IIT ISM DHANBAD

By :- Abhinav Jha

Report

INTRODUCTION

We have to build our own Machine Learning Algorithm Library.

In which we have to implement Linear Regression, Polynomial Regression, Logistic Regression, K-nearest neighbor, and an N-layered Neural Network also K-means from **scratch.

Linear Regression

The training dataset comprises 20 feature columns with 50,000 entries. The purpose was to fit a straight line as an estimation.

I first used **Z-Score Normalization** to Normalize the dataset as the features differ in their range by a whole lot, which made our code faster, then I defined the model with compute cost in which I used the **mean squared error function** and gradient descent function to optimize the cost.

Then I defined the hyperparameters like learning rate(alpha), for deciding that I first started with a very small value that is 0.001 then increased it by 3 times until I found the right value for the model which is **0.9**. The model worked fine till alpha was **0.9** after that it started overshooting, at other smaller values the model was working fine but this was the learning rate at which the model worked the fastest.

I also initiated weights and bias randomly at the start and used seed(42) to save the value of weights and bias and then updated its values under the loop in the range number of iterations.

The initial values of weights and bias are:

Weights (theta): [0.49671415 -0.1382643 0.64768854 1.52302986 -0.23415337 -0.23413696

1.57921282 0.76743473 -0.46947439 0.54256004 -0.46341769 -0.46572975

0.24196227 -1.91328024 -1.72491783 -0.56228753 -1.01283112 0.31424733

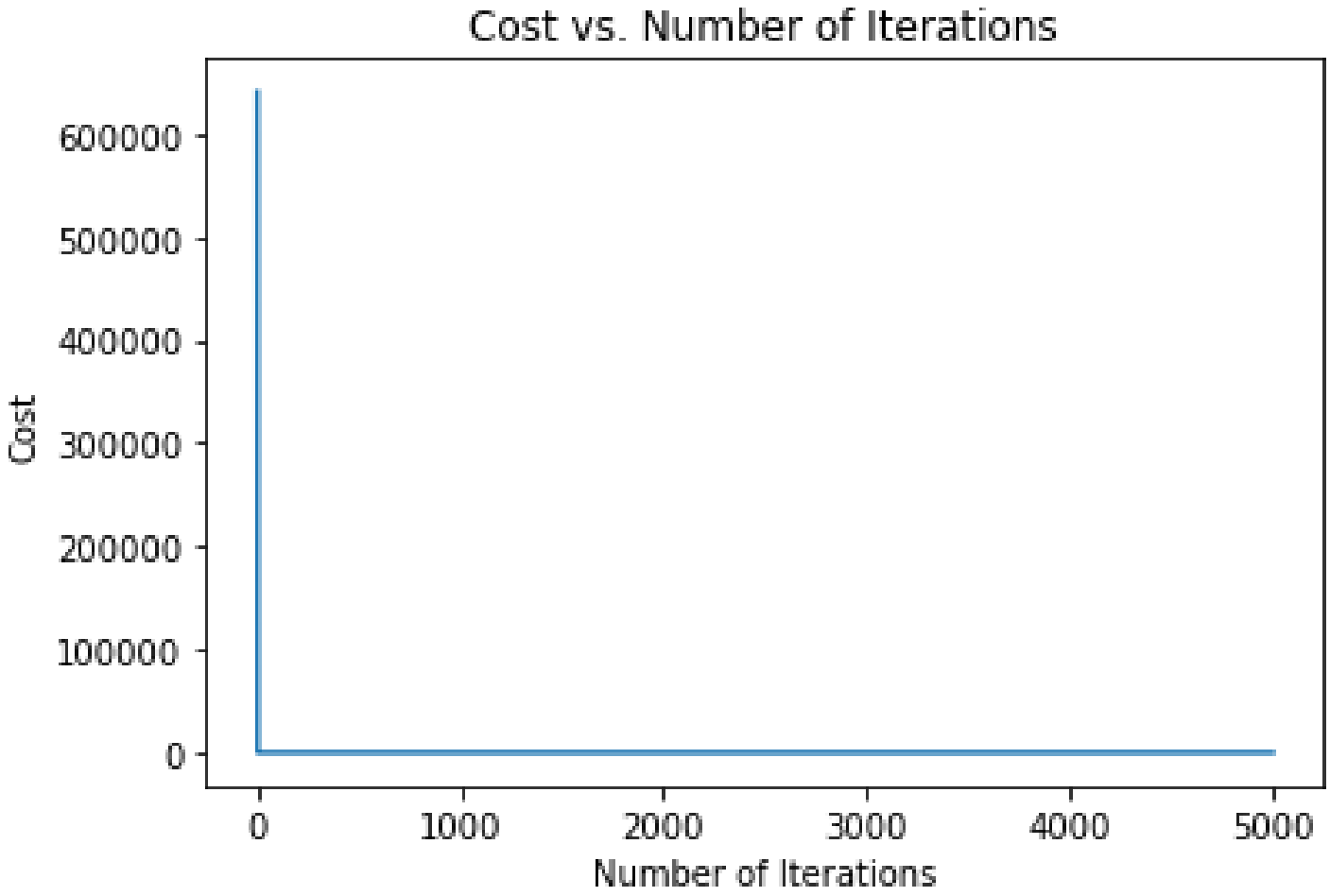
-0.90802408 -1.4123037]

Bias : [1.46564877]

The plot of cost with number of iterations was as following .

Starting cost : .**640960.1948177463**

Final cost : **0.005050994574172918**



You can watch the graph saturate.

The final Weights and Bias are

Weights (theta): [2.20195774e+01 3.31497109e+01 9.95541391e+00 6.05377498e+00

1.75468746e+02 3.29905115e+02 2.53980905e+02 9.02616587e+02

4.35808684e+02 8.38967472e+01 1.21226412e+02 1.60618608e+03

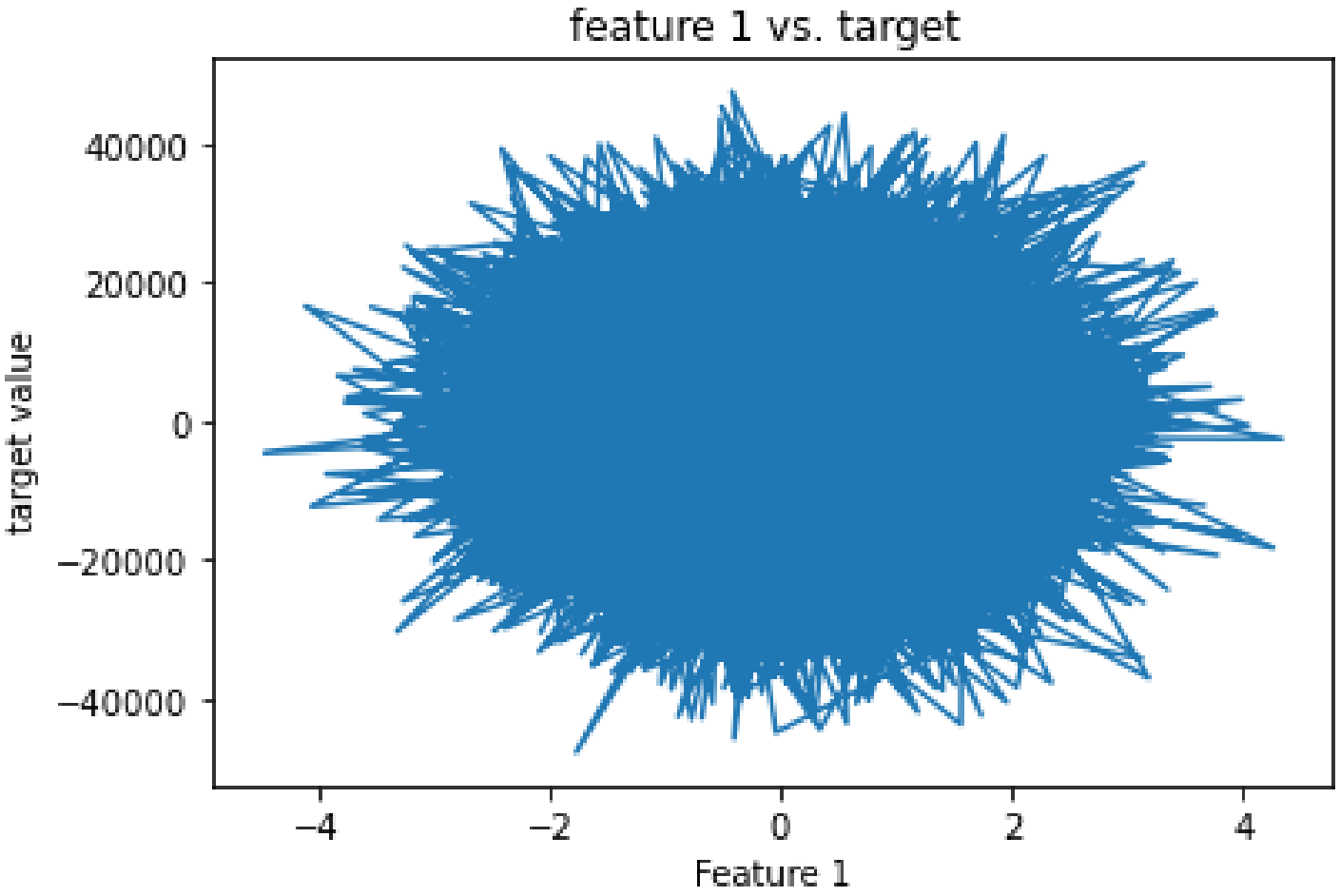
1.67667935e+03 1.31341727e+03 5.99392681e+02 1.05495141e+04

