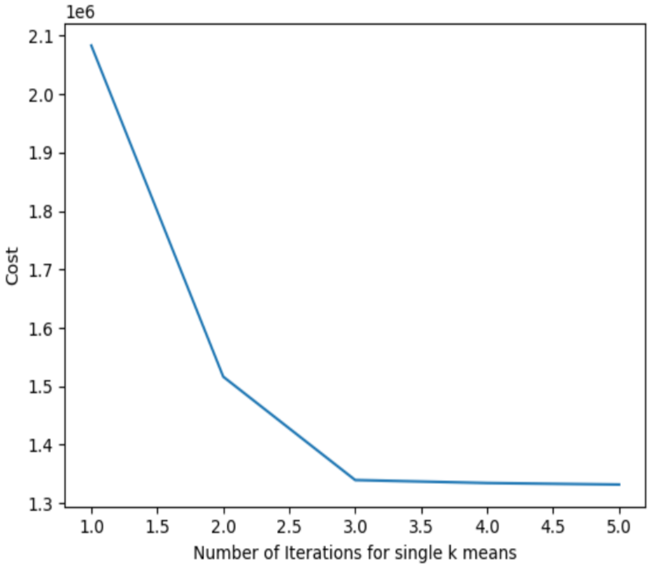
```
For k = 4, I get the following result->
Cost:  1331903.0622637179
```

- The final cluster group containing the final assigned centroids as an array containing 0 to (k-1) which signify the assigned k clusters to each dataset.

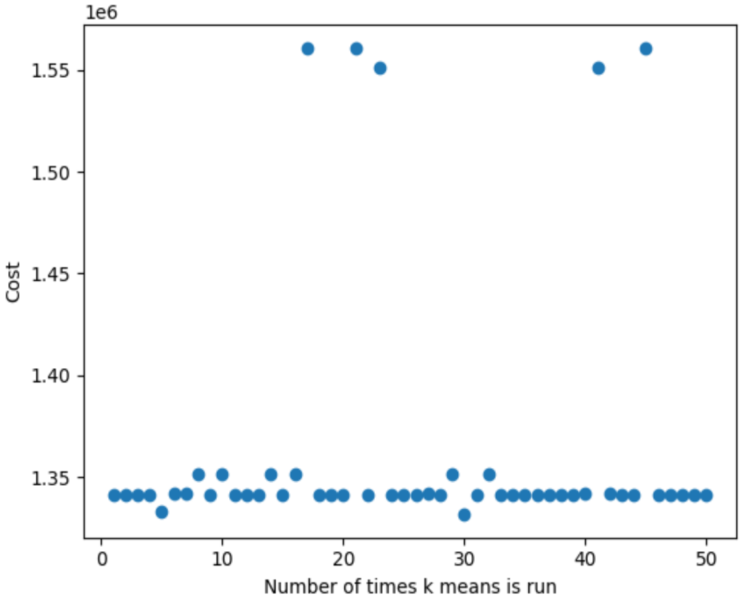


```
The cluster group is as follows->
[0 0 3 3 2 3 3 3 0 0 3 3 3 0 3 3 3 0 3 2 2 2 0 0 2 2 3 3 0 0 3 3 0 3 0 0 0
 0 0 2 2 0 0 2 0 0 0 0 0 3 0 3 3 3 0 0 0 3 3 1 2 1 2 1 1 2 1 1 2 2 0 1 1 0
 0 1 1 1 2 1 1 2 2 1 1 1 1 1 2 2 1 1 1 1 1 0 2 1 2 1 2 1 1 1 2 1 1 1 1 2 1
 1 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 2 2 1 1 2 2 2 2 1 2 2 2 2 1 1 2 2 1 2
 2 1 1 1 1 2 2 2 1 0 2 2 1 2 1 2 2 1 2 2 2 2 1 1 2 2 2 2 2 1]
```

- The plot for the cost v/s the iterations (the minimum cost)-



- The plot for the cost v/s number of times the k means is run-



[Future scope of Improvement:

- Silhouette Score- I can use the silhouette score as well with the elbow method to cross check the best value of k for my model.]