**1. Linear Regression**

**Answer 1.0**

*Importing libraries:*

**from** numba **import** njit

**import** numpy **as** np

**from** numpy **import** linalg **as** LA

**import** matplotlib.pyplot **as** plt

**import** plotly.graph\_objects **as** go

**import** time

%matplotlib inline

*Set seed and precision::*

np.random.seed(0)

np.set\_printoptions(precision = 3, suppress = True)

*Function to generate X and Y::*

# gen X i.e. m datapoints with k features (k dimensions)

@njit

**def** gen\_X(m=1000, mean=0, variance=0.1):

# std. dev.

sd=**pow**(variance, 0.5)

# gen diff features/columns

X0 = np.ones((m, 1))

X1\_10 = np.random.normal(loc=0, scale=1, size=(m, 10))

X11 = np.empty((m, 1))

X12 = np.empty((m, 1))

X13 = np.empty((m, 1))

X14 = np.empty((m, 1))

X15 = np.empty((m, 1))

X16\_20 = np.random.normal(loc=0, scale=1, size=(m, 5))

**for** i **in** **range**(m):

X11[i][0] = X1\_10[i][0] + X1\_10[i][1] + np.random.normal(loc=mean, scale=sd)

X12[i][0] = X1\_10[i][2] + X1\_10[i][3] + np.random.normal(loc=mean, scale=sd)

X13[i][0] = X1\_10[i][3] + X1\_10[i][4] + np.random.normal(loc=mean, scale=sd)

X14[i][0] = 0.1\*X1\_10[i][6] + np.random.normal(loc=mean, scale=sd)

X15[i][0] = 2.0\*X1\_10[i][1] - 10 + np.random.normal(loc=mean, scale=sd)

X=np.concatenate((X0, X1\_10, X11, X12, X13, X14, X15, X16\_20), axis=1)

**return** X

# gen Y/target/labels/output

**def** gen\_Y(X, mean=0, variance=0.1):

#std. dev.

sd=**pow**(variance, 0.5)

m=np.shape(X)[0]

Y=np.empty((m))

**for** i **in** **range**(m):

Y[i] = 10 + **sum**([(**pow**(0.6,j))\*X[i][j] **for** j **in** **range**(1,11)]) + np.random.normal(loc=mean, scale=sd)

**return** Y

*Different Regression Classes::*

**class** NaiveRegression():

**def** \_\_init\_\_(self):

**pass**

**def** fit(self, X, Y):

m, k = np.shape(X)

self.w = np.zeros((k,1))

self.w = np.dot(np.dot(LA.inv(np.dot(X.T, X)), X.T), Y)

#return self.w

**def** predict(self, X):

**return** np.dot(X, self.w)

**def** MSE(self, X, Y\_true):

Y\_pred = self.predict(X)

mse=(np.square(Y\_true - Y\_pred)).mean(axis=None)

**return** mse

**class** RidgeRegression():

**def** \_\_init\_\_(self):

**pass**

**def** fit(self, X, Y, l):

m, k = np.shape(X)

self.w = np.zeros((k,1))

self.w = np.dot(np.dot(LA.inv(np.dot(X.T, X) + l\*np.identity(k)), X.T), Y)

#return self.w

**def** predict(self, X):

**return** np.dot(X, self.w)

# MSE = Mean Squarred Error

**def** MSE(self, X, Y\_true):

Y\_pred = self.predict(X)

mse=(np.square(Y\_true - Y\_pred)).mean(axis=None)

**return** mse

**class** LassoRegression():

**def** \_\_init\_\_(self):

**pass**

#calculate rho (using coordinate descent)

**def** get\_rho(self, X, Y, w, j):

# j is the feature selector

# Delete 'j'th feature

X\_j = np.delete(X, j, 1)

# Delete 'j'th weight

w\_j = np.delete(w, j)

# Pred new X without 'j'th feature

X\_pred\_j = self.internal\_predict(X\_j, w\_j)

# Calc. the residual

residual = Y - X\_pred\_j

# Calc. rho

rho\_j = np.**sum**(X[:,j]\*residual)

**return**(rho\_j)

# Train model

**def** fit(self, X, Y, l, tol):

# m = datapoints, k = features

m, k = np.shape(X)

# w = weights, init with 0s

self.w = np.zeros((k,1))

# calc z value

z = np.**sum**(X \* X, axis = 0)

# while not converged

**while**(True):

prev\_w = np.copy(self.w)

# going 1 by 1 instead of random, was told it works for some weird reason

**for** j **in** **range**(**len**(self.w)):

# calc rho for 'j'th coordinate

rho\_j = self.get\_rho(X, Y, self.w, j)

# update 'j'th weight

**if** j == 0:

self.w[j] = rho\_j/z[j]

**elif** rho\_j < -l\***len**(Y):

self.w[j] = (rho\_j + (l\***len**(Y)))/z[j]

**elif** rho\_j > -l\***len**(Y) **and** rho\_j < l\***len**(Y):

self.w[j] = 0

**elif** rho\_j > l\***len**(Y):

self.w[j] = (rho\_j - (l\***len**(Y)))/z[j]

**else**:

self.w[j] = np.NaN

# Calc. change in weight after updating 'j'th weight

curr\_step = **abs**(prev\_w - self.w)

# calc max step (if less than tolerance then break from loop)

**if**(curr\_step.**max**() < tol):

**break**

self.w=self.w.flatten()

**def** predict(self, X):

**return** np.dot(X, self.w)

# this is used for calculating rho

**def** internal\_predict(self, X, w):

**return** np.dot(X, w)

# MSE = Mean Squarred Error

**def** MSE(self, X, Y\_true):

Y\_pred = self.predict(X)

mse=(np.square(Y\_true - Y\_pred)).mean(axis=None)

**return** mse

**Answer 1.1**

# Create X and Y

X = gen\_X()

Y = gen\_Y(X)

# Naive/OLS Reg

# Create model (object)

naive\_model = NaiveRegression()

# Model training

naive\_model.fit(X, Y)

# Save fitted/trained weights and bias

trained\_bias\_naive = naive\_model.w[0]

trained\_weights\_naive = naive\_model.w[1:]

# Result of solved model

**print**("Trained Weights:", trained\_weights\_naive)

**print**("Trained Bias:", trained\_bias\_naive)

# Calculate true weights and bias

true\_bias = 10

true\_weights = np.zeros((20,1))

**for** i **in** **range**(10):

true\_weights[i] = **pow**(0.6, i+1)

true\_weights = true\_weights.flatten()

# Comparison between trained vs true weights and bias

weight\_terms=**list**(**range**(1,21))

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights', x=weight\_terms, y=trained\_weights\_naive)

])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()