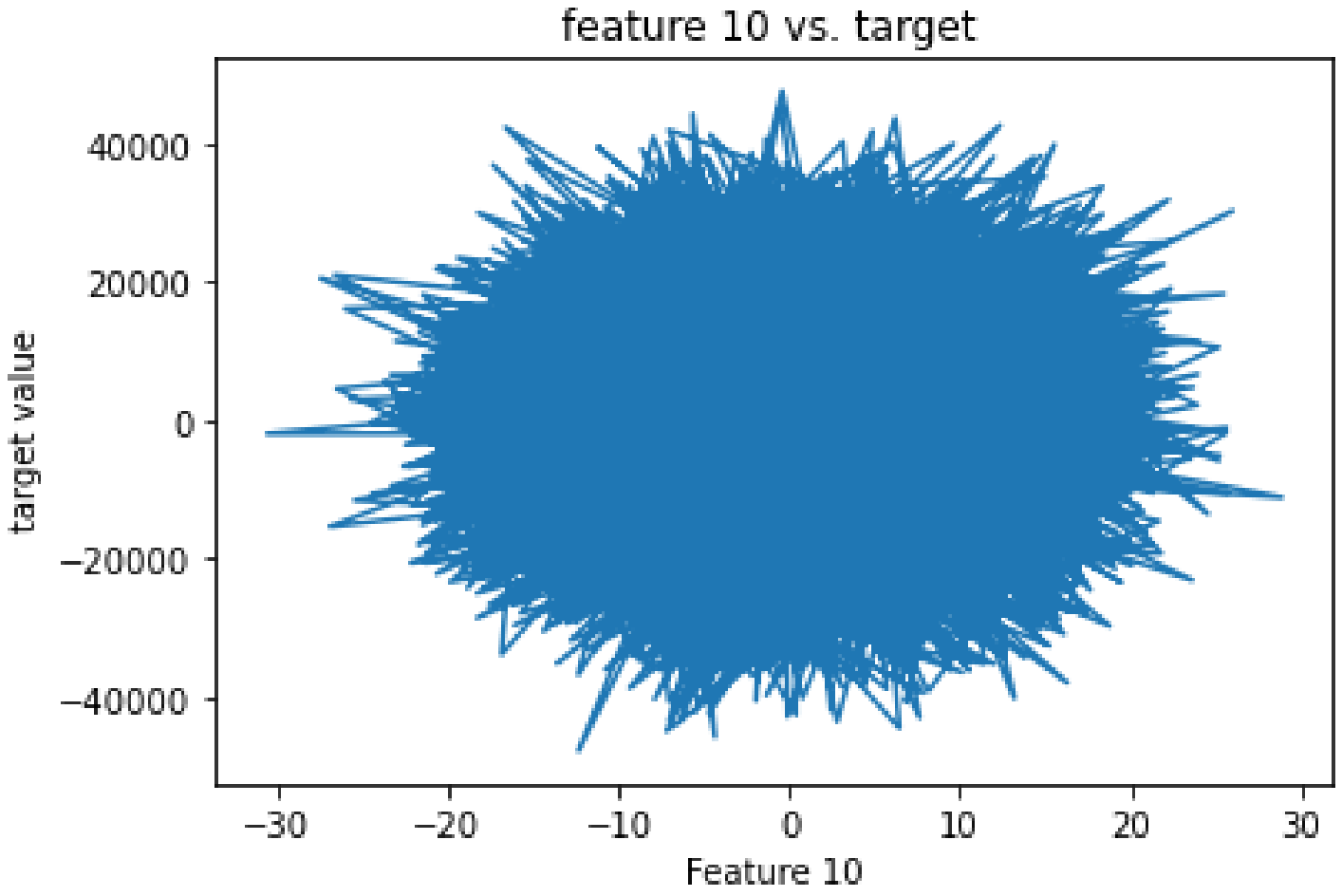
5.04448384e+02 6.78421739e+02 3.16485239e+03 2.30428026e+02]

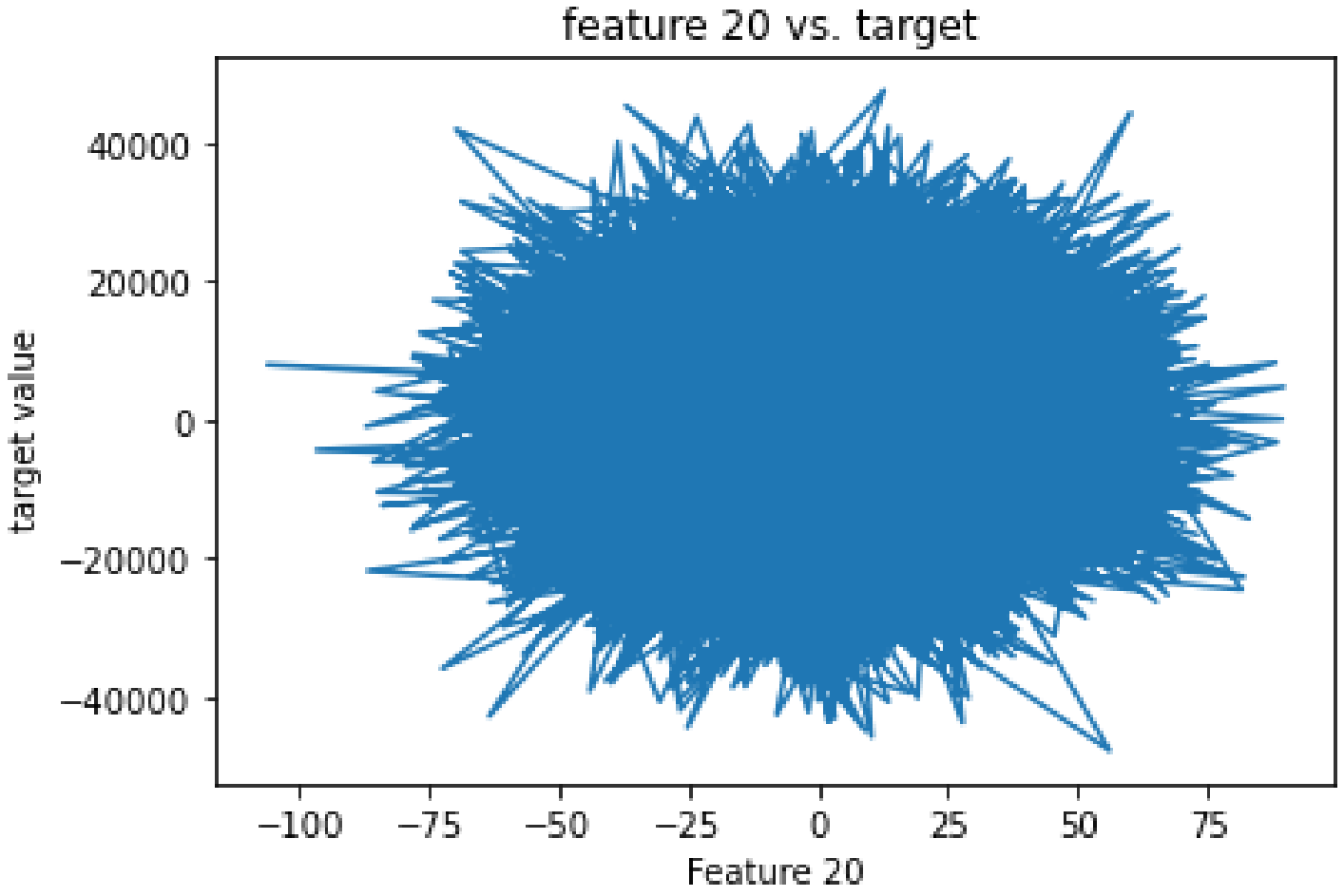
Bias : [60.36378862]

R2 Score: 0.999999999922914

The graph of singular feature vs target values is as follows :


feature 1 vs. target
40000
20000
0
−20000
−40000
target value
−4
−2
0
2
4
Feature 1


feature 10 vs. target
target value
40000
20000
0
−20000
−40000
−30
−20
−10
0
10
20
30
Feature 10


feature 20 vs. target
target value
40000
20000
0
−20000
−40000
−100
−75
−50
−25
0
25
50
75
Feature 20

Polynomial Regression

The training dataset comprises 3 columns of features with 50000 examples.

I first defined a function to create new features according to the degree of the polynomial we are working on by using the beggar's method in which I defined a, b, c with their sum equal to the degree(a + b + c = degree) and their range as

a lie in range (0 to degree +1)

b lie in range (0 to degree +1 - a)

c lie in range (0 to degree +1 -a -b)

and the general term as (feature 1 ^ a feature2 ^ b * feature3 ^ c)

and took the degree as input.

Then I normalized the data by Z-score Normalization, i also specified to take standard deviation as 1 where it was coming out be zero as the data already differ too much on the range, and making new features with their exponential form enhances this making the code slow.

Then I defined the model in which I randomly initiated weights and bias and saved them using seed(42) then I defined the cost function (mean squared error function) and gradient descent since we were working on polynomial regression in which higher degree polynomials usually overfits the dataset I regularize the dataset by L-2 regularization.

To define the hyperparameters, I started with a learning rate of 0.001. I then tried 0.01 and finally found the **appropriate value which was 0.1,** after which overshooting started.

Then for **Lamda (Regularization parameter)**, I tried different values from 0.000001 to 1000 at a higher value of Lamda than 1 the cost was too high and the r2 score was not that good, the same with values just below 1 so the **best choice was 0.001** at which the model does not overfit the data and gives us the best possible cost and r2 score.

For choosing the optimal degree I ran the model for different values of degree and got the best fitting and r2 score on degree 6.

