Assignment 2 Report

CS5691

PRML

K Akranth Reddy
ME20B100

02 April,2023

Question 1

(1) You are given a data-set with 10000 points in (R^{100},R) (Each row corresponds to a data point where the first 100 components are features and the last component is the associated y value).
i. Obtain the least squares solution W_{ML} to the regression problem using the closed form expression.

$$W_{ML} = (XX^T)^{-1}XY$$

In [37]:
```
def find_W_ML(X,Y):
    '''
    X -> input
    Y -> output

    returns the maximum likihood W value W_ML
    '''
    XXt=np.matmul(X,X.T)
    inv_XXt=np.linalg.inv(XXt)
    XY=np.matmul(X,Y)
    W_ML=np.matmul(inv_XXt,XY)
    return W_ML
```

In [38]:
```
W_ML=find_W_ML(X,Y)
```

In [39]:
```
#calculating the mean square error on test data using W_ML weights trained from training data.

Y_fit=np.matmul(X_test.T,W_ML)
error=sum(sum((Y_test-Y_fit)**2)/Y_test.shape[0])
print(r"The mean square error for W_ML =",error)
```

```
The mean square error for W_ML = 0.37072731116978697
```

The mean square error calculated on the test data is

$$\sum_{n=1}^{N_{test}} \frac{(Y_{test} - X_{test}^T W_{ML})^2}{N_{test}} = 0.370727$$

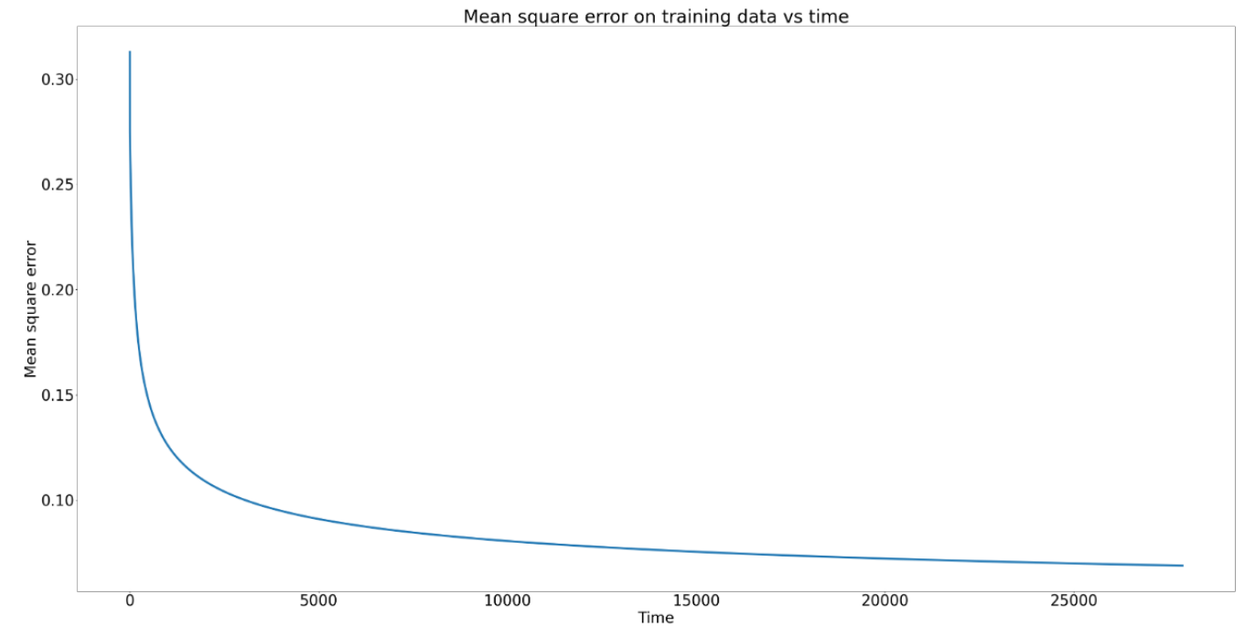
The mean square error calculated on the Training data is