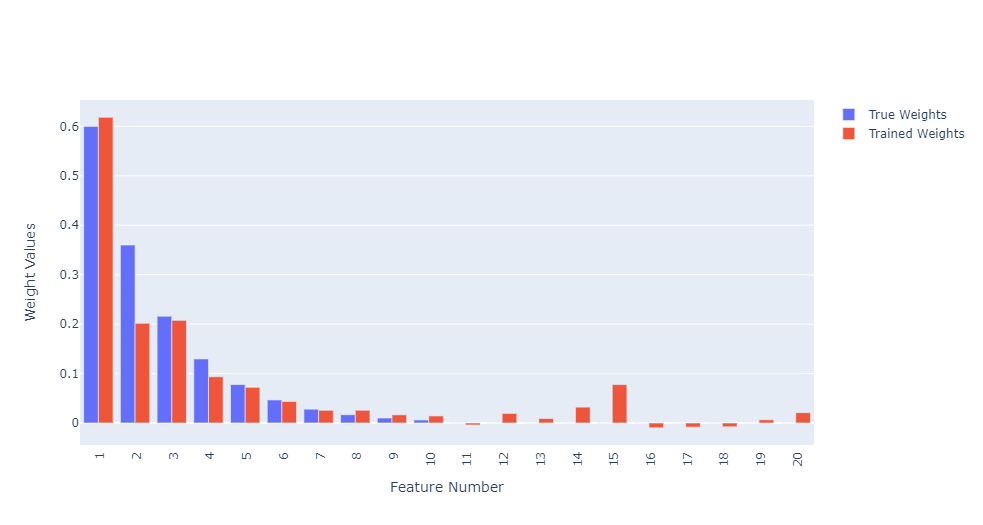
**print**("True Bias:", true\_bias)

**print**("Trained Bias:", trained\_bias\_naive)

Trained Weights: [ 0.618 0.202 0.208 0.094 0.072 0.044 0.026 0.026 0.017 0.014

-0.004 0.019 0.009 0.032 0.078 -0.01 -0.008 -0.008 0.007 0.021]

Trained Bias: 10.763008713927121



True Bias: 10

Trained Bias: 10.763008713927121

So, we can see that the naïve regression is not too shabby (being so simplistic). Almost all the weights upto w10 are in line with the true weights, except for the second one (w2).

Even the bias term is also quite close to the true bias as seen above.

From the trained model (on naïve regression), the most signification feature is X1, which is expected.

The least significant feature could have been anything from X11 to X20, but the model found X11 as the least significant feature (-0.004), although some most weights had fairly low values, except for X12, X14, X15 and X20.

The model was not able to prune anything as none of the weights found were 0 or even close to 0 below 3 point precision.

#Calculating True Error (by generating a large dataset)

X\_large = gen\_X(**int**(1e6))

Y\_large = gen\_Y(X\_large)

# Creating and training a new model since it is not

# mentioned in this question unlike the next question.

# Create model (object)

naive\_model\_large = NaiveRegression()

# Model training

naive\_model\_large.fit(X\_large, Y\_large)

# Calculate Error (MSE)

**print**("True Error:", naive\_model\_large.MSE(X\_large, Y\_large))

**print**("Training Error:", naive\_model.MSE(X, Y))

True Error: 0.09998263140206165

Training Error: 0.09514085940071908

The mean square error of a dataset of size (m=106) is ~0.1

**Answer 1.2**

# Ridge Reg

# Create model (object)

ridge\_model = RidgeRegression()

# l = lambda

l = 0.1

# Model training

ridge\_model.fit(X, Y, l)

# Calculate/Evaluate true error on large test data using the above model

X\_large = gen\_X(**int**(1e6))

Y\_large = gen\_Y(X\_large)

**print**("True Error:", ridge\_model.MSE(X\_large, Y\_large))

True Error: 0.10295346155187932

After training of a dataset of size m=104, we use the same model to find the error (true) for a large dataset (m=106), which is shown above.

# number of lambda values to test

lambda\_count = 130

# list of increasing lambda values

lambda\_list = np.linspace(start=-0.3, stop=1, num=lambda\_count+1, endpoint=True)

# list of errors (true) for increasing lambda values

mse\_list = []

avg\_runs = 1 # 100

**for** l **in** lambda\_list:

curr\_mse = 0

# averaging over 'avg\_runs' runs of each lambda

**for** i **in** **range**(avg\_runs):

# # Setting new "random-ish" seed everytime new data is generated

# # current time in milliseconds

# t = 1000 \* time.time()

# # the seed must be between 0 and and 2\*\*32 - 1

# np.random.seed(int(t) % 2\*\*32)

X = gen\_X()

Y = gen\_Y(X)

# Model training

ridge\_model.fit(X, Y, l)

curr\_mse += ridge\_model.MSE(X,Y)

curr\_mse /= avg\_runs

# Calculate Error (MSE) and append to the list

mse\_list.append(curr\_mse)

plt.plot(lambda\_list, mse\_list)

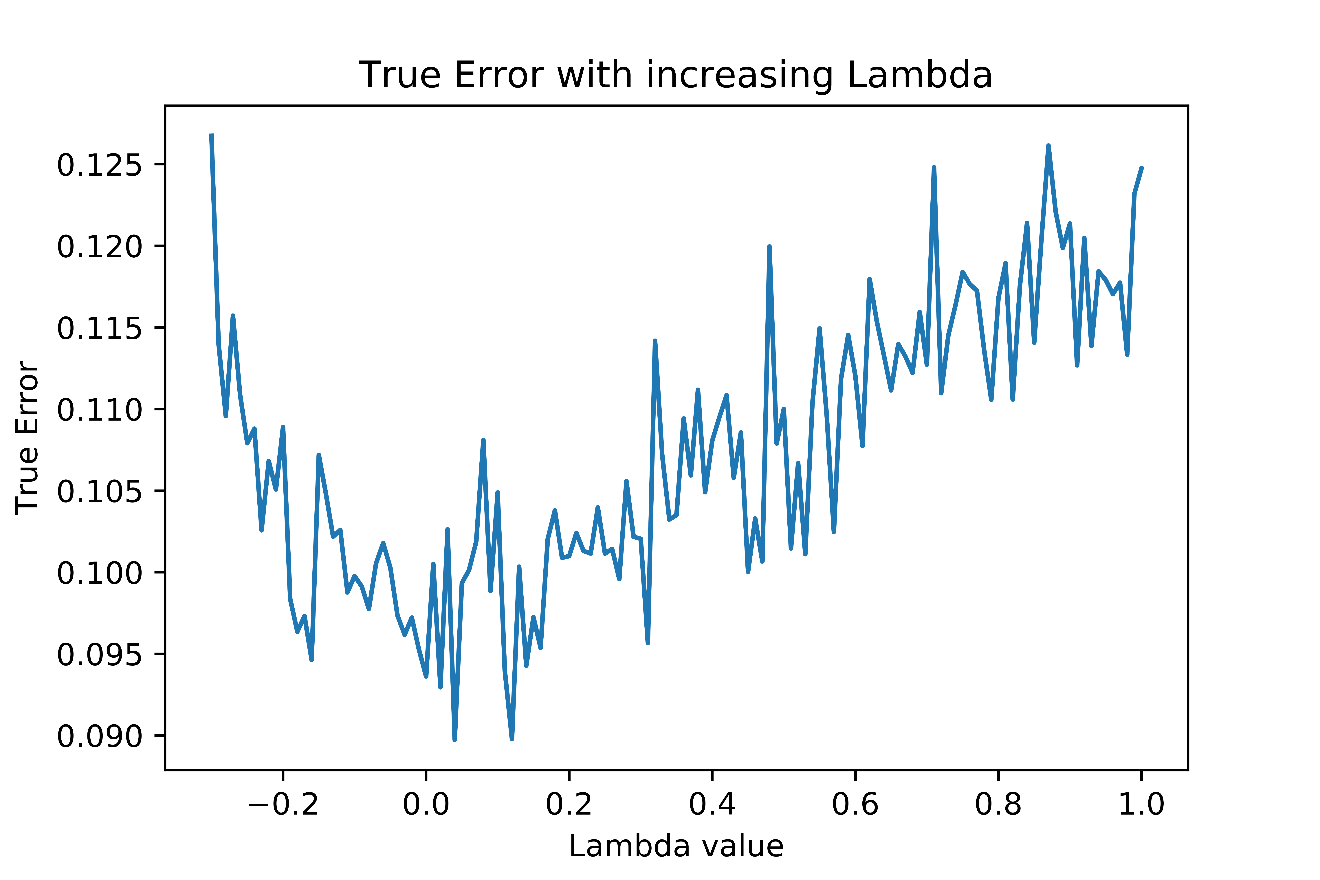
plt.title("True Error with increasing Lambda")