R2 scores with different degrees are as follows :

Degree: 1 - R-2 score: 0.0003433988351401185

Degree: 2 - R-2 score: 0.40964066886997874

Degree: 3 - R-2 score: 0.42030417432893197

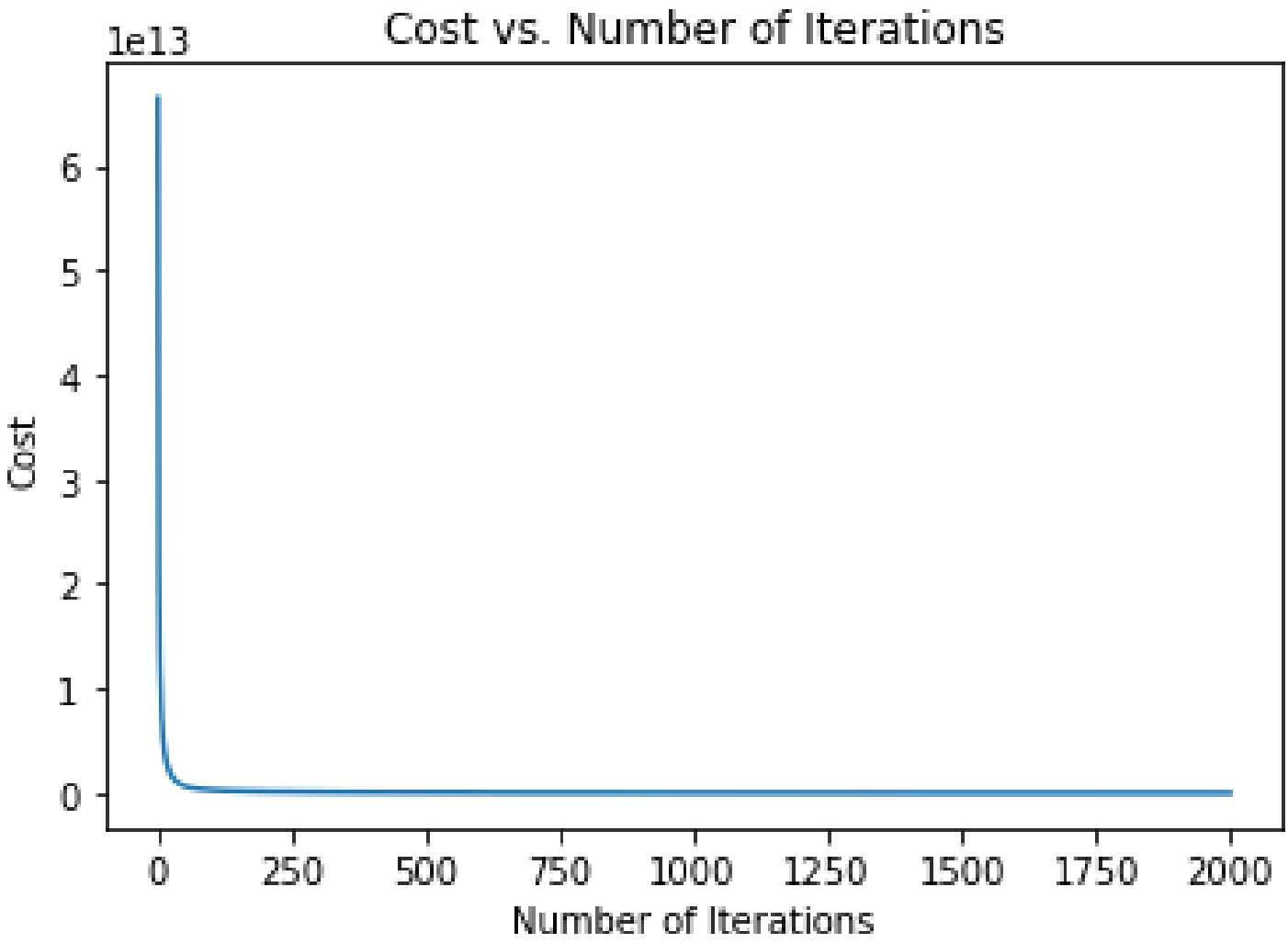
Degree: 4 - R-2 score: 0.9366560685851311