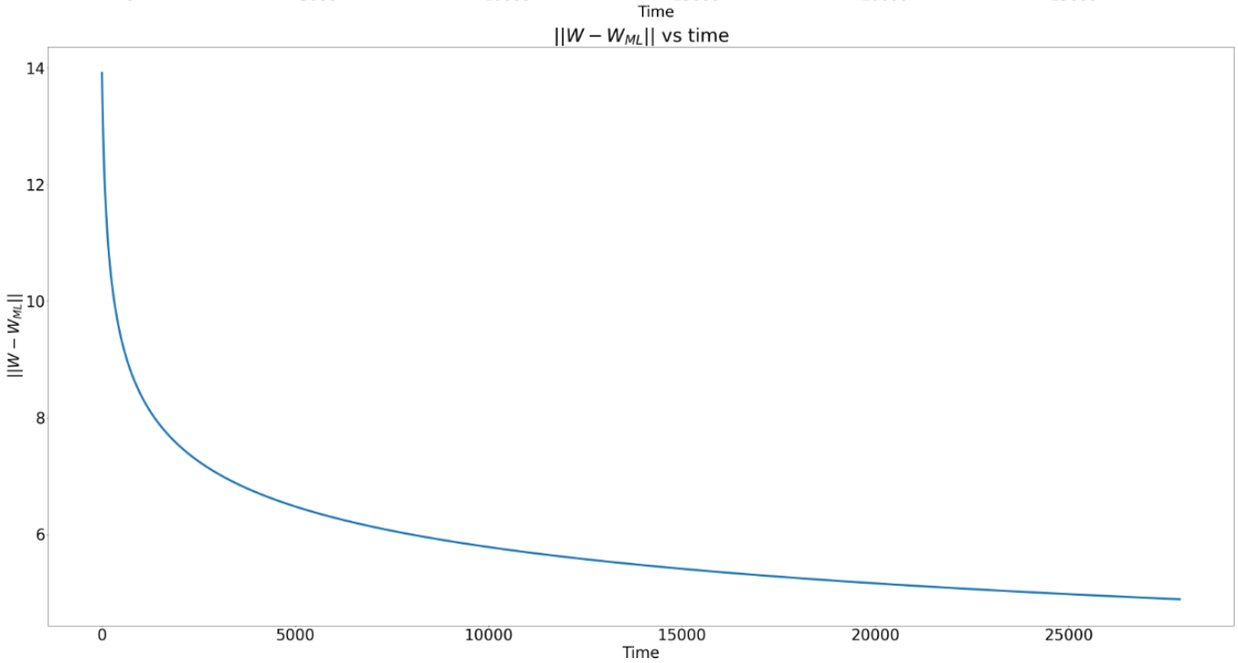
$$\sum_{n=1}^{N} \frac{(Y - X^T W_{ML})^2}{N} = 0.039686$$

ii. Code the gradient descent algorithm with suitable step size to solve the least squares algorithms and plot $||W_t - W_{ML}||$ as a function of t. What do you observe?

First I have used a learning step size as 1/t but the solution was diverging, mean square error became very high.
So I tried to use some $\frac{constant}{t}$ as step size and $\frac{0.0001}{t}$ worked, the solution converged.

CASE-1: η=0.0001/t
Number of iteration took to converge is 27926.
Time taken to run the iterations is 359.449 sec.
Mean square error on the training data = 0.0688547.
Mean square error on the test data = 0.3046369.
The algorithm took a lot of time to converge to W_{ML} because with each iteration the step size decreased further.
The mean square error and $||W_t - W_{ML}||$ as a function of number of iterations (t) is show below.

