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 Squared 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
```

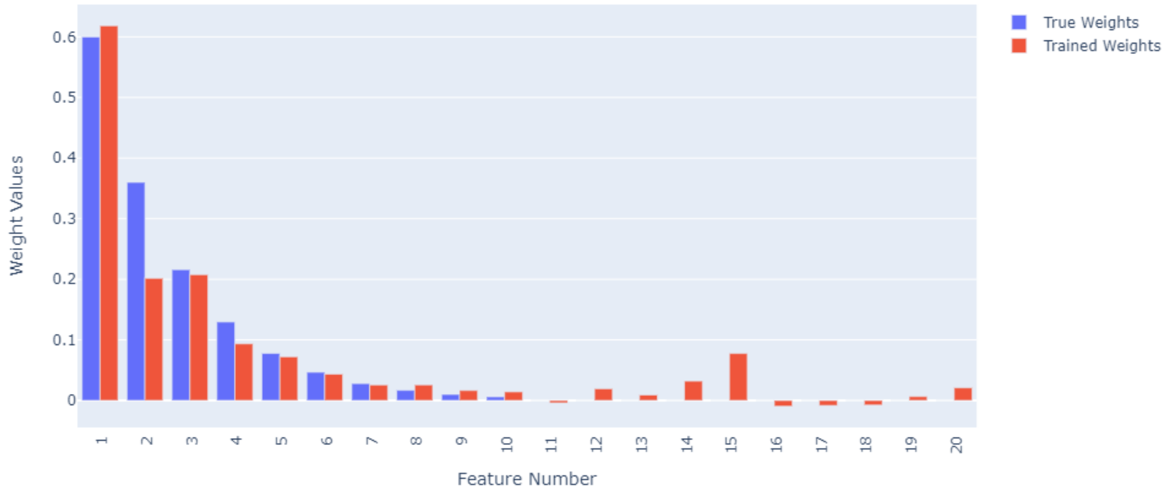
```
        residual = Y - X_pred_j

        # 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 w_{10} are in line with the true weights, except for the second one (w_2).

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 X_1, which is expected.
The least significant feature could have been anything from X_{11} to X_{20}, but the model found X_{11} as the least significant feature (-0.004), although some most weights had fairly low values, except for X_{12}, X_{14}, X_{15} and X_{20}.