Degree: 5 - R-2 score: 0.9486567739754266

Degree: 6 - R-2 score: 0.9999818719512055

The plot of cost vs iterations is as follows

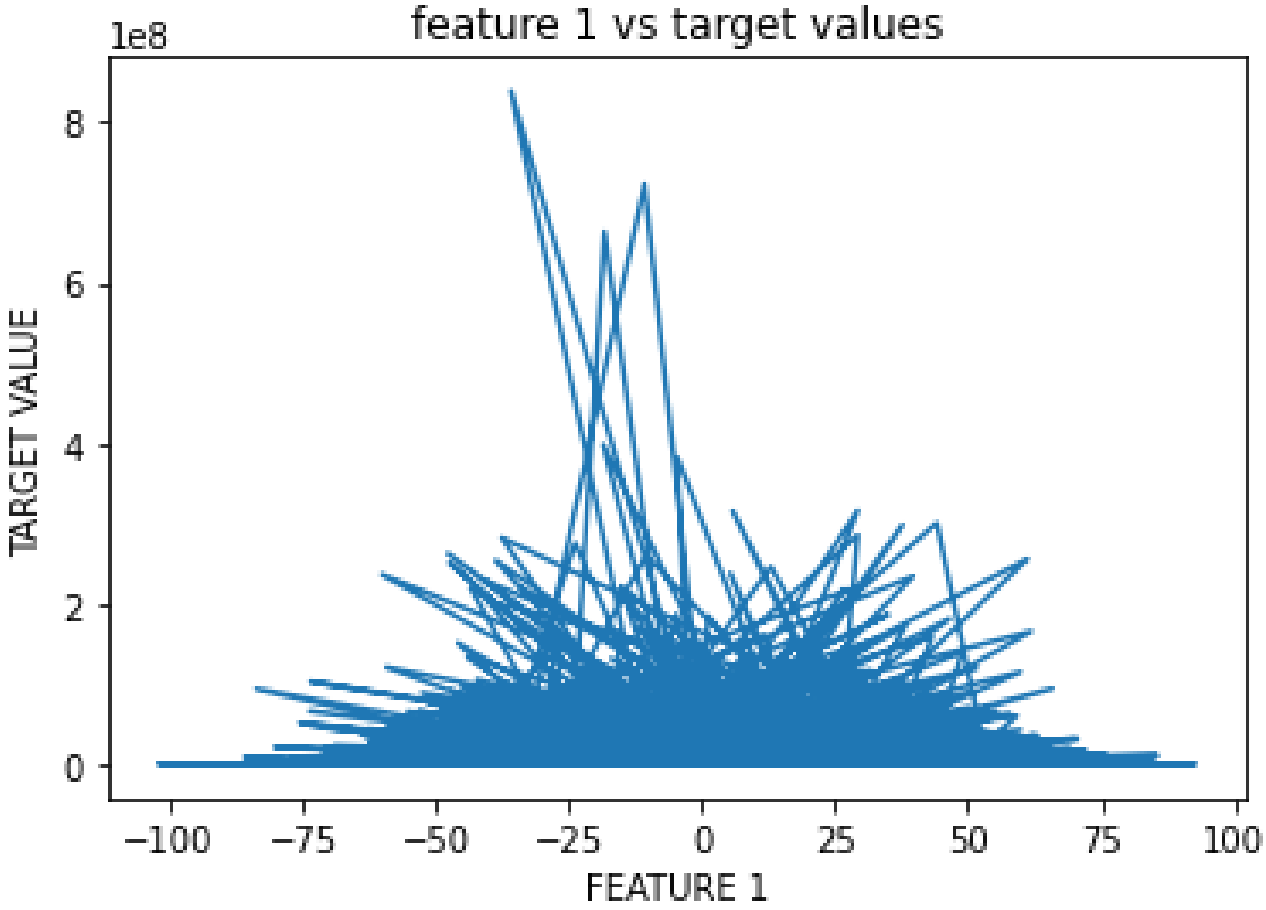
Final Cost: 1180838378.4192524

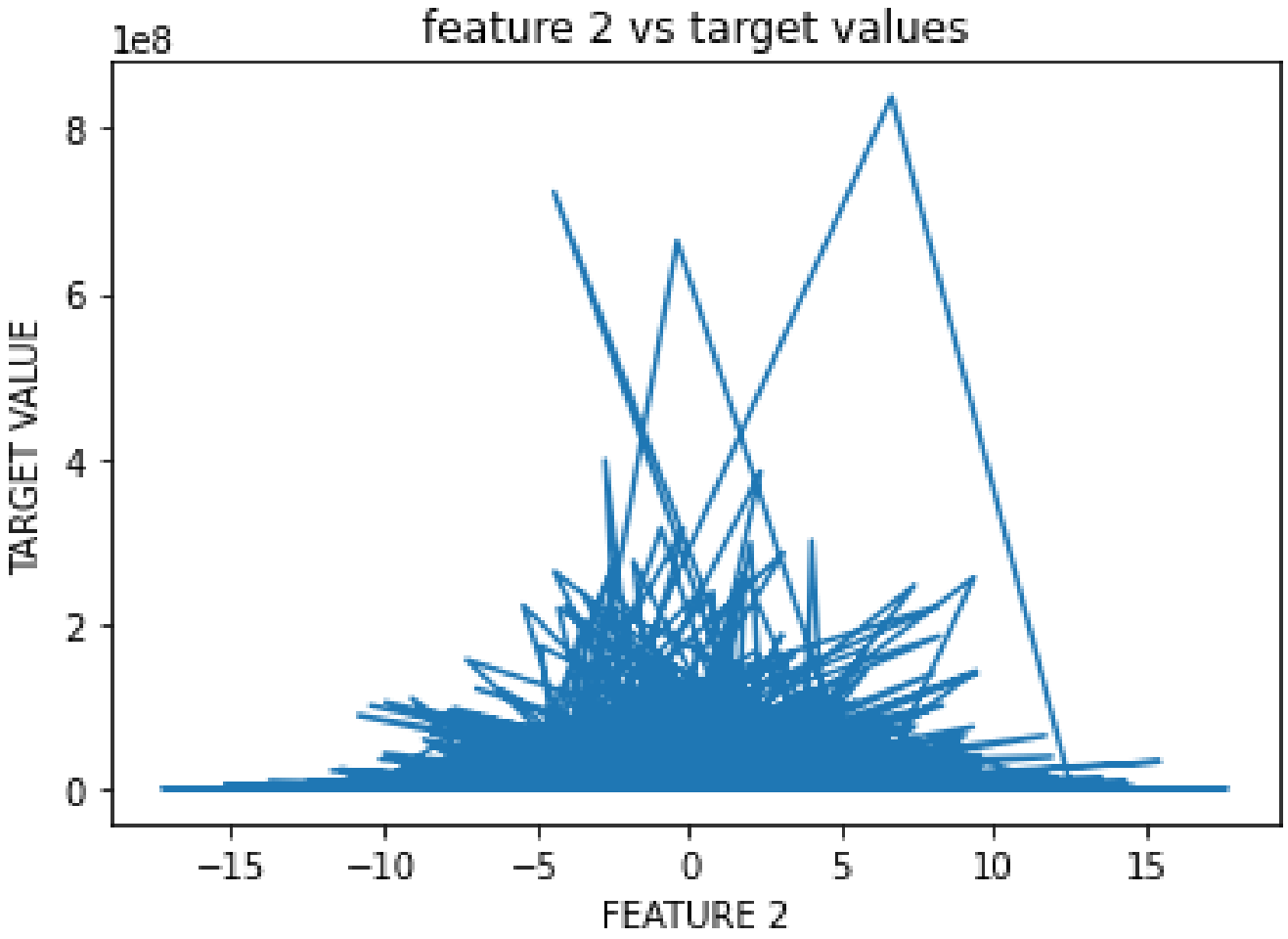


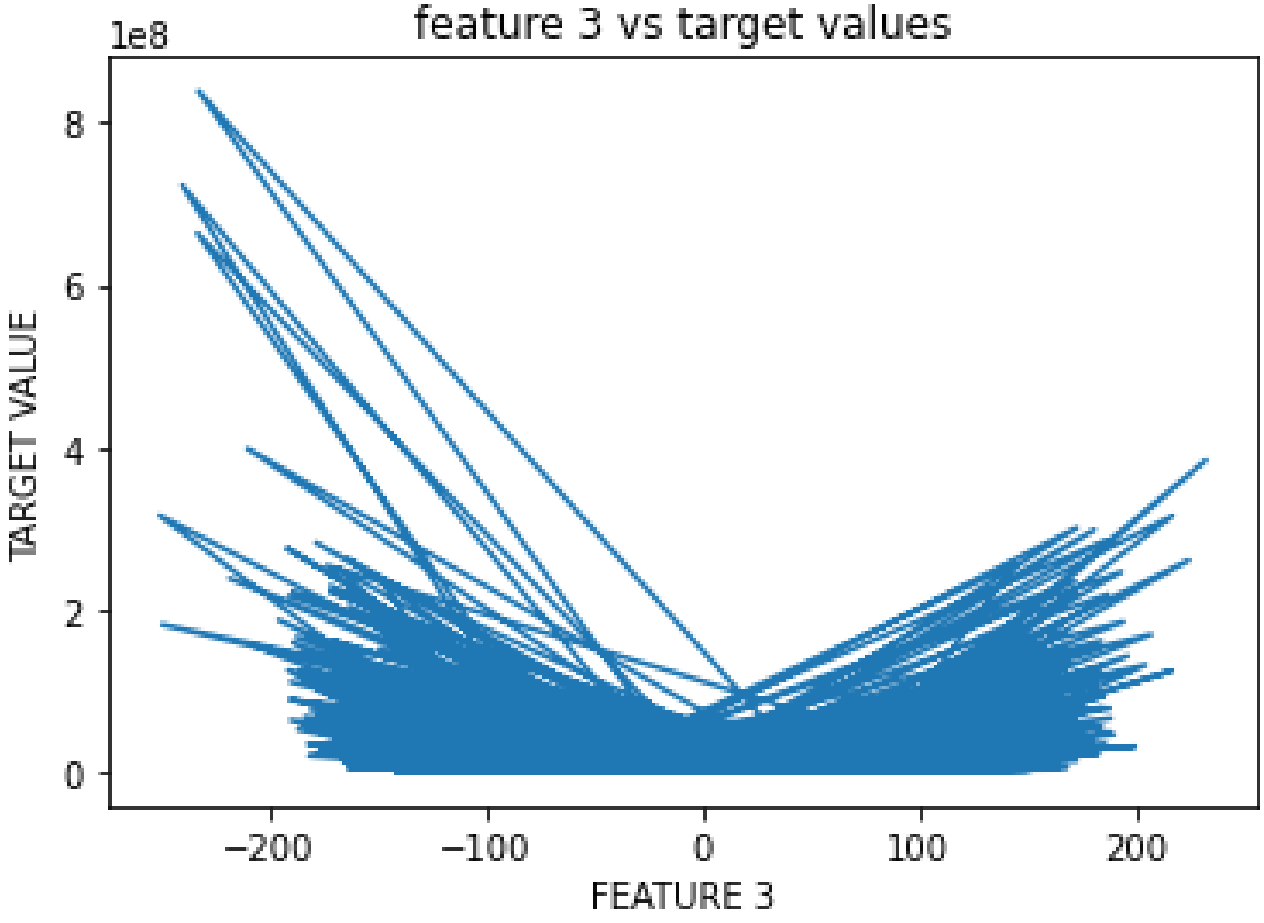
You can watch the graph saturate.

R-2 score: 0.9999818719512055

The graph of every feature with the target value is as follows


feature 1 vs target values
1e8
8
6
4
2
0
TARGET VALUE
−100
−75
−50
−25
0
25
50
75
100
FEATURE 1


feature 2 vs target values
1e8
8
6
4
2
0
TARGET VALUE
−15
−10
−5
0
5
10
15
FEATURE 2

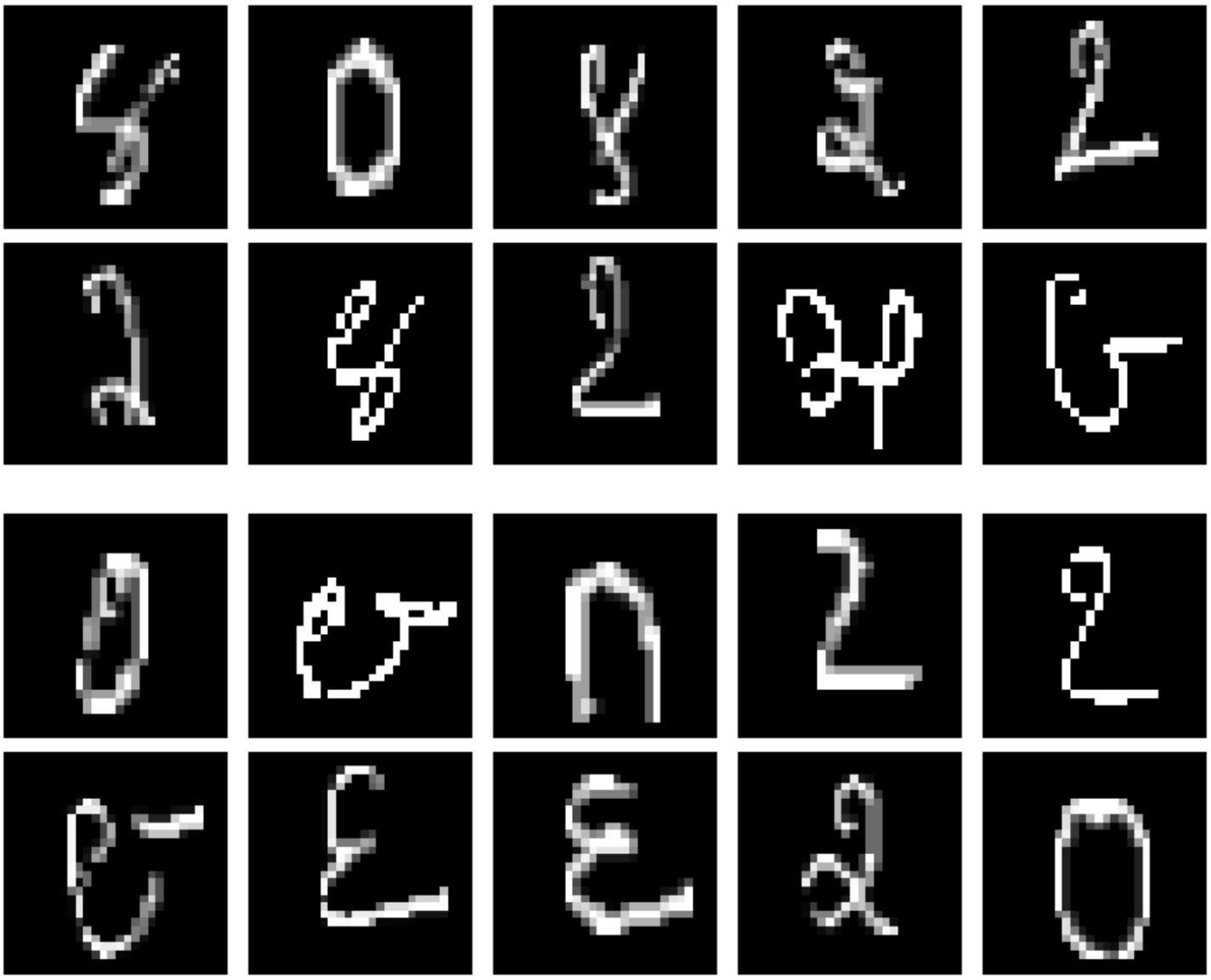

feature 3 vs target values
1e8
8
6
4
2
0
TARGET VALUE
−200
−100
0
100
200
FEATURE 3

The graphs are random and do not give any idea about what should be the degree of the curve.

Classification

The dataset comprises 784 pixels with 30000 examples in each, with the target values in the range 0-9 with 10 different values.

We have to make a classifier using logistic regression, KNN, and neural network.



Logistic Regression

Since the course was about binary classification by logistic regression and for multiple classes we have to use softmax I first thought we had to use softmax only then I came to know that we have to one vs all algorithm, so i studied that by youtube first.

I started with first normalizing the data set by Min Max normalization in which is first subtracted the minimum value and then divided by the maximum value of the same column, since some columns have only 0 as data I added epsilon(a very small number with the maximum 1e-8) as dividing by 0 is not possible.

Then I split the data into training and cross-validation sets in the ratio of 80:20.

Then I defined the cost function (sigmoid function) and used mini-batch gradient descent as it is faster and gives better accuracy.