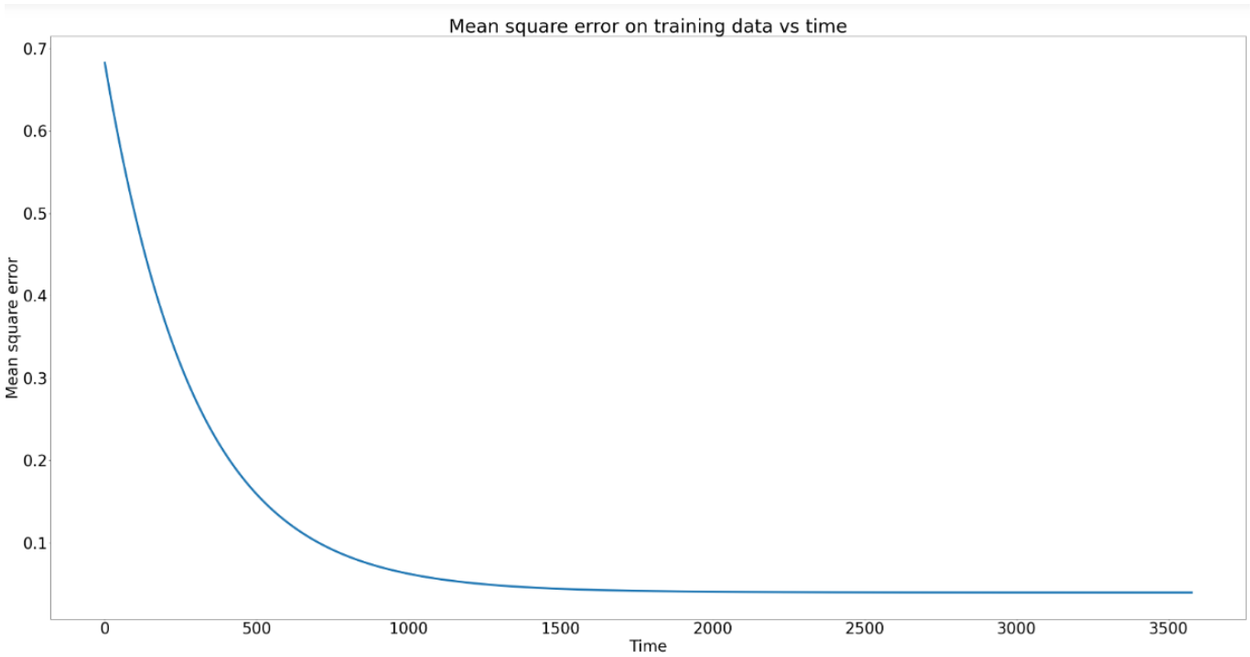


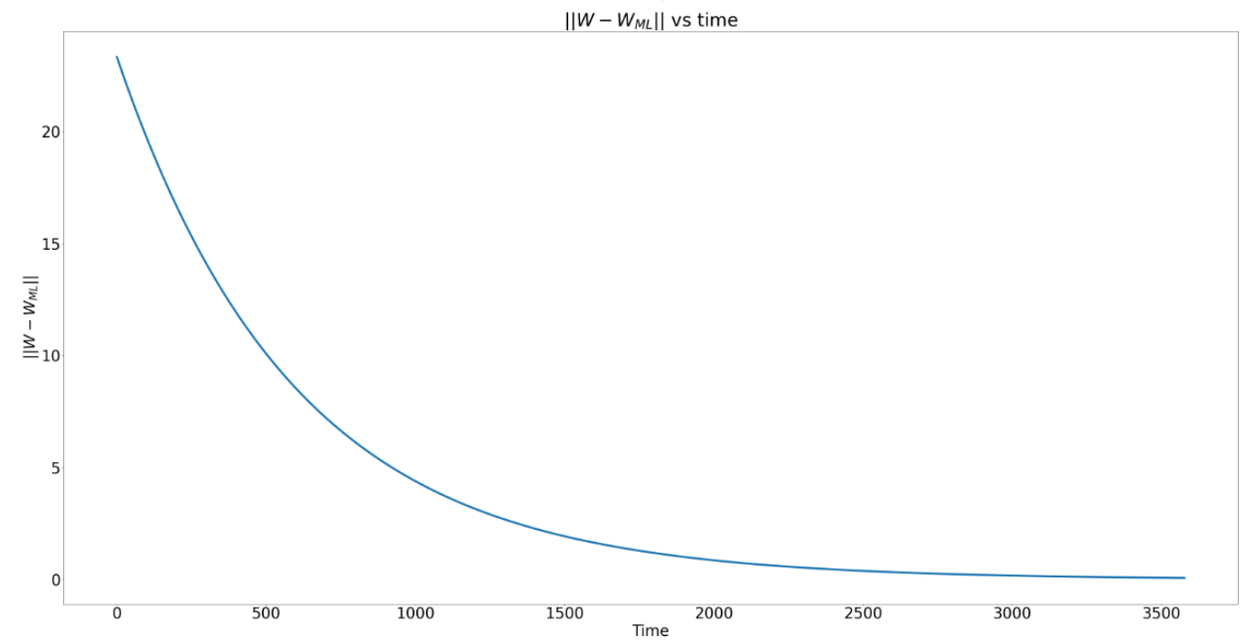


I ran the gradient descent algorithm for a constant learning step size η=0.000001. (I fine-tuned the hyper parameter by trial and error).