plt.xlabel("Lambda value")

plt.ylabel("True Error")

plt.savefig('Q2\_fig.png', dpi=1200)

plt.show()



The estimated error as a function of lambda is shown above.

# Finding optimal Lambda

optimal\_lambda\_index = np.argmin(mse\_list)

optimal\_lambda\_value = lambda\_list[optimal\_lambda\_index]

**print**("Optimal Lambda value: ", optimal\_lambda\_value)

Optimal **Lambda** value: -0.019999999999999962

The optimal Lambda value which minimizes the testing error is -0.02

# Finding weights and biases with optimal lambda

optimal\_w = ridge\_model.fit(X, Y, optimal\_lambda\_value)

# Save fitted/trained weights and bias

trained\_bias\_ridge = ridge\_model.w[0]

trained\_weights\_ridge = ridge\_model.w[1:]

# Result of solved model

**print**("Trained Weights:", trained\_weights\_ridge)

**print**("Trained Bias:", trained\_bias\_ridge)

# Comparison between trained vs true weights and bias

weight\_terms=**list**(**range**(1,21))

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights', x=weight\_terms, y=trained\_weights\_ridge)

])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()

**print**("True Bias:", true\_bias)

**print**("Trained Bias:", trained\_bias\_ridge)

Trained Weights: [ 0.58 0.323 0.262 0.178 0.056 0.063 0.029 0.015 0.012 -0.002

0.02 -0.051 0.014 -0.014 0.002 -0.01 0.009 -0.014 0.011 0.01 ]

Trained Bias: 10.022940502350144



True Bias: 10

Trained Bias: 10.022940502350144

The weights and biases at Lambda=-0.02 and its comparison to true weights are given above.

From the trained model (on ridge regression), the most signification feature is X1, which is expected.

The least significant feature could have been anything from X11 to X20, but the model found X10 and X15 as the least significant feature (-0.002 and 0.002 respectively), although some most weights had fairly low values, except for X12.

The true weight for X10 is 0.06 (quite small), so its not that bad that it got a very small weight after training.

The model was not able to prune anything as none of the weights found were 0 or even close to 0 below 3 point precision.

# Comparison between true, trained\_naive and trained\_ridge weights

weight\_terms=**list**(**range**(1,21))

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights Naive', x=weight\_terms, y=trained\_weights\_naive),

go.Bar(name='Trained Weights Ridge', x=weight\_terms, y=trained\_weights\_ridge)

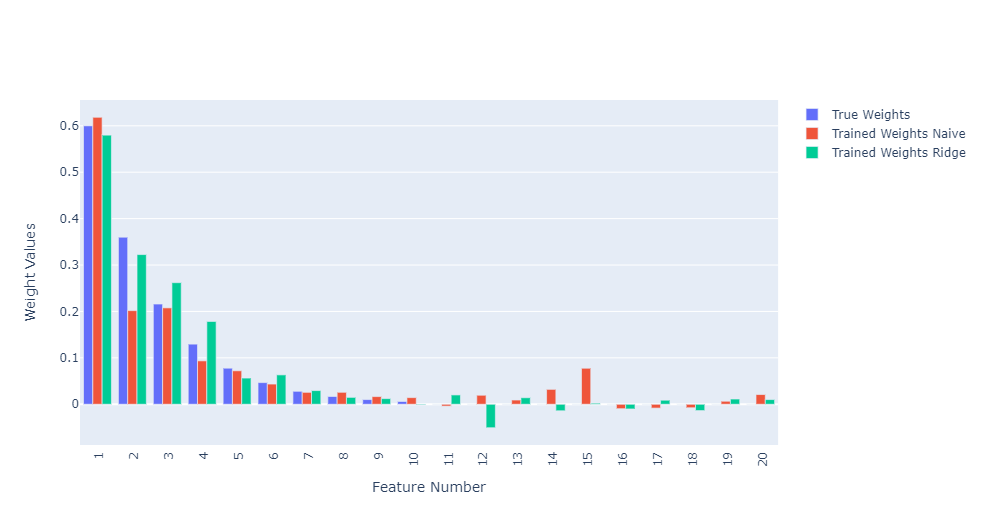
])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()



From the above plot, we can see they both perform fairly similar. If I had to choose one, I would go for the ridge regression model as the second weight (being an important feature) w2 was quite far off in the naïve model.

**Answer 1.3**

# Lasso Reg

# Create model (object)

e = LassoRegression()

# number of lambda values to test

lambda\_count = 10

# list of weights for increasing lambda values

weights\_list = []

# list of increasing lambda values

lambda\_list = np.linspace(start=0, stop=2, num=lambda\_count+1, endpoint=True)

**for** l **in** lambda\_list:

# Model training

lasso\_model.fit(X, Y, l, 0.01)

# Adding to the weights to a list i.e. list of numpy arrays

weights\_list.append(lasso\_model.w[1:])

# Converting to numpy array; now it becomes a 2d numpy array

weights\_array = np.asarray(weights\_list)

# Transposing the weights 2d array for plotting

weights\_vs\_lambda = weights\_array.T

plt.figure(figsize=(15,8))

**for** i **in** **range**(**len**(weights\_vs\_lambda)):

wl = weights\_vs\_lambda[i]

plt.plot(lambda\_list, wl, label = "w"+**str**(i+1))

plt.xlabel("Lambda value")

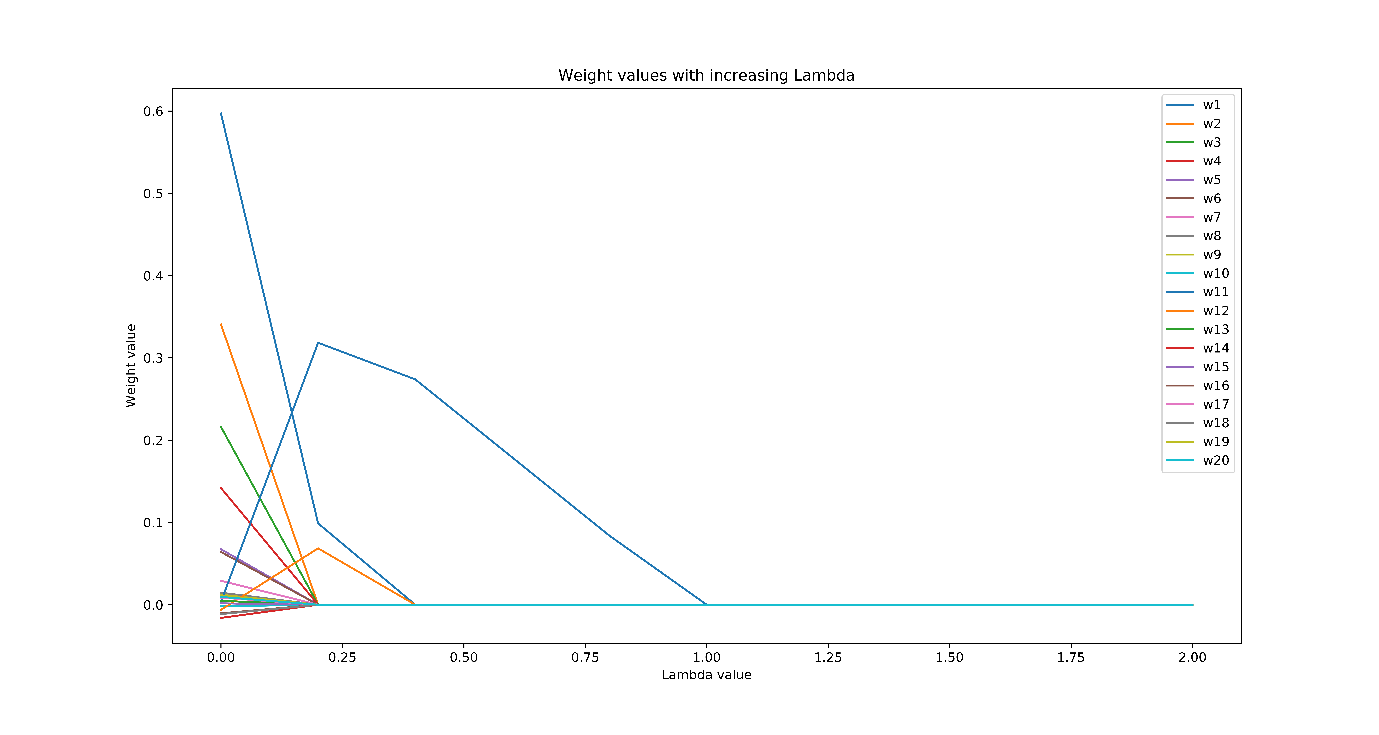
plt.ylabel("Weight value")

plt.legend(loc = "upper right")

plt.title("Weight values with increasing Lambda")

plt.savefig('Q3\_fig.png', dpi=1200)

plt.show()



From the above plot we see that features get eliminated (weights become 0) as lambda increases. At around lambda=1, we see all features eliminated.

**Answer 1.4**

# number of lambda values to test

lambda\_count = 200

# list of increasing lambda values

lambda\_list = np.linspace(start=0, stop=0.2, num=lambda\_count+1, endpoint=True)

# list of errors (true) for increasing lambda values

mse\_list = []

avg\_runs = 1 # 100

**for** l **in** lambda\_list:

curr\_mse = 0

# averaging over 'avg\_runs' runs of each lambda

**for** i **in** **range**(avg\_runs):

# # Setting new "random-ish" seed everytime new data is generated

# # current time in milliseconds

# t = 1000 \* time.time()

# # the seed must be between 0 and and 2\*\*32 - 1

# np.random.seed(int(t) % 2\*\*32)

X = gen\_X()

Y = gen\_Y(X)

# Model training

lasso\_model.fit(X, Y, l, 0.0001)

curr\_mse += lasso\_model.MSE(X,Y)

curr\_mse /= avg\_runs

# Calculate Error (MSE) and append to the list

mse\_list.append(curr\_mse)

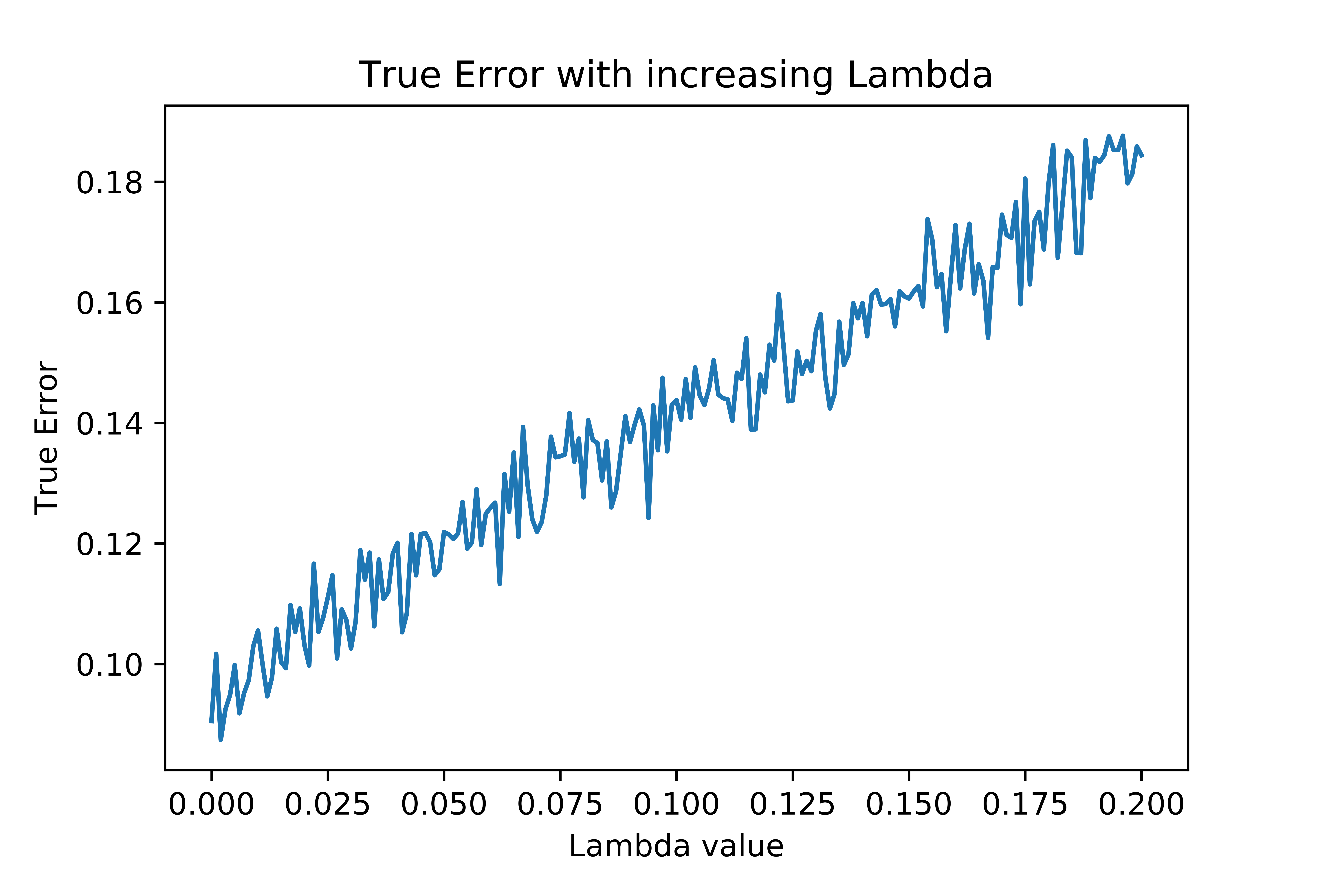
plt.plot(lambda\_list, mse\_list)