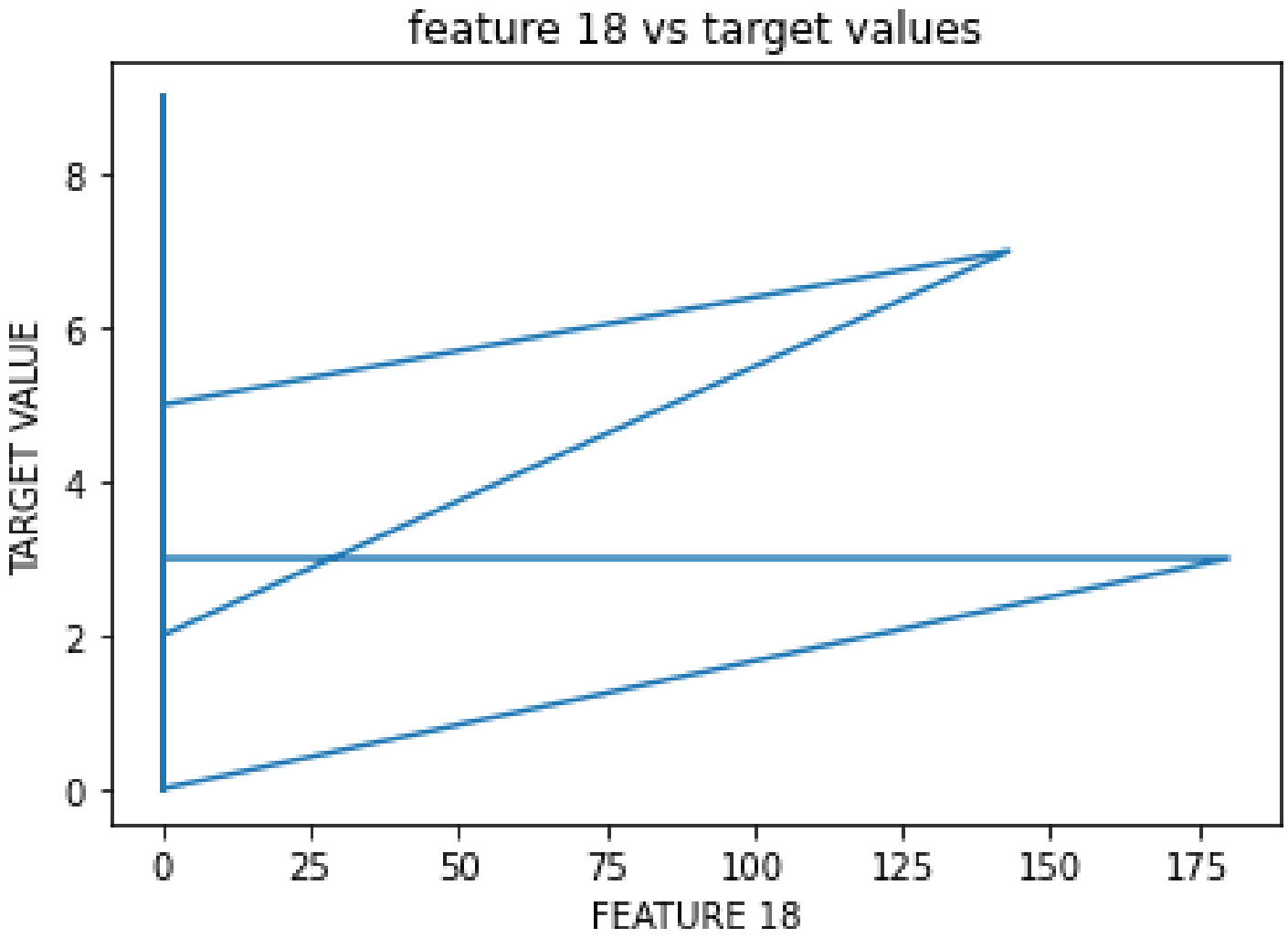
Since we have to classify in multiple classes I used the one vs all algorithm in which I created unique classes with the use of np. unique according to the y values and ran a loop in the range of unique classes in which it takes different values from it for example it took 5, it marked it as 5 and every other value from 0-9 as 0 then it did the binary classification between 0 and 1.

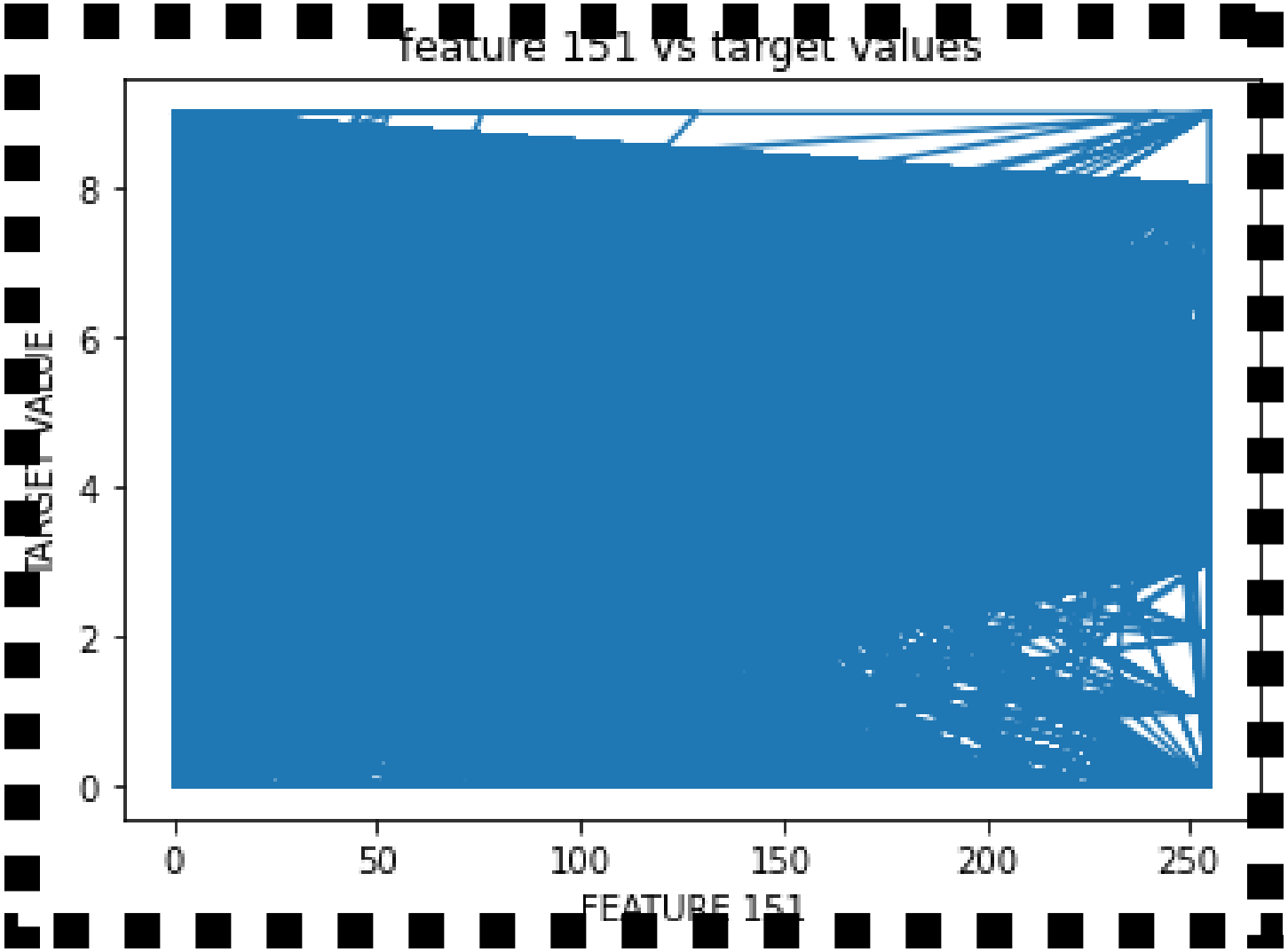
For hyperparameter selection, I started with 0.001 and then took 0.01, and then moved to 0.1 with 0.5 as following, then i **chose 0.5 as the optimal value** for it as it does not overshoot and is faster than the rest of the smaller values.

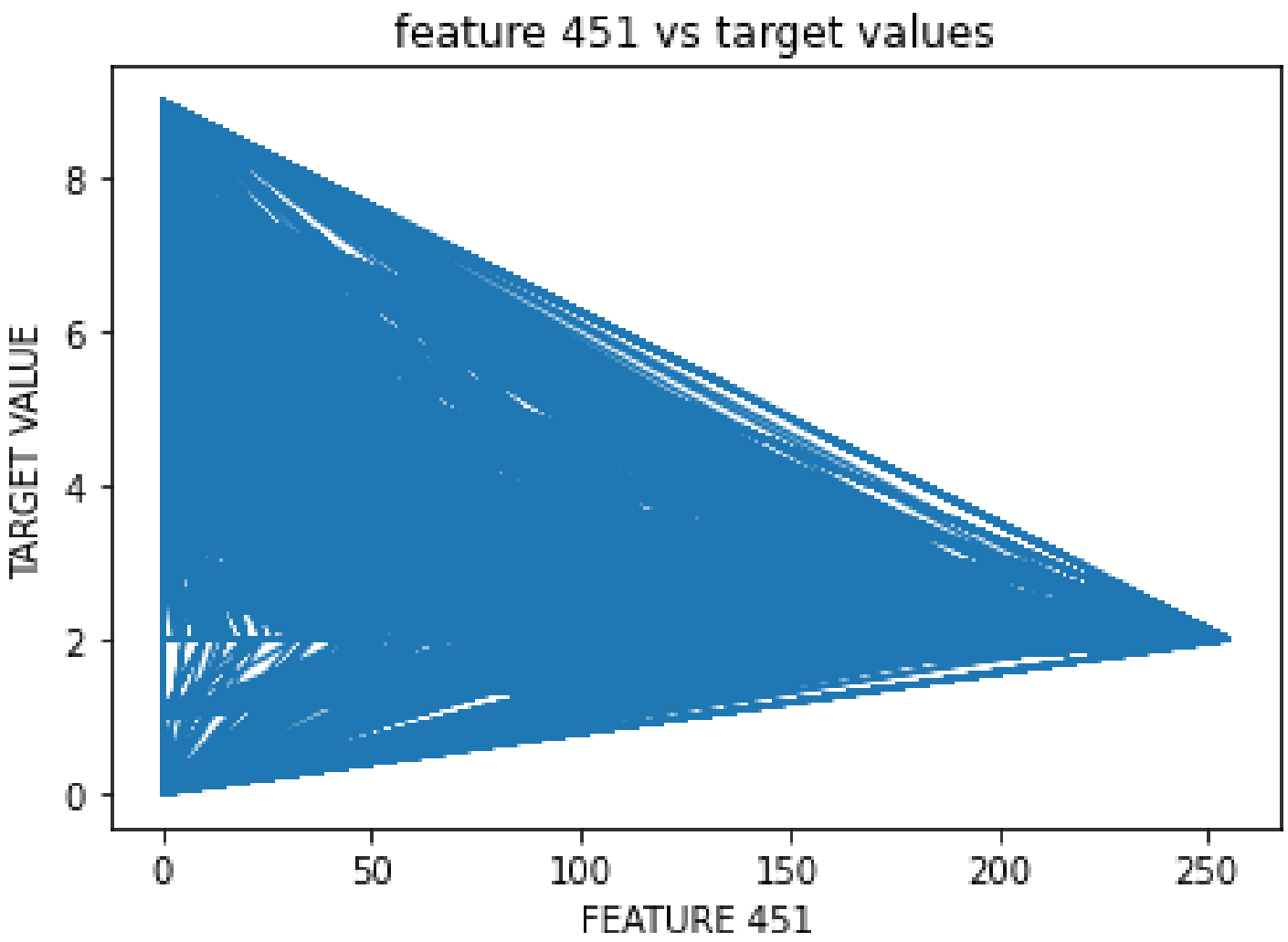
For the **batch size, I took the common value of 64.**