CASE-2: η=0.00001
For this constant learning step size the gradient descent algorithm came pretty close to the W_{WL} value.

Number of iteration = 2956.
Time taken to run = 37.965 sec.
Mean square error on the training data = 0.0396929
Mean square error on the test data = 0.36948838
Notice that the number of iterations it took to converge is very less compared to above CASE-1 η.
Below are the results of **constant learning rate**.





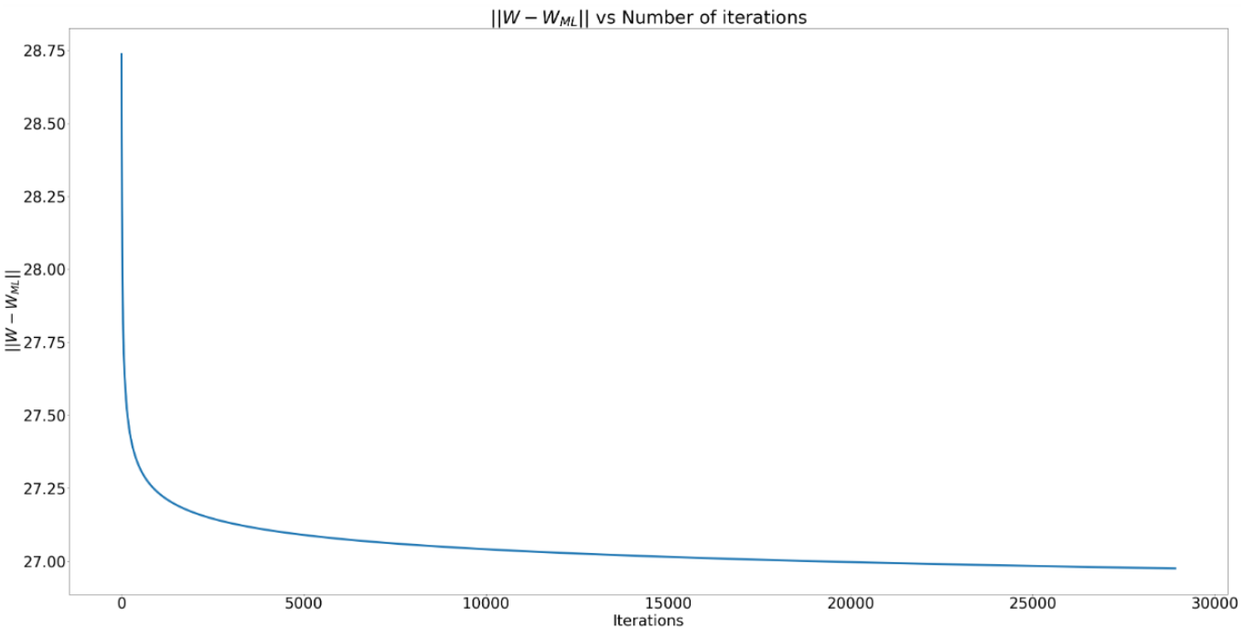
As we can see the for the case of dynamically changing step size Weights W are coming little close to W_{ML} and by that time t becomes larger thereby step size become very low and the

Weights are gonna need a lot of iteration more to converge more close to W_{ML}, but for the case of constant step size the algorithm is finding the solution very close to W_{ML} in less time. As the number of iteration increase, the weights W^t are almost approaching W_{ML} there by the mean square error on the test data calculated using W^t is almost the same as W_{ML}.

iii. Code the stochastic gradient descent algorithm using batch size of 100 and plot $||W_t - W_{ML}||$ as a function of t. What are your observations?