plt.title("True Error with increasing Lambda")

plt.xlabel("Lambda value")

plt.ylabel("True Error")

plt.show()



The estimated error as a function of lambda is shown above.

# Finding optimal Lambda

optimal\_lambda\_index = np.argmin(mse\_list)

optimal\_lambda\_value = lambda\_list[optimal\_lambda\_index]

**print**("Optimal Lambda value: ", optimal\_lambda\_value)

Optimal **Lambda** value: 0.002

The optimal Lambda value which minimizes the testing error is 0.002

# Finding weights and biases with optimal lambda

lasso\_model.fit(X, Y, optimal\_lambda\_value, 0.0001)

# Save fitted/trained weights and bias

trained\_bias\_lasso = lasso\_model.w[0]

trained\_weights\_lasso = lasso\_model.w[1:]

# Result of solved model

**print**("Trained Weights:", trained\_weights\_lasso)

**print**("Trained Bias:", trained\_bias\_lasso)

# Comparison between trained vs true weights and bias

weight\_terms=**list**(**range**(1,21))

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights', x=weight\_terms, y=trained\_weights\_lasso)

])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()

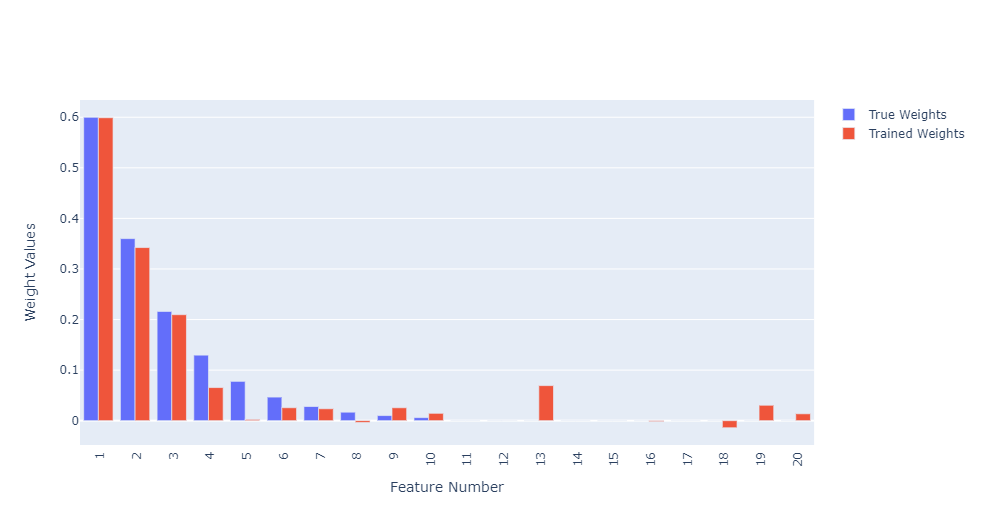
**print**("True Bias:", true\_bias)

**print**("Trained Bias:", trained\_bias\_lasso)

Trained Weights: [ 0.599 0.342 0.21 0.065 0.003 0.026 0.024 -0.004 0.026 0.015

0. 0. 0.069 0. 0. -0.002 0. -0.014 0.031 0.014]

Trained Bias: 10.004316493840976



True Bias: 10

Trained Bias: 10.004316493840976

The weights and biases at Lambda=0.002 and its comparison to true weights are given above.

From the trained model (on ridge regression), the most signification feature is X1, which is expected.

The least significant feature could have been anything from X11 to X20, but the model found X11, X12, X14, X15 and X17 as the least significant feature (weight 0 in 3 point precision).

The model was able to prune X11, X12, X14, X15 and X17

# Comparison between true, trained\_naive and trained\_ridge weights

weight\_terms=**list**(**range**(1,21))

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights Naive', x=weight\_terms, y=trained\_weights\_naive),

go.Bar(name='Trained Weights Ridge', x=weight\_terms, y=trained\_weights\_lasso)

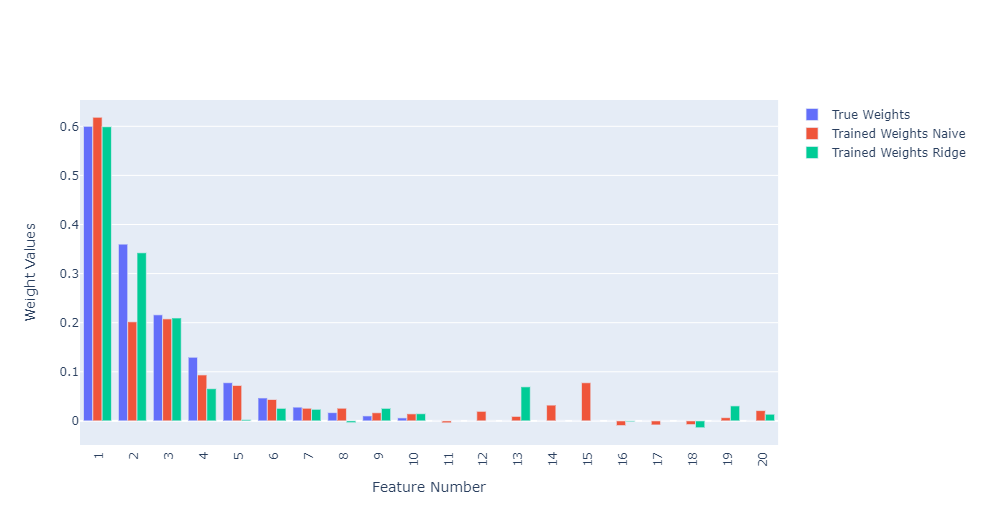
])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()



From the above plot, we can see they both perform fairly similar for weights w1 to w10, although the lasso regression model performs a bit better.