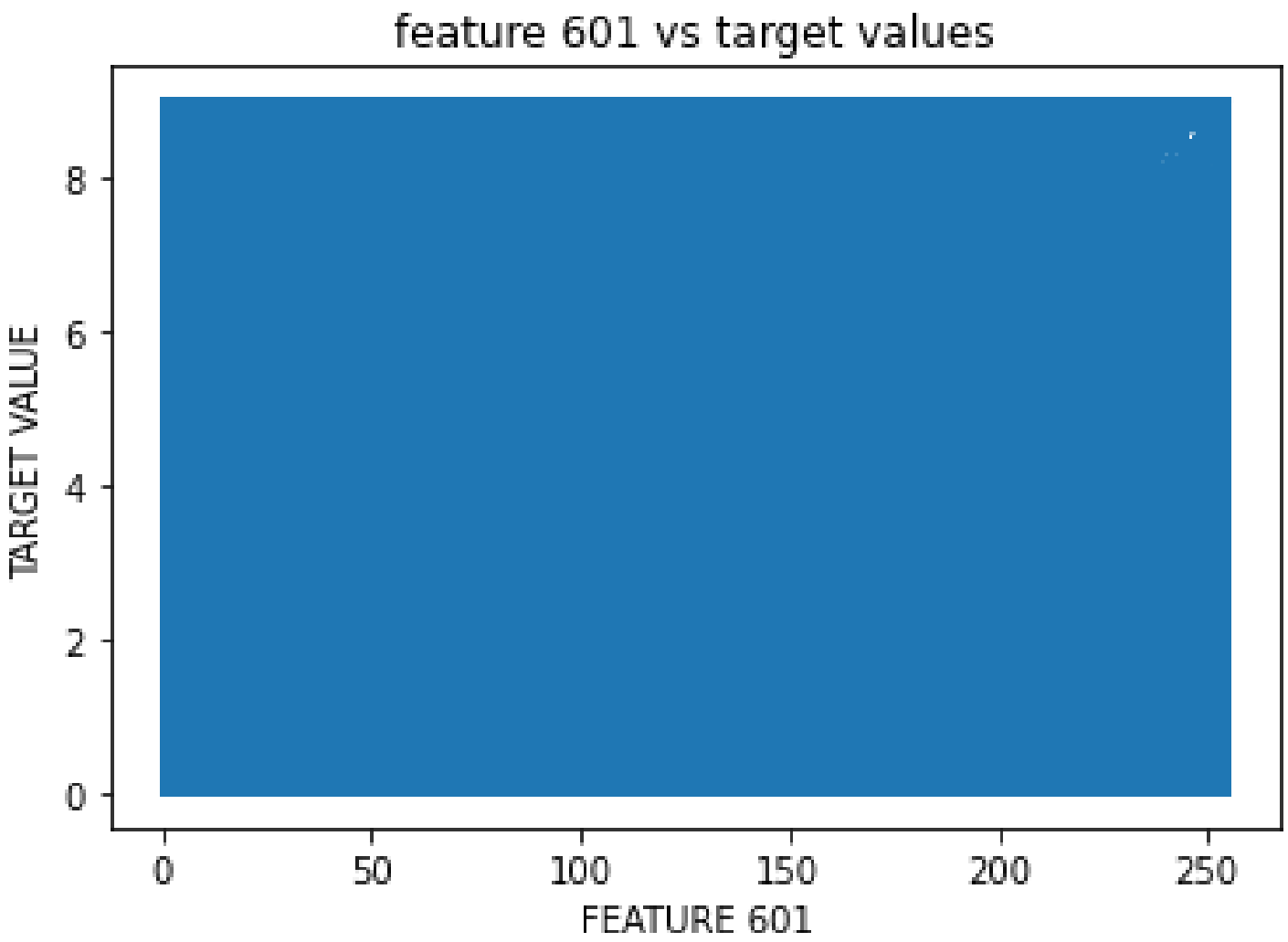
Then I calculated the accuracy of the model by testing it on the cross-validation set and finding its accuracy by np. mean(y_pred == y_test), which gives the accuracy as : **0.968333333333333.**

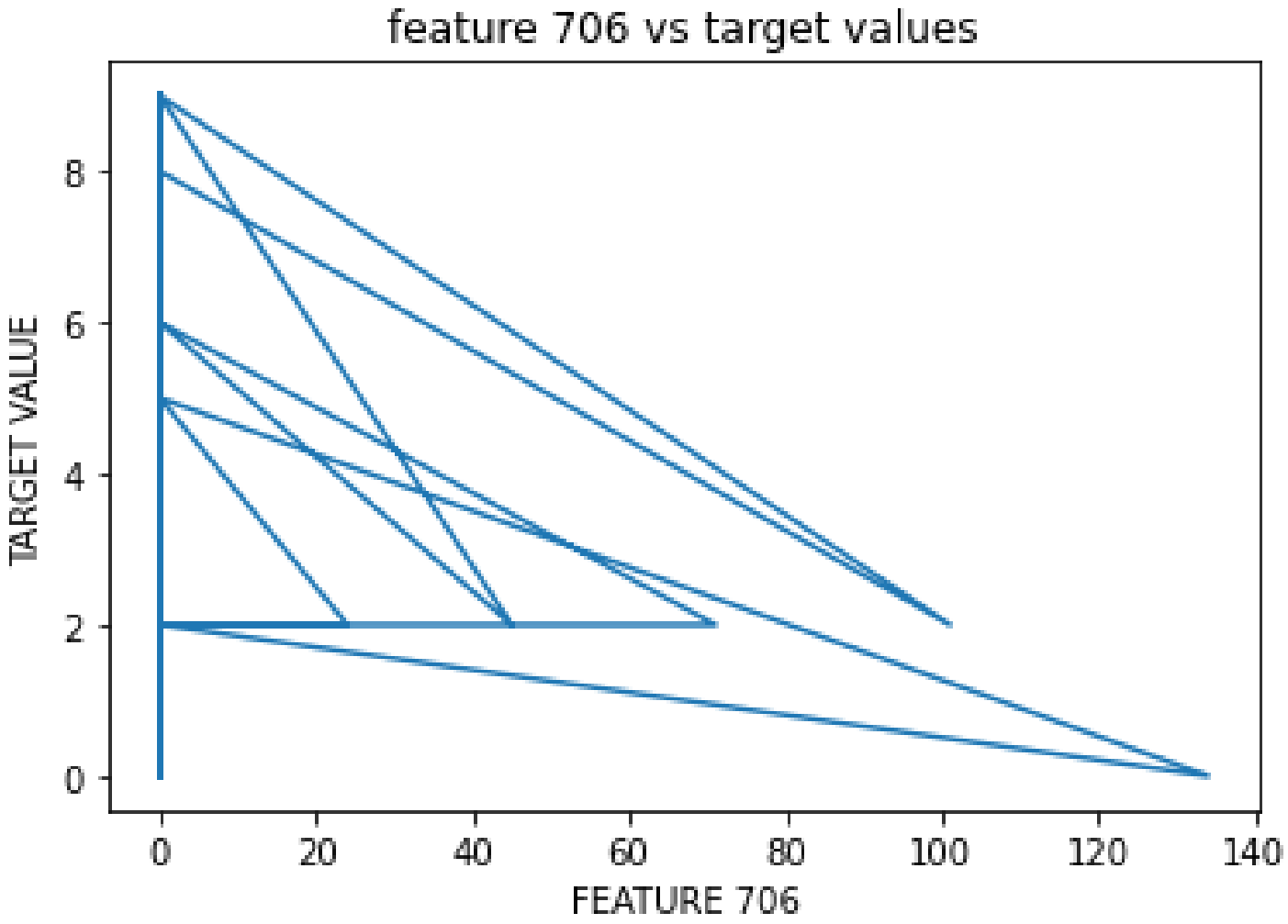
The graph of different pixels vs target values is as follows :


feature 18 vs target values
TARGET VALUE
FEATURE 18
0
2
4
6
8
0
25
50
75
100
125
150
175


feature 151 vs target values
TARGET VALUE
0
2
4
6
8
0
50
100
150
200
250
FEATURE 151


feature 451 vs target values
TARGET VALUE
FEATURE 451
0
2
4
6
8
0
50
100
150
200
250


feature 601 vs target values
TARGET VALUE
0
2
4
6
8
0
50
100
150
200
250
FEATURE 601