The data set size is 10000 samples, in Stochastic gradient descent we use small batches of size 100 to compute $\tilde{X}$ and $\tilde{Y}$ next average the computed gradient over some samples.
$W = W + \eta \nabla f_{avg}$
$\nabla f_{avg} = \sum_{n=1}^{N} \frac{2(\tilde{X}\tilde{X}^T W - \tilde{X}\tilde{Y})}{N}$
$\tilde{X} \in$ (d,$\tilde{n}$)
X $\in$ (d,n)

$\eta = 0.0001/t$
Number of iteration = 28919
Time taken = 474.988 sec.
Mean square error on training data = 0.922222
Mean square error on test data = 0.8250336



The beauty of stochastic gradient descent comes out when we use a large data set where each computation of the derivative matrix is a huge computational workload so using batch instead of whole data set to compute the derivative matrix reduces computational load.
Because instead of doing gradient computation which involves X of size (d,n) we use $\tilde{X} \in (d, \tilde{n})$ where $\tilde{n} < n$.
In the given question the data set is not considerable large, hence the performance of Stochastic gradient descent is almost the same as gradient descent.

Question 2

(2) Consider the same data-set as in Question (1). You are additionally given a data-set with 500 points for testing which you cannot use during train/cross-validation.

i. Code the gradient descent algorithm for ridge regression.

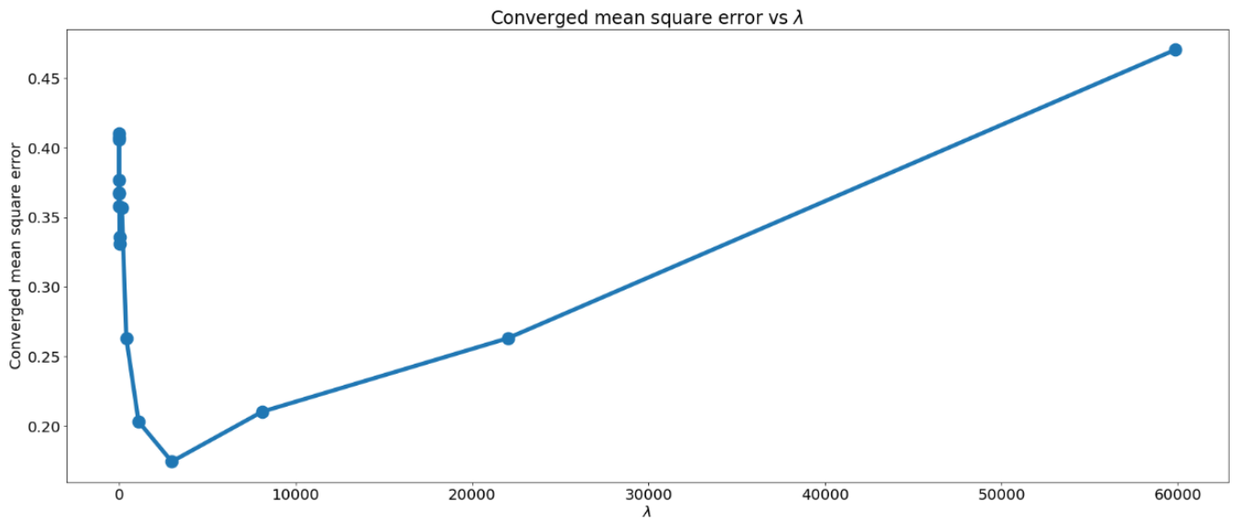
Gradient descent algorithm code for ridge regression

```
def error_ridge(Y,W,X,lamda):
    '''Returns the mean square error between actual Y and predicted Y_fit'''
    er=np.matmul(X.T,W)
    er=sum(((Y-er)**2)/2)+sum((W**2)*0.5*lamda)
    return er/Y.shape[0]

def derivative_ridge(X,W,Y,lamda):
    '''Returns the calculated derivative value for ridge regression problems,
    returns a matrix of size same as W'''
    XXt=np.matmul(X,X.T)
    xy=np.matmul(X,Y)
    return (np.matmul(XXt,W)-xy+W*lamda)
```