Furthermore, for weights w11 to w20, it was able to prune 5 features (as seen above), which is quite good.

**Answer 1.5**

X = gen\_X()

Y = gen\_Y(X)

# Create model (object)

lasso\_model = LassoRegression()

# from previous answer

optimal\_lambda\_value = 0.002

lasso\_model.fit(X, Y, optimal\_lambda\_value, 0.01)

trained\_weights\_lasso = lasso\_model.w[1:]

trained\_weights\_lasso

array([ 0.586, 0.308, 0.214, 0.119, 0.066, 0.052, 0.012, 0.005,

0.006, -0.002, 0.012, 0. , 0. , 0.029, 0.005, 0.001,

0.006, -0.008, 0. , -0.005])

With optimal lambda=0.002 (from answer 4), we trained a model on a new dataset. The weights corresponding to each feature is given above.

Now, we create a new data for X (called X\_pruned) which will not contain the features having corresponding weights=0.

# pruning weights lower than 3rd precision point

trained\_weights\_lasso\_precision = [**round**(num, 3) **for** num **in** trained\_weights\_lasso]

prunining\_index\_list = []

**for** i **in** **range**(**len**(trained\_weights\_lasso\_precision)):

**if**(trained\_weights\_lasso\_precision[i] == 0):

# pruning index increased by 1 since first column is 1 (for bias inclusion in weight)

prunining\_index\_list.append(i+1)

**print**(prunining\_index\_list)

[12, 13, 19]

# pruning away insignificant features i.e. below 3 points precision

X\_pruned = np.delete(X, prunining\_index\_list, axis=1)

We will find a good ridge regression regularization constant by using the same procedure we used in answer 2, i.e. test it for a range of lambda values and take the one with the least error.

# Create model (object)

ridge\_model = RidgeRegression()

# number of lambda values to test

lambda\_count = 100

# list of increasing lambda values

lambda\_list = np.linspace(start=-0.001, stop=0.001, num=lambda\_count+1, endpoint=True)

# list of errors (true) for increasing lambda values

mse\_list = []

**for** l **in** lambda\_list:

# Model training

ridge\_model.fit(X\_pruned, Y, l)

# Calculate Error (MSE) and append to the list

mse\_list.append(ridge\_model.MSE(X\_pruned, Y))

plt.plot(lambda\_list, mse\_list)

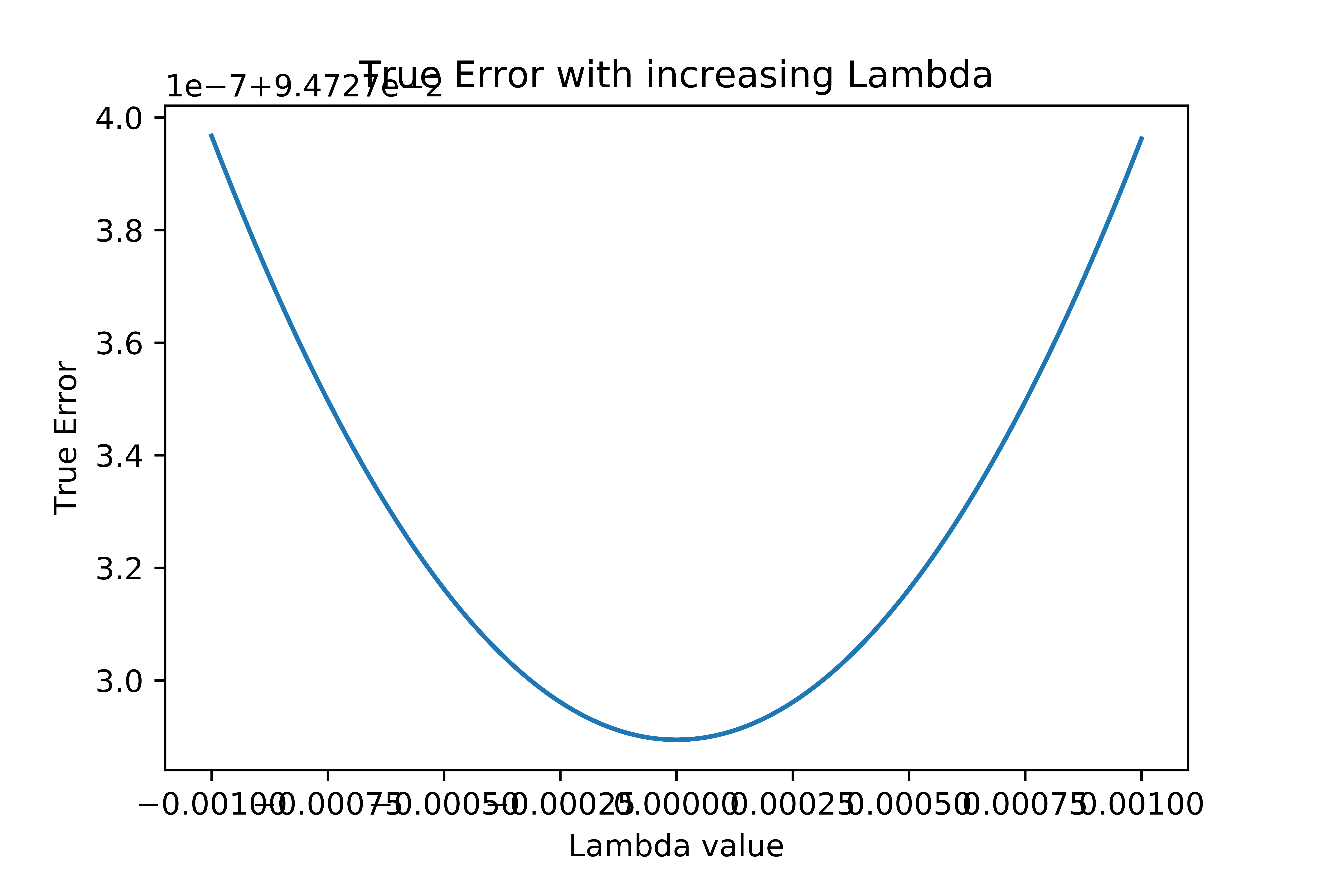
plt.title("True Error with increasing Lambda")

plt.xlabel("Lambda value")

plt.ylabel("True Error")

plt.savefig('Q5\_fig.png', dpi=1200)

plt.show()



# Finding optimal Lambda

optimal\_lambda\_index = np.argmin(mse\_list)

optimal\_lambda\_value = lambda\_list[optimal\_lambda\_index]

**print**("Optimal Lambda value: ", optimal\_lambda\_value)

Optimal **Lambda** value: 0.0

We find that error is lowest at lambda=0, i.e. ridge regression doesn’t do anything beneficial (same as naïve regression).

