assignment2 ml

2

a

The initial problem that was faced was when computing the inverse matrix when calculating the $w = (X^T * X)^{-1} * X^T y$

Generally when cacluating the inverse matrix for A, the usual caclulation is A = $\frac{adjointofA}{det(A)}$

So for singular matrixes, the det(A) = 0 and these leads to error. So there are other ways of calculating the inverse of matrix that work for singular matrixes as well and that is using the Singular value decomposition. So svd of matrix is written as: UAV^T

Instead of the the np.linalg.inv we use the np.linalg.pinv that implements this functionality for inverse caclulations.

Also after fixing this , its observed that when the number of features increases the error varies:

features| error
10 0.280539137865
75 0.265191182315
100 0.261729170317
150 0.199932942545
200 0.194947202658
250 0.18211574321
383 0.168788097724

As you could the see the error was decreasing but slightly as the number of features increased. After 200 features the rate of decrease in error has been reduced.

b

There are two ways to do the multiple splits. One is using random data points from the dataset or also we can do a cross fold validation. I have choosen to do the later. The reasons are stated below:

Firstly we can specify how many folds the data should be folded.

In a cross fold method, every point in the dataset has the chance to be part of the test set and training set , but not at the same time.

Since given the freedom of choosing the number of folds we have the advantage of controlling the bias and variace of the prediction errors.

The more the number of folds: we have more bias and less variance.

The less the number of folds we have less bias and more variance. Because the error decreases as the trainset increases.

This way we can control the bias and varaince. A better testing would be either ten fold or 5 fold which tries to manintain a balance with bias as well as varaince.

c

There are many ways we can do this :

1.) Supply through the command line: We specify the paramters at the command line with algorithm that we wish to run.

2.) specify in the regressionAlgs dictionary: But in this approach both the paramters and the model is tightly coupled.

3.) Use the seperate params{} dictionary to store the parameters for each iteration as shown below:

```
params = { 'Mean': [{}],
        'Random':[{}],
        'FSLinearRegression': [ {'features': range(50)} , {'features':range(75)}, {'features':range
        'RidgeRegression': [{'regularizer': 0.01}, {'regularizer': 0.1}, {'regularizer': 1.0} ],
        'SGDRegression':[{'alpha':0.01}],
        'BatchRegression':[{'feathres':range(200)}],
        'LassoRegression':[{'features':range(50)}],

    }
```

With this approach we have params seperately declared. And its easy when they are not tightly coupled. Also this makes the implementation of tracking the error for a number of parameters extremely easy.

d

The perfomance of Ridge regression is some what better than the FSLinearRegression even though all the parameters are included.

regularizer| error

0.01 0.16467384
0.1 0.16462635
1.0 0.16441865

As you can see the regularizer increases the error decreases.

e

f

As the regularizer increases the error increases :

Avg Error: 0.318793566974
Standard Error 0.00325972643039

Avg Error: 0.329134784779 Standard Error 0.00286290008664

g

I have taken alpha = 0.1 and tested it for all the features for 7999 training points.

h