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 ($m=10^6$) is ~0.1

Answer 1.2

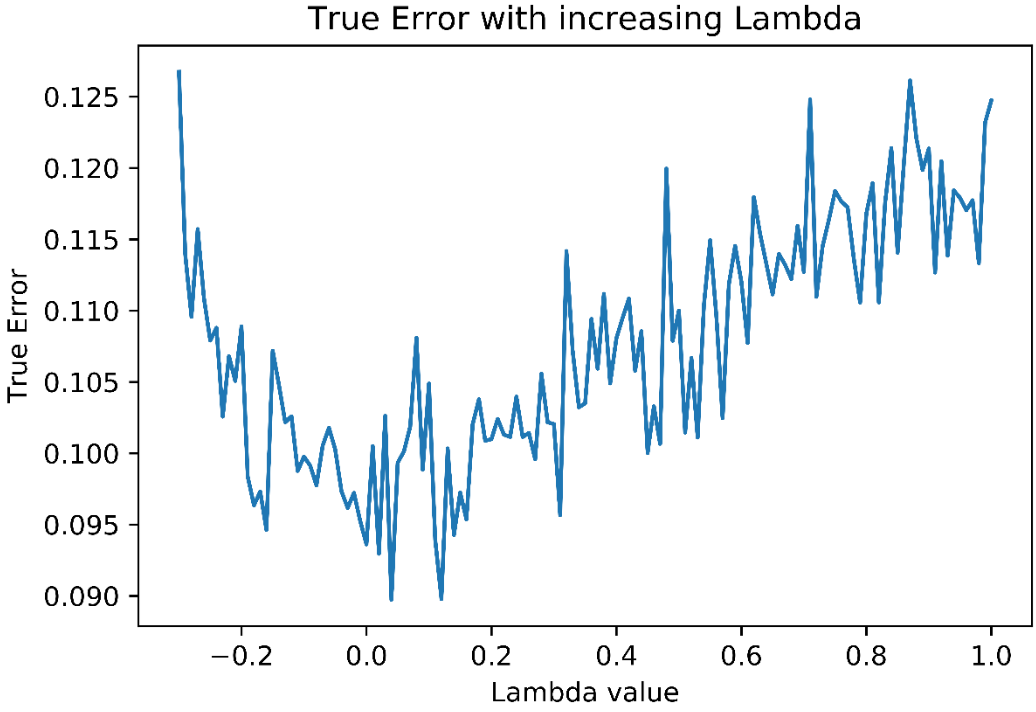
```
# Ridge Reg
# Create model (object)
ridge_model = RidgeRegression()

# 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=10^4, we use the same model to find the error (true) for a large dataset (m=10^6), 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.

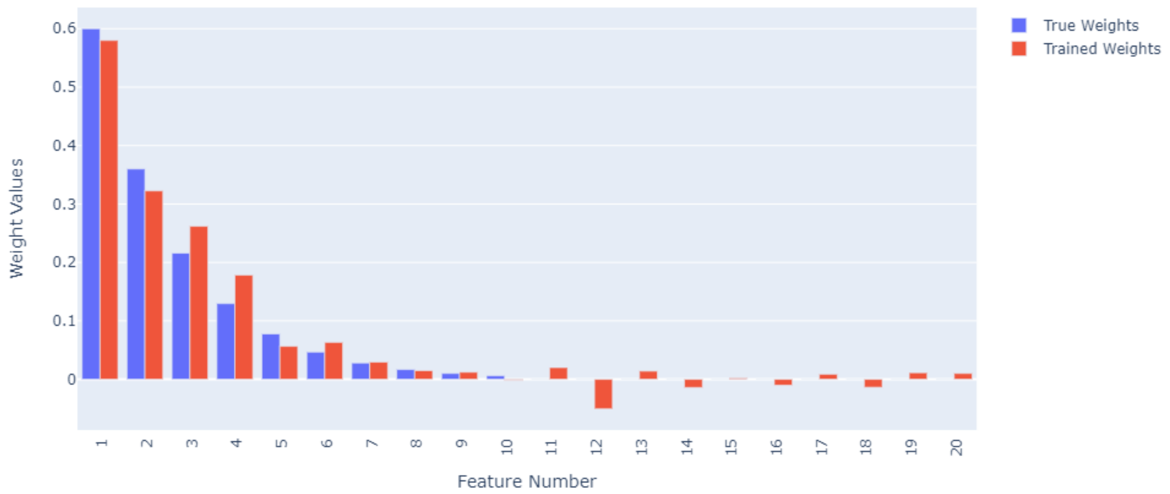
```
# Finding optimal Lambda
optimal_lambda_index = np.argmin(mse_list)
optimal_lambda_value = lambda_list[optimal_lambda_index]
print("Optimal Lambda value: ", optimal_lambda_value)
```

```
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 X_1, which is expected.
The least significant feature could have been anything from X_{11} to X_{20}, but the model found X_{10} and X_{15} as the least significant feature (-0.002 and 0.002 respectively), although some most weights had fairly low values, except for X_{12}.
The true weight for X_{10} 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'))
```

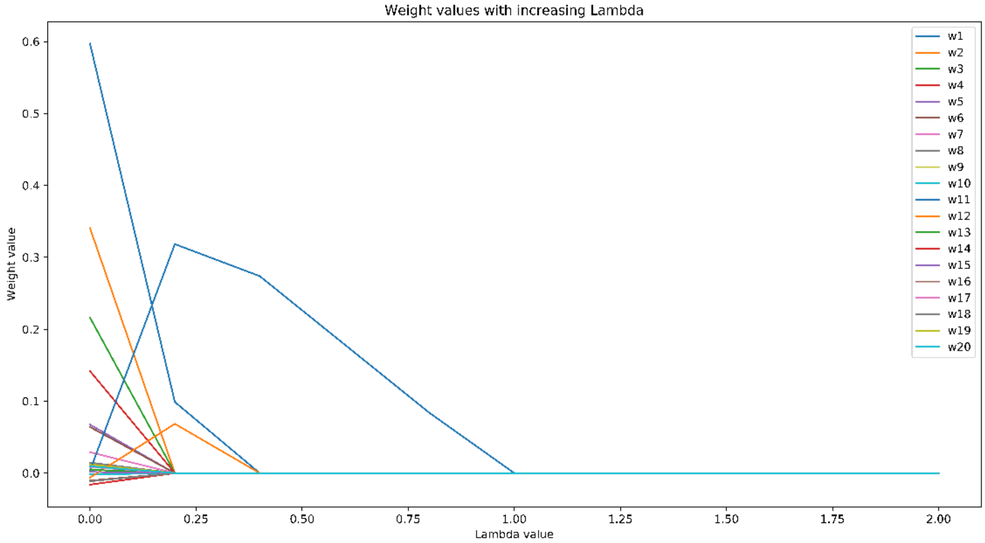
```
fig.update_xaxes(title_text = "Feature Number")
fig.update_yaxes(title_text = "Weight Values")
fig.show()
```

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) w_2 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()
```