# Finding weights and biases with optimal lambda

optimal\_w = ridge\_model.fit(X\_pruned, Y, optimal\_lambda\_value)

# Save fitted/trained weights and bias

trained\_bias\_ridge\_after\_lasso = ridge\_model.w[0]

trained\_weights\_ridge\_after\_lasso = ridge\_model.w[1:]

# Result of solved model

**print**("Trained Weights:", trained\_weights\_ridge\_after\_lasso)

**print**("Trained Bias:", trained\_bias\_ridge\_after\_lasso)

# Comparison between trained vs true weights and bias

weight\_terms=**list**(**range**(1,21))

# adding 0 weights to pruned features to compare

**for** i **in** prunining\_index\_list:

trained\_weights\_ridge\_after\_lasso = np.insert(trained\_weights\_ridge\_after\_lasso, i-1, 0)

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights', x=weight\_terms, y=trained\_weights\_ridge\_after\_lasso)

])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()

**print**("True Bias:", true\_bias)

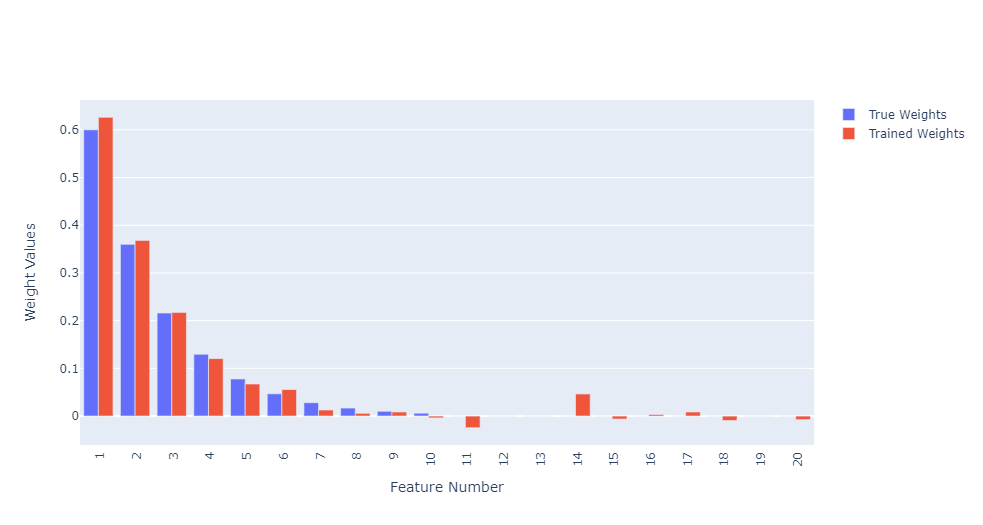
**print**("Trained Bias:", trained\_bias\_ridge\_after\_lasso)

Trained Weights: [ 0.626 0.368 0.217 0.12 0.067 0.056 0.013 0.006 0.009 -0.004

-0.025 0.046 -0.006 0.003 0.009 -0.009 -0.008]

Trained Bias: 9.932176951877114

*The weights given above are after pruning away the features so it wont have 20 weights.*

*In order to compare with the true weights or weights achieved in the naïve model, we insert the 0 weights to their respective positions while plotting.*

True Bias: 10

Trained Bias: 9.932176951877114

# Comparison between true, trained\_naive and trained\_ridge\_after\_lasso weights

weight\_terms=**list**(**range**(1,21))

fig = go.Figure(data=[

go.Bar(name='True Weights', x=weight\_terms, y=true\_weights),

go.Bar(name='Trained Weights Naive', x=weight\_terms, y=trained\_weights\_naive),

go.Bar(name='Trained Weights Ridge after Lasso', x=weight\_terms, y=trained\_weights\_ridge\_after\_lasso)

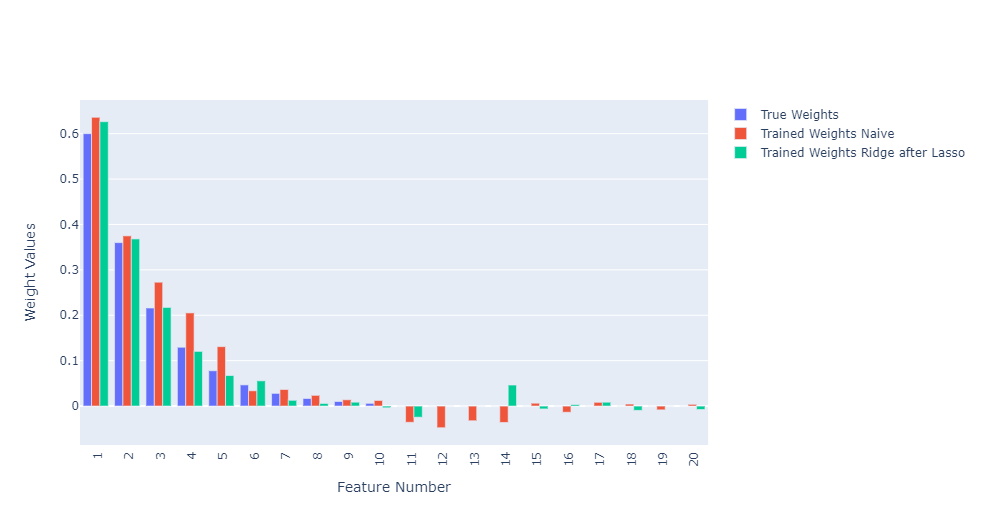
])

fig.update\_layout(barmode = 'group', xaxis\_tickangle = -90, xaxis = **dict**(tickmode = 'linear'))

fig.update\_xaxes(title\_text = "Feature Number")

fig.update\_yaxes(title\_text = "Weight Values")

fig.show()



The feature X12, X13 and X19 are pruned. Apart from those there are some more weights between w11 to w20 which are quite small.

All features except X12, X13 and X19 are significant (having non-zero weights).

Features X12, X13 and X19 are insignification (having zero weights).

# Comparing errors

**print**("Error of naive model:", naive\_model.MSE(X, Y))

**print**("Error of ridge (after lasso) model:", ridge\_model.MSE(X\_pruned,Y))

Error of naive model: 0.09562558616203298

Error of ridge (after lasso) model: 0.09197824756578925

The error on lasso-ridge model is lower (better) than the naïve model.