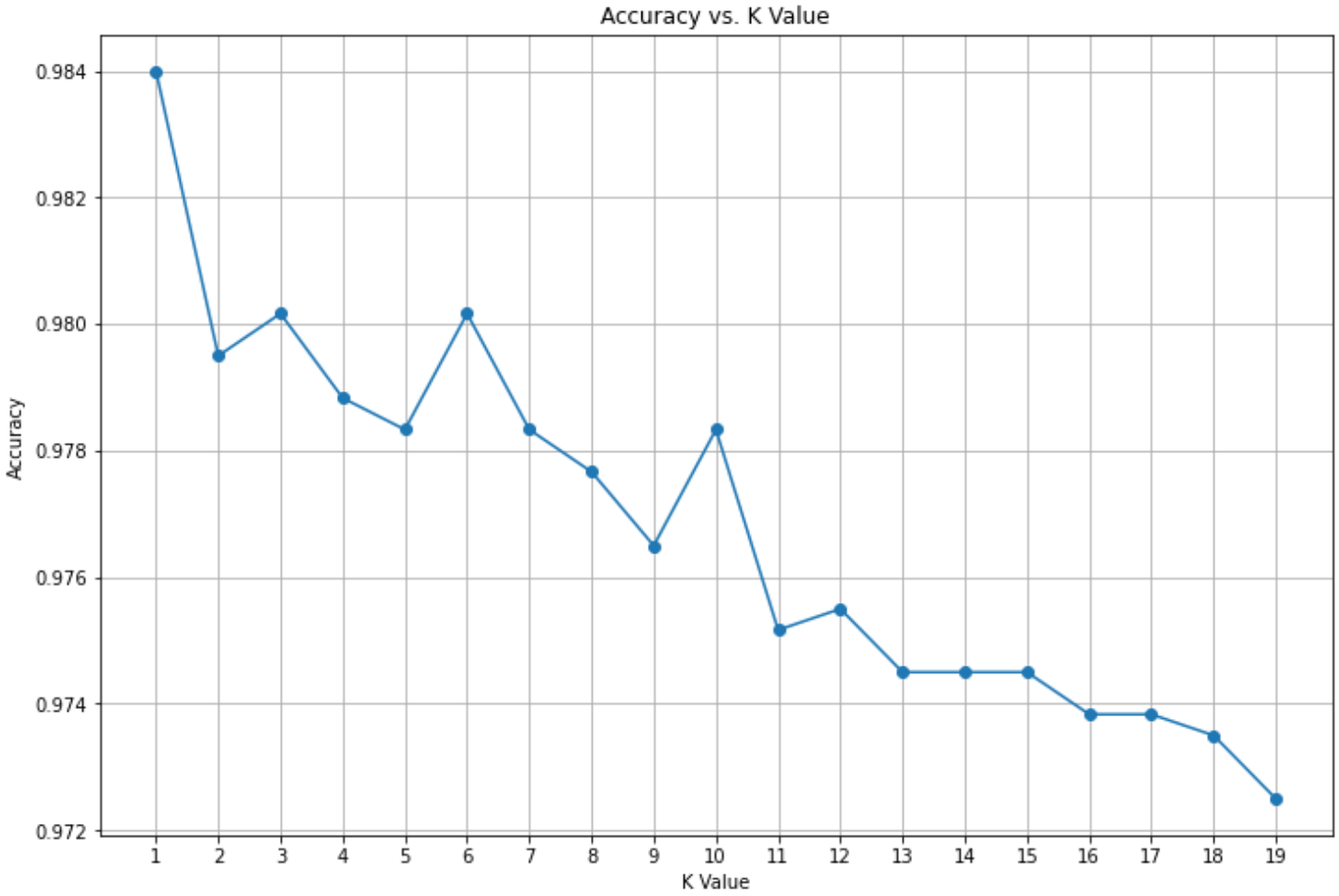


This shows that some pixels only contain zero values(the pixels at the extreme corner) while some contain only a few values and rest as zero while others more specifically in the middle contain many pixel values.

KNN

I first started by splitting the dataset into train and test in the ratio of 80:20, then i defined the Euclidean distance function with a pre calculate function used to calculate the distance between the training and testing points I then defined the predict function in which it will take the value of k and then perform KNN.

For selecting the value of k I plot the graph of accuracy vs value of k for k in the range 1-20



Giving the maximum value at **k =1 as 98.4% accuracy, and after that at k = 3 was the maximum accuracy of 98.04%.**

I chose k = 3 .

The reason for maximum accuracy at k=1 is not exactly clear but I think it is due to the similarity between the training and testing data set.

Neural Network

I first normalized the data using Z-Score Normalization: and then split the dataset into training and testing in the ratio of 80:20.

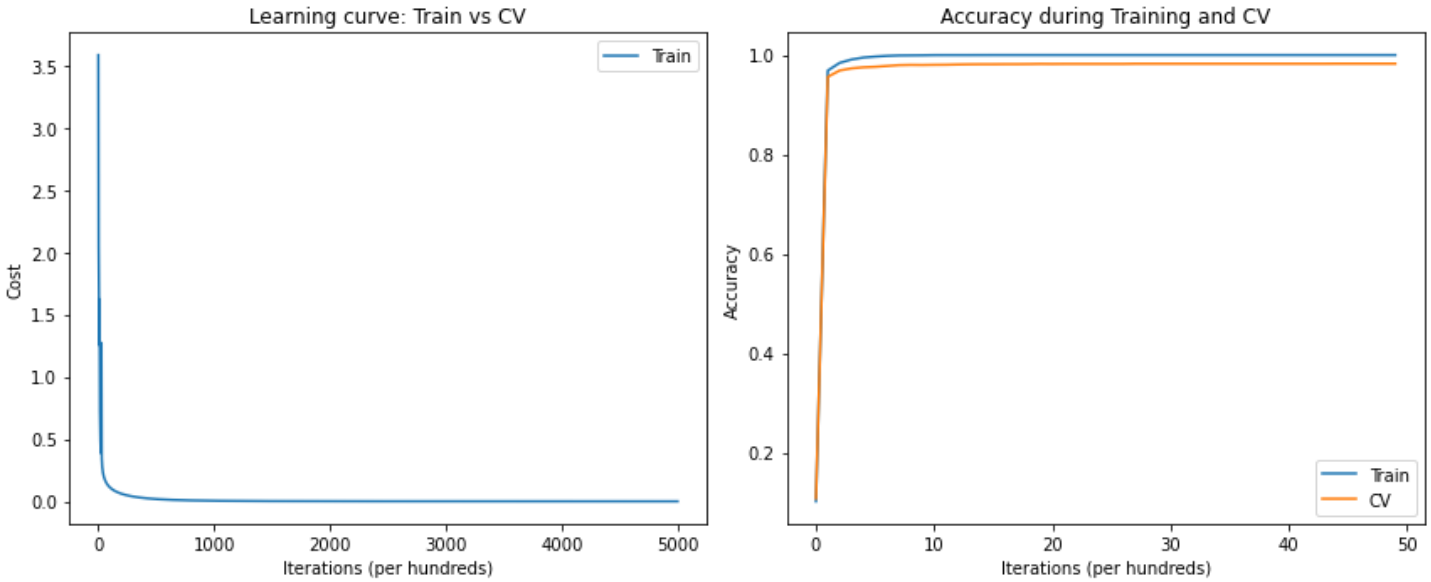
Then I defined the different functions used in the hidden layer(Relu) and output layer(softmax), followed by randomly initializing the parameters as He initialization.

Then I defined the forward propagation, compute cost, backward propagation, and update parameters functions and implemented the batch gradient descent function to find the minima of the cost function.

The model was trained for n layered neural network for which you can give the input of the number of hidden layers, with neurons depending upon number of hidden layers as

layer_dims = [input_layer_size] + [32 * (2 ** max(0, num_hidden_layers - i - 1)) for i in range(num_hidden_layers)] + [output_layer_size]

The model predicts the training dataset with 100% accuracy and the cross-validation set with 98.34 percent accuracy but the problem here is that it is taking 40 mins to run and 5000 iterations to do that.



Final Cost: 0.00028768212410534106

```
Iteration 0: Train Cost: 3.5837662395977725, Train Accuracy: 0.10175, CV Accuracy: 0.10766666666666666
Iteration 100: Train Cost: 0.11177275759427353, Train Accuracy: 0.9687916666666667, CV Accuracy: 0.9561666666666667
Iteration 200: Train Cost: 0.060938294155138094, Train Accuracy: 0.9842083333333334, CV Accuracy: 0.9688333333333333
Iteration 300: Train Cost: 0.03867988426849877, Train Accuracy: 0.991, CV Accuracy: 0.9731666666666666
Iteration 400: Train Cost: 0.026244890756415434, Train Accuracy: 0.9949583333333333, CV Accuracy: 0.9753333333333334
Iteration 500: Train Cost: 0.018459288490675484, Train Accuracy: 0.9969166666666667, CV Accuracy: 0.9763333333333334
Iteration 600: Train Cost: 0.0133938098949359, Train Accuracy: 0.998375, CV Accuracy: 0.978
Iteration 700: Train Cost: 0.010027972472295955, Train Accuracy: 0.9992083333333334, CV Accuracy: 0.9796666666666667
Iteration 800: Train Cost: 0.00771213591397623, Train Accuracy: 0.999375, CV Accuracy: 0.9801666666666666
Iteration 900: Train Cost: 0.006032495830198716, Train Accuracy: 0.9995416666666667, CV Accuracy: 0.98
Iteration 1000: Train Cost: 0.00484679735552591, Train Accuracy: 0.999875, CV Accuracy: 0.9803333333333333
Iteration 1100: Train Cost: 0.003975373054262511, Train Accuracy: 0.9999166666666667, CV Accuracy: 0.9805
Iteration 1200: Train Cost: 0.003315090035036740 6, Train Accuracy: 0.9999166666666667, CV Accuracy: 0.981
Iteration 1300: Train Cost: 0.0028085212255694316, Train Accuracy: 0.9999166666666667, CV Accuracy: 0.9813333333333333
Iteration 1400: Train Cost: 0.002412401042306014, Train Accuracy: 0.9999583333333333, CV Accuracy: 0.9815
Iteration 1500: Train Cost: 0.002100820323018277, Train Accuracy: 1.0, CV Accuracy: 0.9815
Iteration 1600: Train Cost: 0.0018512910558861535, Train Accuracy: 1.0, CV Accuracy: 0.9816666666666667
Iteration 1700: Train Cost: 0.0016484311575417144, Train Accuracy: 1.0, CV Accuracy: 0.9816666666666667
Iteration 1800: Train Cost: 0.0014813860077107707, Train Accuracy: 1.0, CV Accuracy: 0.9818333333333333
Iteration 1900: Train Cost: 0.0013416106164748224, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2000: Train Cost: 0.0012228903907017335, Train Accuracy: 1.0, CV Accuracy: 0.982
Iteration 2100: Train Cost: 0.0011213154961574878, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2200: Train Cost: 0.0010335215431484 14, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2300: Train Cost: 0.0009570684849974591, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2400: Train Cost: 0.0008898703733843558, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2500: Train Cost: 0.0008304431660768935, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2600: Train Cost: 0.0007775727291860632, Train Accuracy: 1.0, CV Accuracy: 0.9821666666666666
Iteration 2700: Train Cost: 0.0007302586850364821, Train Accuracy: 1.0, CV Accuracy: 0.9825
Iteration 2800: Train Cost: 0.0006877279940389323, Train Accuracy: 1.0, CV Accuracy: 0.9825
Iteration 2900: Train Cost: 0.0006493105813233564, Train Accuracy: 1.0, CV Accuracy: 0.9825
Iteration 3000: Train Cost: 0.0006145034291325694, Train Accuracy: 1.0, CV Accuracy: 0.9825
Iteration 3100: Train Cost: 0.0005828300669989826, Train Accuracy: 1.0, CV Accuracy: 0.9825
Iteration 3200: Train Cost: 0.0005539054104286355, Train Accuracy: 1.0, CV Accuracy: 0.9825
Iteration 3300: Train Cost: 0.0005274292905192338, Train Accuracy: 1.0, CV Accuracy: 0.9825
```