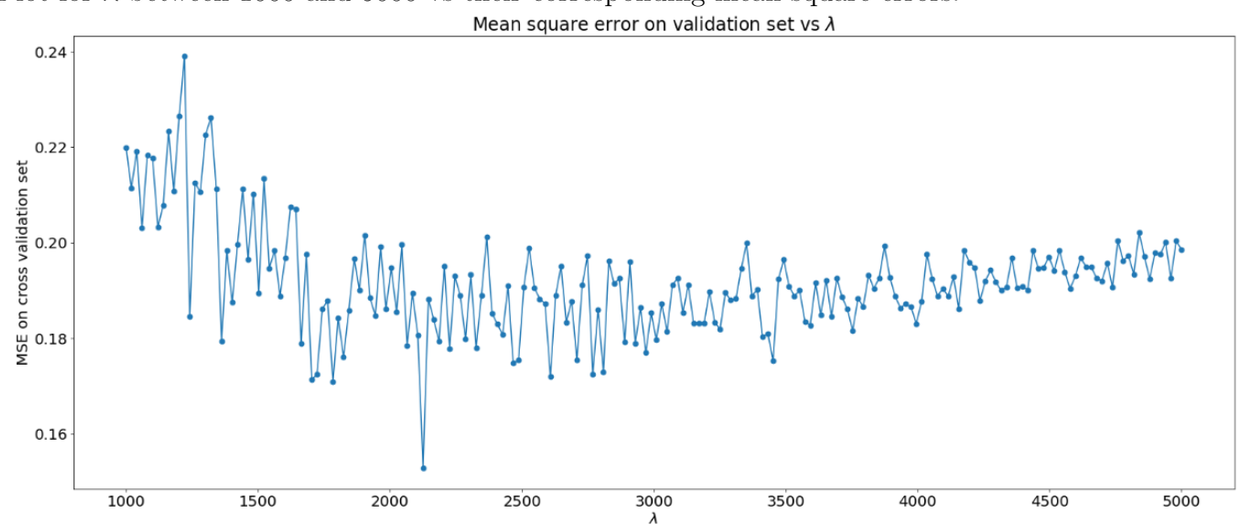
```
def run_ridge_regression(X,Y,lamda,tolerance=0.000005):
    '''
    X is input
    Y is lables
    lamda is hyperparameter which is fine-tuned using cross-validation set

    returns the weights learned for the ridge regression problem.
    '''
    W=np.random.rand(100,1)
    error_store=np.array([1e7])
    count=0
    verify=10
    i=0
    while(verify>tolerance):
        i=i+1
        step_size=0.0001/i
        W=W-step_size*derivative_ridge(X,W,Y,lamda)
        error_store=np.hstack((error_store,error_ridge(Y,W,X,lamda)))
        count=count+1
        verify=abs(error_store[count-1]-error_store[count])

    print("Number of iteration=",count)
    plt.plot(error_store[1:])
    plt.title("Mean square error plot")
    print("Mean square error converges to: ",error_store[count-1])
    return W
```

ii. Cross-validate for various choices of λ and plot the error in the validation set as a function of λ. For the best λ chosen, obtain W_r Also obtain W_{ML} for the training data. Compare the test error of W_r with W_{ML}. Which is better and why?

The training data is randomly spilt into 80:20 ratio into training set and cross validation set. Training set is used to train the model for a given λ, the validation set measures the goodness of λ.
I have used λ=[0.0497 0.1353 0.3678 1 2.71 7.38 20.08 54.59 148.41 403.42 1096.63 2980.97 8103.08 22026.46 59874.14] values and ran gradient descent algorithm, below are the plots of Mean square error vs λ.



From the graph we can see for some λ, $1000 \leq \lambda \leq 5000$ mean square error has minimum value.
Let's zoom into that region.
Plot for λ between 1000 and 5000 vs their corresponding mean square errors.



Minimum value for Mean square error on validation set occurs at $\lambda = 2125.62$, the weights calculated is W_r.
Mean square error on test data using W_r weights is 0.221126.
Mean square error on test data using W_{ML} weights is 0.370727.
Notice the MSE of W_r is less than MSE of W_{ML}.
Ridge regression is performing well here because as we are penalizing high weights values, the redundancy in the weights gets reduced.