**2. SVMs**

**Answer 2.1**

*Importing libraries:*

**import** numpy **as** np

**from** scipy **import** optimize

*Create X and Y:*

X=np.asarray([[1, 1], [-1, 1], [-1, -1], [1, -1]])

Y=np.asarray([-1, 1, -1, 1])

*The Dual SVM Classifier:*

**class** DualSVM:

**def** \_\_init\_\_(self, kernel):

self.kernel = kernel

self.alpha = None

**def** fit(self, X, Y):

m = **len**(Y)

# init alpha

self.alpha = np.asarray([0.25, 0.25, 0.25, 0.25])

**def** obj\_func(alpha, epsilon\_t, X, Y, maximize\_sign):

# this sum is w/o the log terms

obj\_sum\_part1 = 0

# breaking into smaller sums for readability

sum1 = sum2 = sum3 = sum4a = sum4b = sum5 = 0

# skipping first term (as per given obj func)

# also alpha indexes reduced by 1 since passing alpha\_2 to alpha\_4

# (since we have to calc alpha\_1 from these)

**for** i **in** **range**(1, m):

sum1 += alpha[i-1] \* (Y[i] \* Y[0])

sum2 += alpha[i-1]

sum3 += alpha[i-1] \* (Y[i] \* Y[0])

sum4a += alpha[i-1] \* (Y[i] \* Y[0])

#sum4b += alpha[i-1] \* Y[i] \* (np.dot(X[i], X[0]))

sum4b += alpha[i-1] \* Y[i] \* self.kernel(X[i], X[0])

**for** j **in** **range**(1, m):

#sum5 += alpha[i-1] \* Y[i] \* (np.dot(X[i], X[j])) \* Y[j] \* alpha[j-1]

sum5 += alpha[i-1] \* Y[i] \* self.kernel(X[i], X[j]) \* Y[j] \* alpha[j-1]

sum1 = -sum1

#sum3 = pow(-sum3, 2) \* np.dot(X[0], X[0])

sum3 = **pow**(-sum3, 2) \* self.kernel(X[0], X[0])

sum4a = -sum4a

obj\_sum\_part1 = sum1 + sum2 - 0.5 \* (sum3 + 2\*(sum4a \* Y[0] \* sum4b) + sum5)

# ===== 1st part done =====

# this sum is the log terms

obj\_sum\_part2 = 0

sum6 = sum7 = 0

**for** i **in** **range**(1, m):

# print(sum(alpha))

sum6 += np.log(alpha[i-1])

sum7 += alpha[i-1] \* (Y[i] \* Y[0])

sum7 = np.log(-sum7)

# ===== 2nd part done =====

obj\_sum\_part2 = epsilon\_t \* (sum6 + sum7)

**return** maximize\_sign \* (obj\_sum\_part1 + obj\_sum\_part2)

# using this contraint to keep the first log term positive

**def** constraint1(alph):

**return** **sum**(alph) - 1e-8

cons1 = {'type': 'ineq', 'fun': constraint1}

# tunable hyper parameters

# t = time instances

t = 1000

# epsilon tends to 0 w.r.t. to time 't'

epsilon\_t\_list = np.linspace(start=1e-2, stop=1e-8, num=t, endpoint=True)

**for** e\_t **in** epsilon\_t\_list:

# we are optimizing over alphas except the first one

# args has -1 in the end since we want to maximize

optimal\_soln = optimize.minimize(fun = obj\_func,

x0 = self.alpha[1:],

args = (e\_t, X, Y, -1),

constraints = cons1,

method = 'SLSQP')

# update alphas as given by the solver

self.alpha[1:] = optimal\_soln.x

# calculate alpha\_1 using given formula

self.alpha[0] = -**sum**([self.alpha[k] \* (Y[k] \* Y[0]) **for** k **in** **range**(1,m)])

# to calc. bias (not asked in question though)

postive\_alpha\_index = self.alpha > 1e-8

self.support\_vector\_X = X[postive\_alpha\_index]

self.support\_vector\_Y = Y[postive\_alpha\_index]

# using equation 33 of Lecture notes "Worked SVMs ..."

self.b = self.support\_vector\_Y[0] - **sum**([self.alpha[j] \* Y[j] \* \

self.kernel(X[j], self.support\_vector\_X[0]) **for** j **in** **range**(m)])

# ==================================== END OF FIT ====================================

# using equation 34 of Lecture notes "Worked SVMs ..."

**def** predict(self, X):

m = np.shape(X)[0]

**return** np.sign(**sum**([self.alpha[i] \* self.support\_vector\_Y[i] \* self.kernel(self.support\_vector\_X[i], X) \

+ self.b **for** i **in** **range**(m)]))

The above contains the implemention of a barrier method dual SVM solver. The object of the DualSVM class after fitting/training stores the alpha values.

I initialized alpha values as 0.25 each. They need to be always greater than 0. And here the boundary region is so we are well within the boundary region. Also since the program will converge to the optimal solution, it doesn’t matter what value we pick but its better to pick a value between 0 and 1.

The barrier method is designed in such a way such that it doesn’t allow to move outside the constraint region. But we have to make sure to take a large number of steps or smaller step sizes to prevent jumping outside. Also in the optimizer I used a constraint that all alphas should sum up to more than equal to zero to prevent negative terms in the log term.

I chose ranging from 1e-2 to 1e-8 (close to 0), inclusive, with 1000 steps.

The solver finds optimal to (here ) and calculates from equation (4) of the question paper.

**Answer 2.2**

*The Kernel Function:*

**def** polyKernel(x, y):

**return** **pow**(1 + np.dot(x, y), 2)

*Main:*

# Create model

svm\_model = DualSVM(kernel=polyKernel)

# Train model

svm\_model.fit(X, Y)

# alpha values

svm\_model.alpha

array([0.12499954, 0.12503677, 0.125074 , 0.12503677])

# bias value

svm\_model.b

-3.709480115521302e-06

Programatically, we got alpha values almost equal to 0.125 each i.e. 1/8 and the bias value is almost 0.

The following is taken from my Answer in HW3 where the same was proven by hand:

The dual SVM problem is:

Expanding the above:

**Part 1:**

**For the Polynomial Kernel:**

and

X and Y values

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

Now our objective function becomes:

or,

Applying  **[i=A,B,C,D]**, we get:

or (rearranging),

Solving by Inverse Matrix Method:

A.X=B

B

Therefore,

By hand, we proved that the alpha values are 1/8 each and hence we can confirm the result we obtained via the solver.

**Answer 2.3**

From the above result, since all 4 inputs are support vectors, so optimum value of the

Hence,

or,

Referencing

and

A = ((1, 1), -1)

B = ((-1, 1), 1)

C = ((-1, -1), -1)

D = ((1, -1), 1)

or,

The first element of w\* is the bias:

The separating function is

or,

or,

or,

And this is the correct classifier for the XOR problem.

**References:**

1. Lecture Notes and Videos.
2. My HW3.
3. <https://tohtml.com/python/>
4. <https://www.youtube.com/watch?v=geFER2oVvvU>

\*\*\*