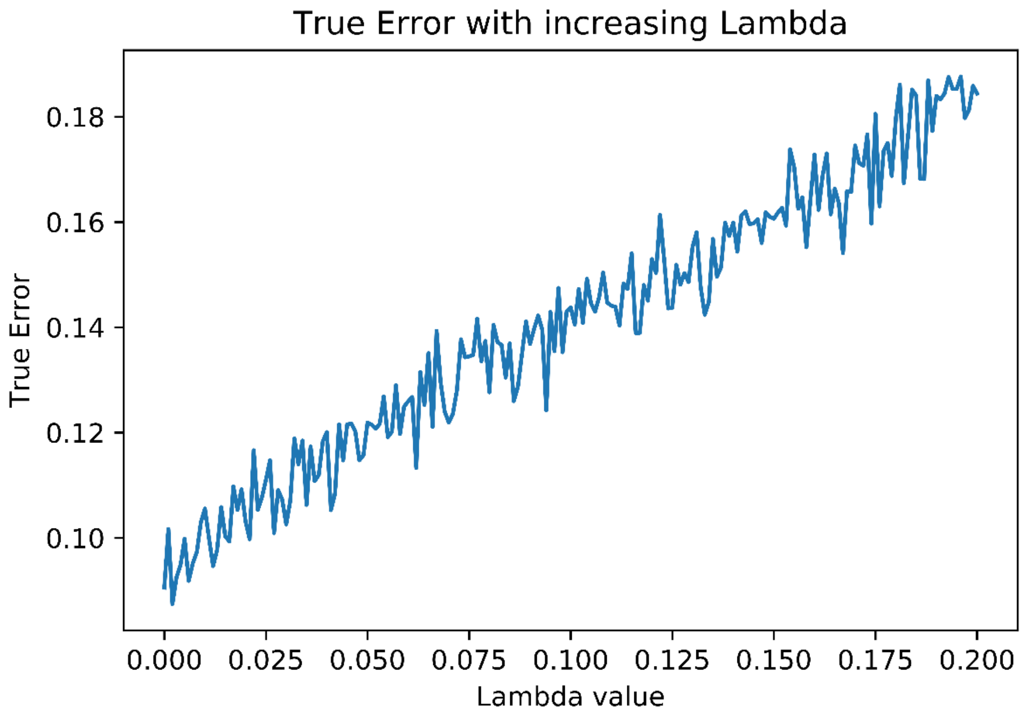


From the above plot we see that features get eliminated (weights become 0) as lambda increases. At around lambda=1, we see all features eliminated.