You can watch the cost saturate.

K means

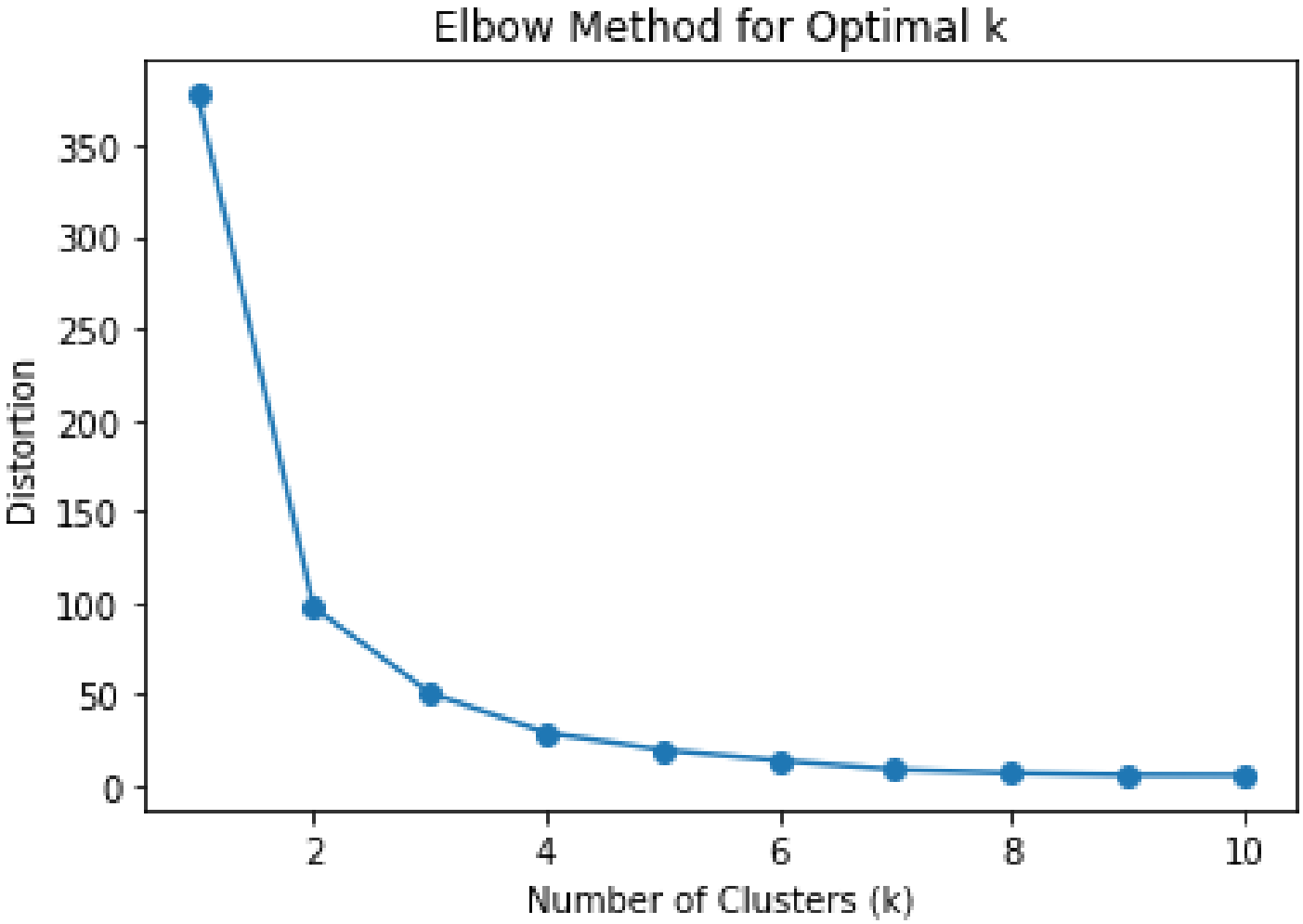
The dataset contains 13 features and 178 examples.

I first normalized the data by **Z-Score Normalization** and defined the distortion function following with k means model in which centroid will be updated till new centroid = old centroid , with initial centroids generated randomly .

I defined the model to run for k defined in maximum clusters and print the distortion for optimal k only.

I used the elbow method to find the optimal k .

Here's the graph of k vs distortion.



I chose k=6 as my optimal k and found distortion at k =6 for different normalization.

Without Normalization: **Final Distortion: 647326.0020260847**

With Z-Score Normalization: **Final Distortion: 13.907104727986646**

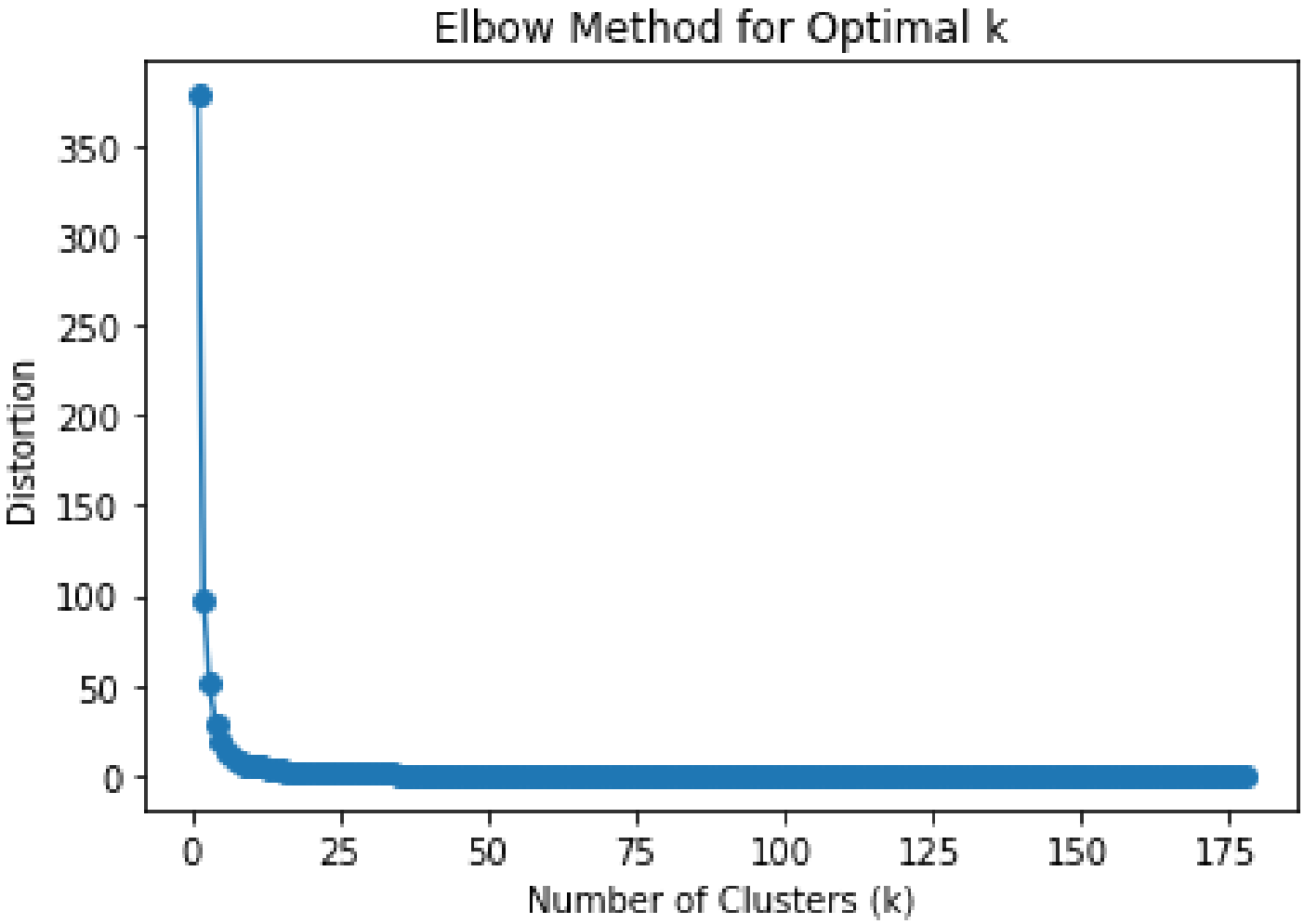
With Min Max Normalization: **Final Distortion: 0.22935303359767736**

The model working properly is identified by running the model **for k = 178** will give distortion as 0.

Final Distortion: 0.0

and it is verified.

The graph for value of k from 0 to 178 is



Human Error

I tested myself identifying the dataset and I cannot predict the dataset with 100 percent accuracy , so with that i think my neural network is working well.

In [82]: