Foundations of Machine Learning

DA5400 - Assignment 1 - BE21B037

(1) You are given a data-set in the file FMLA1Q1Data train.csv with 10000 points in (R2, R) (Each row corresponds to a datapoint where the first 2 components are features and the last component is the associated y value).
i. Write a piece of code to obtain the least squares solution wML to the regression problem using the analytical solution.
The analytical solution

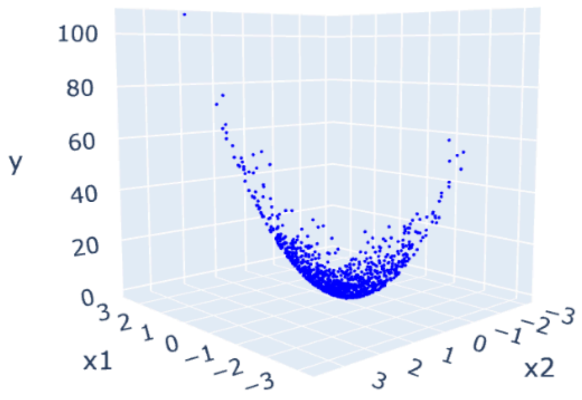
$$W_{ML} = (XX^T)^{-1}XY$$

The code for which is given by

```
import pandas as pd                  # Data Handling
import numpy as np                   # Data manipulation
from numpy.linalg import inv         # Calculate Matrix Inverse
import matplotlib.pyplot as plt      # Plot Graphs
import plotly.express as px          # Plot 3D figures to visualize data
import random                        # To randomize
import statistics                    # To calculate central tendencies
import warnings                      # To ignore warnings
warnings.filterwarnings('ignore')
```

```
train = pd.read_csv("/content/FMLA1Q1Data_train.csv",header = None)
test = pd.read_csv("/content/FMLA1Q1Data_test.csv",header = None)
train.columns = ["x1","x2","y"]
test.columns = ["x1","x2","y"]

fig = px.scatter_3d(train,x='x1', y='x2', z='y')
fig.update_traces(marker=dict(size=1,color = 'blue'))
fig.show()
```

The training data plotted in 3D gives us a visual representation of the problem we are trying to solve



The analytical solution is the calculated using the following code

```
XT = np.matrix(train[["x1","x2"]])
Y = np.matrix(train[["y"]])
X = XT.transpose()
XXT = np.dot(X,XT)
XXT_inv = inv(XXT)
XXT_invX = np.dot(XXT_inv,X)
Wml = np.dot(XXT_invX,Y)
print(f'The analytical solution, gives the ideal weights as \n w1 = {float(Wml[0])}
and w2 = {float(Wml[1])}' )
```

The output of which is :
The analytical solution, gives the ideal weights as
w1 = 1.4459991397233638 and w2 = 3.884211779400299

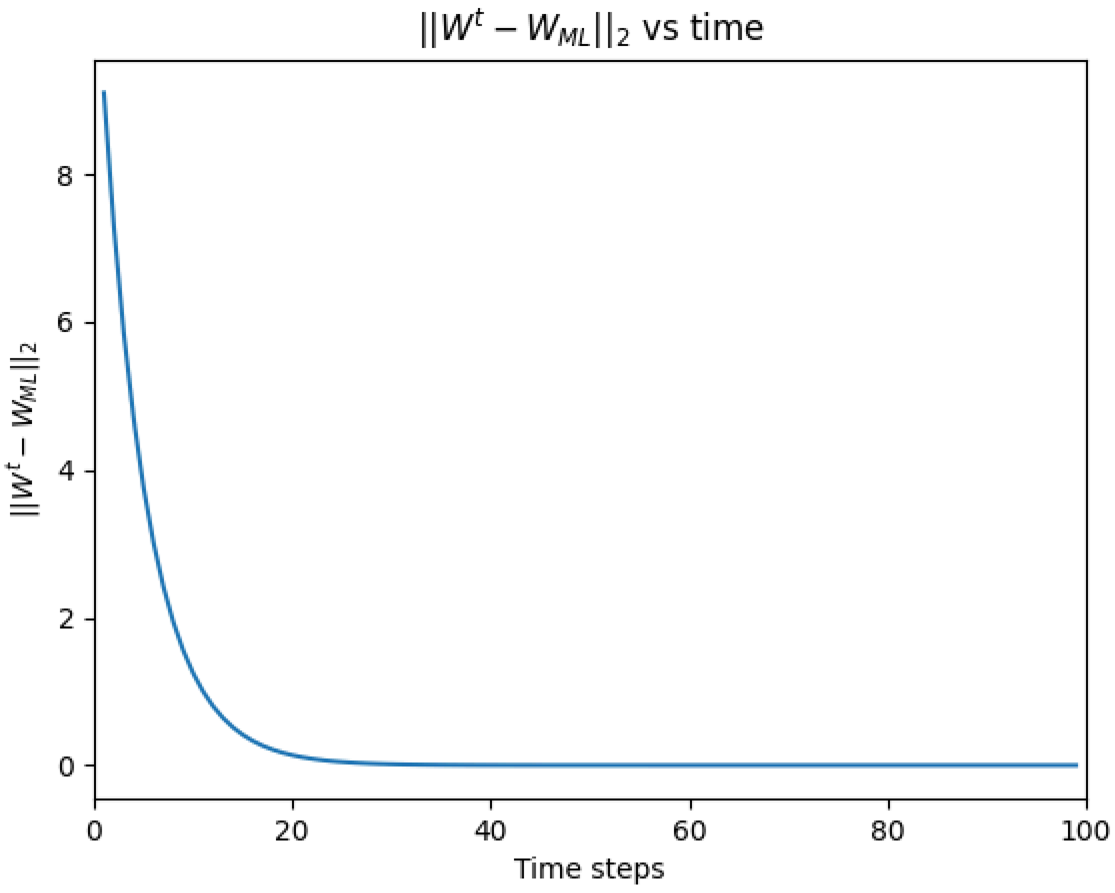
ii. Code the gradient descent algorithm with suitable step size to solve the least squares algorithms and plot // wt – wML // 2 as a function of t. What do you observe?

The gradient descent algorithm is run for 100 time steps with a step size of n = 1/(10000), the code of which is attached below

```
# Gradient descent Algorithm
random.seed(5400)
Wt = np.matrix([[random.randint(1, 10)],[random.randint(1, 10)]])
XXT = np.dot(X,XT)
XY = np.dot(X,Y)
err_func = []

for t in range(1,100):
    n =  1/(10000)
    gradf = np.dot(XXT,Wt)-XY             # gradient of least square
    err = float(np.linalg.norm(Wt - Wml)) # Error measurement
    err_func.append(err)
    Wt1 = Wt - (n*2*gradf)
    Wt = Wt1

y = [i for i in range(1,100)]

plt.plot(y,err_func)
plt.xlabel("Time steps")
plt.ylabel('$||W^t - W_{ML}||_2$')
plt.xlim(0,100)
plt.title("$||W^t - W_{ML}||_2$ vs time")
plt.show()
```

```
print(f'The weights after 100 steps of gradient descent algorithm, the weights are w1
= {float(Wt[0])} and w2 = {float(Wt[1])}, with an error of {np.linalg.norm(Wt-Wml)}')
```

The output graph and weights are



The weights after 100 steps of gradient descent algorithm, the weights are w1 = 1.4459991423073815 and w2 = 3.884211780872506, with an error of 2.9739771629036482e-09

Observation:

1) The gradient descent algorithm reaches the analytical solution in less than 30 time steps.
2) The final weights are w1 = 1.4459991423073815 and w2 = 3.884211780872506,
3) The weights have an error of 2.9739771629036482e-09 w.r.t the analytical solution

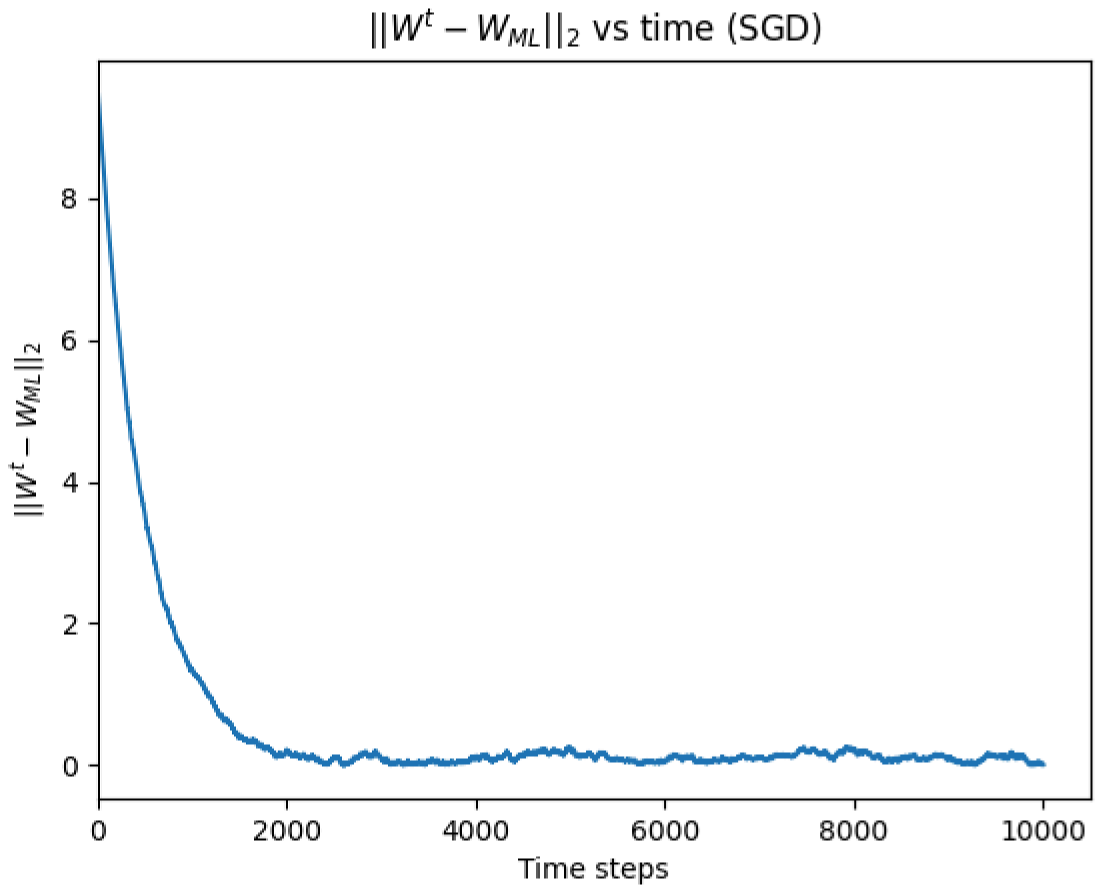
iii. Code the stochastic gradient descent algorithm using batch size of 100 and plot // wt – wML // 2 as a function of t. What are your observations?

We run the stochastic gradient descent algorithm (SGD) with a batch size of 100 and a step size of n = 1/(10000)

```
# Stochastic Gradient decent
random.seed(5400)
seeds = [random.randint(1,10000) for i in range(10000)]
```

```
error = []

Wt = np.matrix([[random.randint(1, 10)],[random.randint(1, 10)]])
weight_0 = []
weight_1 = []

for i in range(10000):
    seed = seeds[i]
    df = train.sample(100, random_state = seed)
    X = np.matrix(df[["x1","x2"]]).transpose()
    Y = np.matrix(df[["y"]])

    XT = X.transpose()
    XXT = np.dot(X,XT)
    XY = np.dot(X,Y)

    n = 1/(10000)
    gradf = np.dot(XXT,Wt)-XY
    Wt1 = Wt - (n*2*gradf)
    err = float(np.linalg.norm(Wt1 - Wml))
    error.append(err)
    Wt = Wt1
    weight_0.append(float(Wt[0]))
    weight_1.append(float(Wt[1]))

W_SGD = [statistics.mean(weight_0),statistics.mean(weight_1)]
print(W_SGD)

x = [i for i in range(10000)]

plt.plot(x,error)
plt.xlabel("Time steps")
plt.ylabel('$||W^t - W_{ML}||_2$')
plt.xlim(0,)
plt.title("$||W^t - W_{ML}||_2$ vs time (SGD)")
plt.show()
```

The output of which will be

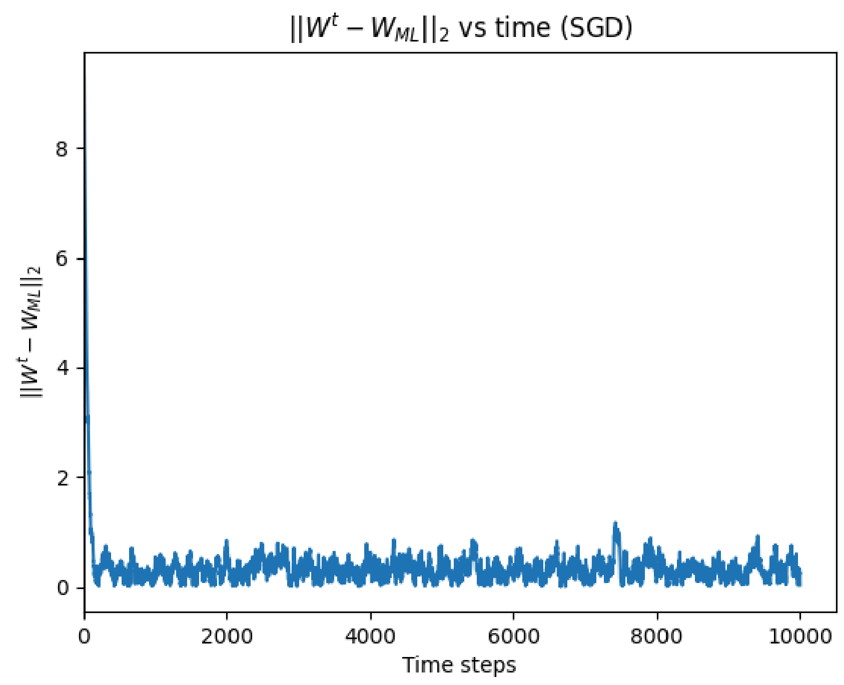


Observations:

1) SGD takes lesser computational power to calculate the weights, as we sample 100 data points at random for each step, but take larger timesteps to do the same.
2) The curve reaches the analytical solution by the 2000th step and continues to fluctuate, we calculate the SGD weights by taking the mean weights.
3) The weights are w1 = 1.8636048090161477, w2 = 4.1000792373831 84

For a larger step size of n = 1/(1000), the fluctuation is more

To get the below graph, change the parameter n to 1/1000 and re-run the above code

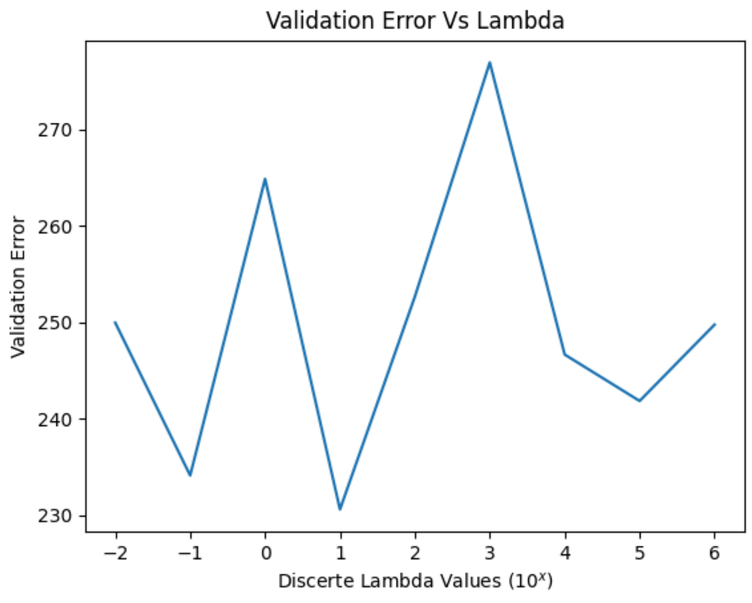


w1 = 1.4741934363501217 and w2 = 3.9093497953112726

iv. Code the gradient descent algorithm for ridge regression. 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 wR. Compare the test error (for the test data in the file FMLA1Q1Data test.csv) of wR with wML. Which is better and why?

To run the ridge regression algorithm we use 5-fold cross validation with, varying values of lambda to choose the lambda that gives the least error.

We first vary lambda scaling for every power of 10, and narrow down a lowest error for the lambda value of lambda = 613.



Validation error for lambda = 613 is 249.002336

We use this to find Wr which is w1 = 5.080070198336393, w2 = 5.506595500761838

```
# k-fold cross validation
random.seed(1000)
k = 5

def ridge(X_train,Y_train,X_test, Y_test,n,l, Wml):
    # Training
    X = X_train
    Y = Y_train

    XT = X.transpose()
    XY = np.dot(X,Y)
    XXT = np.dot(X,XT)
    Wt = np.matrix([[random.randint(1, 10)],[random.randint(1, 10)]])
    # print(X)

    for t in range(1,100):
        gradf = np.dot(XXT,Wt)-XY + l*Wt           # gradient of ridge regression
        Wt1 = Wt - (n*2*gradf)
        Wt = Wt1

    Wr = Wt # Solution weight
    # Testing
    Y_pred = np.dot(X_test.transpose(),Wr)
    err = 0
    for i in range(len(Y_pred)):
        err += (float(Y_pred[i]) - Y_test.iloc[i])**2

    return err/len(Y_pred), Wr

n = 1/(10000000)
l_values = [0.01, 0.1, 1, 10, 100, 1000,10000,100000,1000000]
l_values_log = [-2, -1, 0,1, 2, 3,4,5,6]


# l_values_0 = [i*10 for i in range(200)]
# l_values_1 = [i for i in range(600,650)]
# l_values_final = [i for i in range(610,620)]

l_values = [613] # comment this line to be able to get data for the plot


folds = np.array_split(train,k)
error = []
```

```python
Weight_0 = []
Weight_1 = []

for l in l_values:
    err = 0
    for i in range(k):
        folds[i].columns = ["x1","x2","y"]
        X_test = folds[i][["x1","x2"]].transpose()
        Y_test = folds[i][["y"]]

        train_set = pd.concat([folds[j] for j in range(5) if j != i])
        train_set.columns = ["x1","x2","y"]
        X_train = train_set[["x1","x2"]].transpose()
        Y_train = train_set[["y"]]

        E,Wr = ridge(X_train,Y_train,X_test, Y_test,n,l, Wml)

        err += E
        Weight_0.append(float(Wr[0]))
        Weight_1.append(float(Wr[1]))

    err = err/5

    error.append(err)

# uncomment the below code to plot the graph

# plt.xlabel("Discerte Lambda Values ($10^x$)")
# plt.ylabel('Validation Error')
# plt.title("Validation Error Vs Lambda")
# plt.plot(l_values_log ,error)
# plt.savefig("Validation_scale.png")
# plt.show()

W_r = [statistics.mean(Weight_0),statistics.mean(Weight_1)]
print(f"Wr is {W_r}")

# Testing against Wml
test0 = pd.read_csv("/content/FMLA1Q1Data_test.csv",header = None)
test0.columns = ["x1","x2","y"]

xtest0 = np.matrix(test0[["x1","x2"]])
ytest0 = test0["y"]
```

```
Wr = np.matrix(W_r).transpose()

ypred_ridge = np.dot(xtest0,Wr)
ypred_ml = np.dot(xtest0,Wml)

err_ridge = 0
err_ml = 0
for i in range(len(ytest0)):
    err_ridge += (ypred_ridge[i] - float(ytest0[i]))**2
    err_ml += (ypred_ml[i] - float(ytest0[i]))**2

print(f"Error for test data for Ridge regression is {float(err_ridge)/len(ytest0)}
and for the analytical solution is {float(err_ml)/len(ytest0)}")
```

Error for test data for Ridge regression is 163.7329315065954 and for the analytical solution is 142.76610663299087.

The analytical solution performs better than the ridge regression method.
Ridge regression also is better fine tuned for when the test data has a lot of noise. Ridge regression is also a stochastic process while the analytical solution gives the accurate solution, but is more computationally expensive. In this particular case it might be that the test dataset and the training dataset follow the same distribution with very little noise.

v. Assume that you would like to perform kernel regression on this dataset. Which Kernel would you choose and why? Code the Kernel regression algorithm and predict for the test data. Argue why/why not the kernel you have chosen is a better kernel than the standard least squares regression.

The two kernels which are suitable candidates for kernel regression would be a polynomial kernel with p = 2 (quadratic)
K2 = $(X^TX' + 1)^p$
or a gaussian kernel
K1 = $\exp(-||X-X'||^2/ 2\sigma^2)$ with a suitable sigma
Since the data points do not follow a linear structure, it would be better to use such a nonlinear kernel.

```
train1 = pd.read_csv("/content/FMLA1Q1Data_train.csv",header = None)
test1 = pd.read_csv("/content/FMLA1Q1Data_test.csv",header = None)
train1.columns = ["x1","x2","y"]
test1.columns = ["x1","x2","y"]

xT = np.matrix(train1[["x1","x2"]])
y = np.matrix(train1[["y"]])

n = len(train1)
```

```python
def k1(xi,xj):
    sigma = 1
    kernal = np.exp(-float(np.linalg.norm(xi-xj)**2)/(2*(sigma**2)))

    return kernal

# def k2(xi,xj):
#     p = 2
#     kernal = (float(np.dot(xi,xj.transpose()))+1)**p
#     return kernal

K = np.matrix(np.zeros((n, n)))

for i in range(n):
    for j in range(n):
        K[i,j] = k1(xT[i],xT[j])


Kinv = inv(K + np.identity(n))
alpha = np.dot(Kinv,y)


xtest = np.matrix(test1[["x1","x2"]])
ytest = test1["y"]

y_pred = []
for j in range(len(xtest)):
    ypred = 0
    for i in range(n):
        ypred += float(alpha[i])*k1(xT[i],xtest[j])

    y_pred.append(ypred)


err = 0
for i in range(len(y_pred)):
    err += (y_pred[i] - float(ytest[i]))**2
print(f'The determinant of the Kernel matrix before adding the identity is
{np.linalg.det(K)}')

print(f'The error for kernel regression is {err/len(y_pred)}')
```

The error for each of the two kernel cases for the datasets are
K2 (quadratic) error = 25167178.231218964, which is worse than the linear regression case

K1 (Guassian) error = 2.7540754238275436 which is a much better performance (sigma = 1).

Note that:

```
np.linalg.det(K)
```

Outputs -0.0
The K matrix is likely to have determinant zero, so we add an Identity matrix to the K matrix to take care of such corner case.
So the gaussian kernel would be the ideal candidate for this particular dataset.

The reason why the gaussian kernel would work better than the standard least squares regression, is because the dataset is non linear in structure. The dataset being that the train and test data follow a non- linear distribution. We can visually see this in this particular dataset by plotting the 3-dimensional plot. [x1,x2,y]. As shown in the first question, the data points are clearly nonlinear in nature.