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)
```

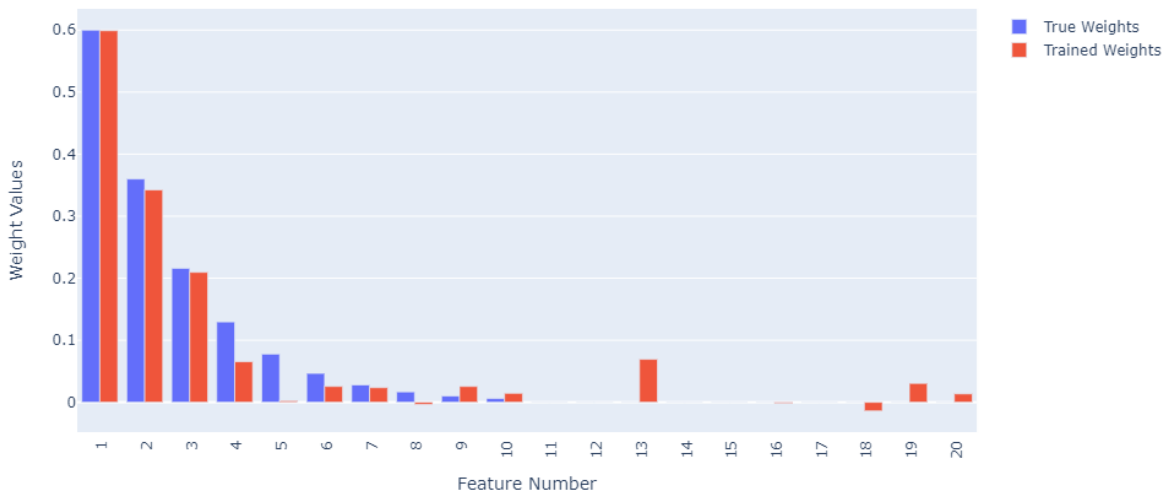
```
Optimal Lambda value:  0.002
```

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
```



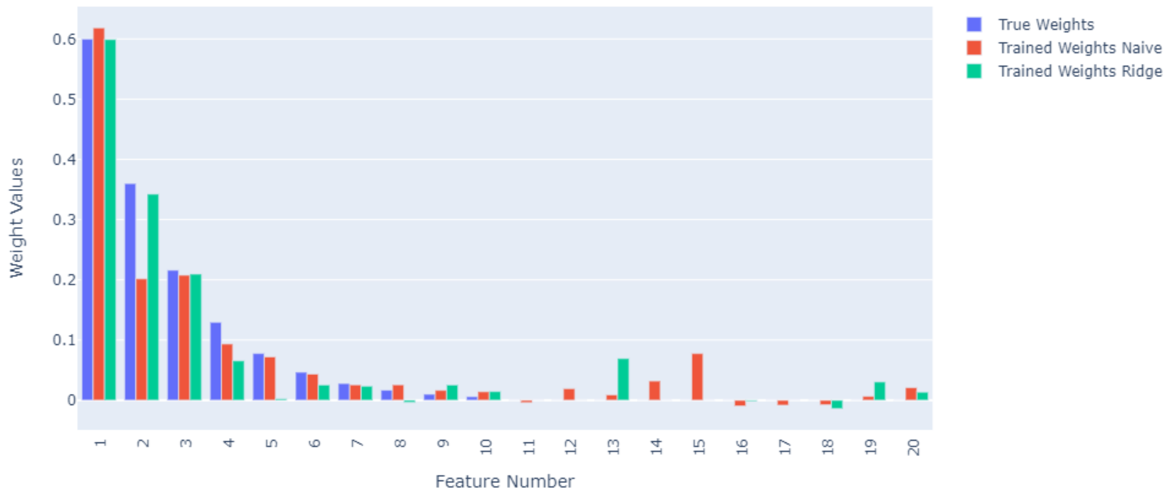

```
True Bias: 10
Trained Bias: 10.004316493840976
```

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 X_1, which is expected.
The least significant feature could have been anything from X_{11} to X_{20}, but the model found X_{11}, X_{12}, X_{14}, X_{15} and X_{17} as the least significant feature (weight 0 in 3 point precision).

The model was able to prune X_{11}, X_{12}, X_{14}, X_{15} and X_{17}

```
# 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 w_1 to w_{10}, although the lasso regression model performs a bit better.

Furthermore, for weights w_{11} to w_{20}, 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.

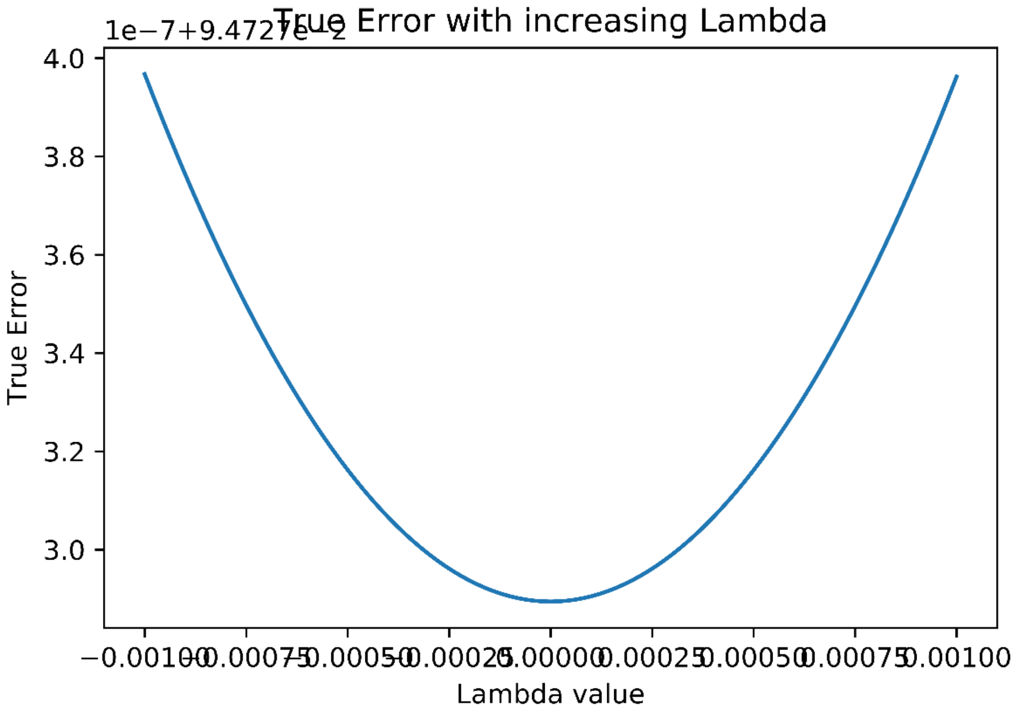
```
# pruning weights lower than 3rd precision point
trained_weights_lasso_precision = [round(num, 3) for num in
trained_weights_lasso]

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).

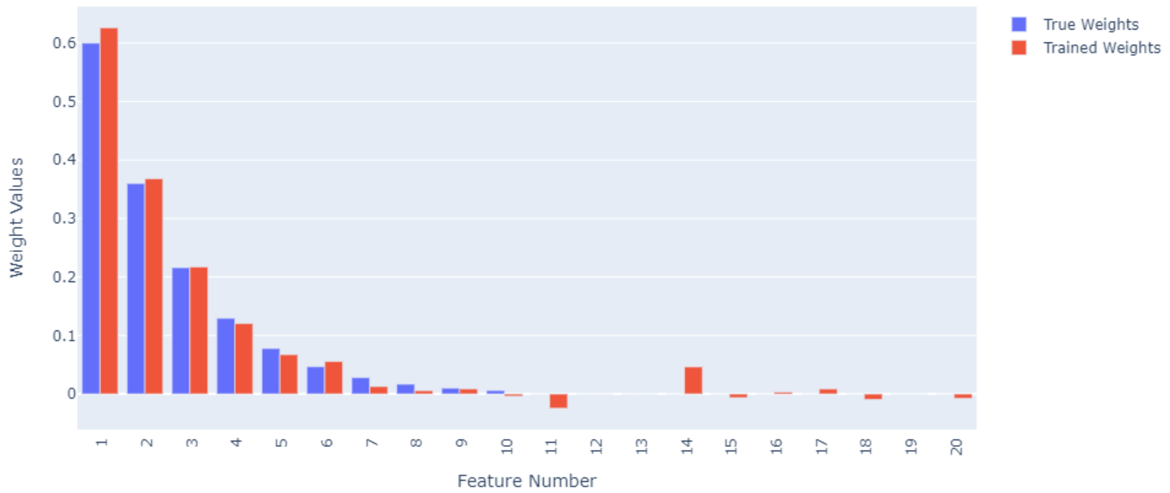
```
# Finding weights and biases with optimal lambda
optimal_w = ridge_model.fit(X_pruned, Y, optimal_lambda_value)

# 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 pruning_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 w_{11} to w_{20} 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 $(\pm 1, \pm 1)$ 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 ϵ_t ranging from 1e-2 to 1e-8 (close to 0), inclusive, with 1000 steps.

The solver finds optimal α_2 to α_m (here α_4) and calculates α_1 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:

$$\max_{\underline{\alpha}} \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i y^i (\underline{x}^i . \underline{x}^j) y^j \alpha_j$$
$$(s.t.) \sum_{i=1}^{m} \alpha_i y^i = 0$$
$$\forall_i : \alpha_i \geq 0$$

Expanding the above:

$$\max_{\alpha_A,\alpha_B,\alpha_C,\alpha_D} [\alpha_A + \alpha_B + \alpha_C + \alpha_D] - \frac{1}{2}[\alpha_A K(A,A)\alpha_A - \alpha_A K(A,B)\alpha_B + \alpha_A K(A,C)\alpha_C - \alpha_A K(A,D)\alpha_D]$$
$$-\frac{1}{2}[-\alpha_B K(B,A)\alpha_A + \alpha_B K(B,B)\alpha_B - \alpha_B K(B,C)\alpha_C + \alpha_B K(B,D)\alpha_D]$$
$$-\frac{1}{2}[\alpha_C K(C,A)\alpha_A - \alpha_C K(C,B)\alpha_B + \alpha_C K(C,C)\alpha_C - \alpha_C K(C,D)\alpha_D]$$
$$-\frac{1}{2}[-\alpha_D K(D,A)\alpha_A + \alpha_D K(D,B)\alpha_B - \alpha_D K(D,C)\alpha_C + \alpha_D K(D,D)\alpha_D]$$

$$(s.t.)\alpha_A - \alpha_B + \alpha_C - \alpha_D = 0$$

$$\alpha_A, \alpha_B, \alpha_C, \alpha_D \geq 0$$

Part 1:
For the Polynomial Kernel: $\boldsymbol{K(\underline{x}, \underline{y}) = (1 + \underline{x}.\underline{y})^2}$

$$\varphi(\underline{x}) = [1, x_1^2, \sqrt{2}x_1x_2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2]^T$$

and

$$\varphi(\underline{y}) = [1, y_1^2, \sqrt{2}y_1y_2, y_2^2, \sqrt{2}y_1, \sqrt{2}y_2]^T$$

X and Y values

A = ((1, 1), -1)
B = ((-1, 1), 1)
C = ((-1, -1), -1)
D = ((1, -1), 1)

$$K(A,A) = (1 + \underline{A}.\underline{A})^2 = (1 + (\mathbf{1},\mathbf{1})\boldsymbol{dot}(\mathbf{1},\mathbf{1}))^2 = (1+1+1)^2 = 9$$
$$K(A,B) = (1 + \underline{A}.\underline{B})^2 = (1 + (\mathbf{1},\mathbf{1})\boldsymbol{dot}(-\mathbf{1},\mathbf{1}))^2 = (1-1+1)^2 = 1$$
$$K(A,C) = (1 + \underline{A}.\underline{C})^2 = (1 + (\mathbf{1},\mathbf{1})\boldsymbol{dot}(-\mathbf{1},-\mathbf{1}))^2 = (1-1-1)^2 = 1$$
$$K(A,D) = (1 + \underline{A}.\underline{D})^2 = (1 + (\mathbf{1},\mathbf{1})\boldsymbol{dot}(\mathbf{1},-\mathbf{1}))^2 = (1+1-1)^2 = 1$$

$$K(B,A) = (1 + \underline{B}.\underline{A})^2 = (1 + (-\mathbf{1},\mathbf{1})\boldsymbol{dot}(\mathbf{1},\mathbf{1}))^2 = (1-1+1)^2 = 1$$
$$K(B,B) = (1 + \underline{B}.\underline{B})^2 = (1 + (-\mathbf{1},\mathbf{1})\boldsymbol{dot}(-\mathbf{1},\mathbf{1}))^2 = (1+1+1)^2 = 9$$
$$K(B,C) = (1 + \underline{B}.\underline{C})^2 = (1 + (-\mathbf{1},\mathbf{1})\boldsymbol{dot}(-\mathbf{1},-\mathbf{1}))^2 = (1+1-1)^2 = 1$$
$$K(B,D) = (1 + \underline{B}.\underline{D})^2 = (1 + (-\mathbf{1},\mathbf{1})\boldsymbol{dot}(\mathbf{1},-\mathbf{1}))^2 = (1-1-1)^2 = 1$$

$$K(C,A) = (1 + \underline{C}.\underline{A})^2 = (1 + (-\mathbf{1},-\mathbf{1})\boldsymbol{dot}(\mathbf{1},\mathbf{1}))^2 = (1-1-1)^2 = 1$$
$$K(C,B) = (1 + \underline{C}.\underline{B})^2 = (1 + (-\mathbf{1},-\mathbf{1})\boldsymbol{dot}(-\mathbf{1},\mathbf{1}))^2 = (1+1-1)^2 = 1$$
$$K(C,C) = (1 + \underline{C}.\underline{C})^2 = (1 + (-\mathbf{1},-\mathbf{1})\boldsymbol{dot}(-\mathbf{1},-\mathbf{1}))^2 = (1+1+1)^2 = 9$$
$$K(C,D) = (1 + \underline{C}.\underline{D})^2 = (1 + (-\mathbf{1},-\mathbf{1})\boldsymbol{dot}(\mathbf{1},-\mathbf{1}))^2 = (1-1+1)^2 = 1$$

$$K(D,A) = (1 + \underline{B}.\underline{A})^2 = (1 + (\mathbf{1},-\mathbf{1})\boldsymbol{dot}(\mathbf{1},\mathbf{1}))^2 = (1+1-1)^2 = 1$$
$$K(D,B) = (1 + \underline{B}.\underline{B})^2 = (1 + (\mathbf{1},-\mathbf{1})\boldsymbol{dot}(-\mathbf{1},\mathbf{1}))^2 = (1-1-1)^2 = 1$$
$$K(D,C) = (1 + \underline{B}.\underline{C})^2 = (1 + (\mathbf{1},-\mathbf{1})\boldsymbol{dot}(-\mathbf{1},-\mathbf{1}))^2 = (1-1+1)^2 = 1$$
$$K(D,D) = (1 + \underline{B}.\underline{D})^2 = (1 + (\mathbf{1},-\mathbf{1})\boldsymbol{dot}(\mathbf{1},-\mathbf{1}))^2 = (1+1+1)^2 = 9$$

Now our objective function becomes:

$$\max_{\alpha_A,\alpha_B,\alpha_C,\alpha_D} [\alpha_A + \alpha_B + \alpha_C + \alpha_D] - \frac{1}{2}[9{\alpha_A}^2 - \alpha_A\alpha_B + \alpha_A\alpha_C - \alpha_A\alpha_D] - \frac{1}{2}[-\alpha_B\alpha_A + 9{\alpha_B}^2 - \alpha_B\alpha_C + \alpha_B\alpha_D]$$
$$- \frac{1}{2}[\alpha_C\alpha_A - \alpha_C\alpha_B + 9{\alpha_C}^2 - \alpha_C\alpha_D] - \frac{1}{2}[-\alpha_D\alpha_A + \alpha_D\alpha_B - \alpha_D\alpha_C + 9{\alpha_D}^2]$$

or,

$$\max_{\alpha_A,\alpha_B,\alpha_C,\alpha_D} \alpha_A + \alpha_B + \alpha_C + \alpha_D$$
$$- \frac{1}{2}[9{\alpha_A}^2 - 2\alpha_A\alpha_B + 2\alpha_A\alpha_C - 2\alpha_A\alpha_D + 9{\alpha_B}^2 - 2\alpha_B\alpha_C + 2\alpha_B\alpha_D + 9{\alpha_C}^2 - 2\alpha_C\alpha_D + 9{\alpha_D}^2]$$

Applying $\frac{\boldsymbol{\partial(Obj\ Func(\underline{\alpha}))}}{\boldsymbol{\partial\alpha_i}} = \mathbf{0}$ **[i=A,B,C,D]**, we get:

$$1 - 9\alpha_A + \alpha_B - \alpha_C + \alpha_D = 0$$
$$1 + \alpha_A - 9\alpha_B + \alpha_C - \alpha_D = 0$$
$$1 - \alpha_A + \alpha_B - 9\alpha_C + \alpha_D = 0$$
$$1 + \alpha_A - \alpha_B + \alpha_C - 9\alpha_D = 0$$

or (rearranging),

$$9\alpha_A - \alpha_B + \alpha_C - \alpha_D = 1$$
$$-\alpha_A + 9\alpha_B - \alpha_C + \alpha_D = 1$$
$$\alpha_A - \alpha_B + 9\alpha_C - \alpha_D = 1$$
$$-\alpha_A + \alpha_B - \alpha_C + 9\alpha_D = 1$$

Solving by Inverse Matrix Method:

A.X=B

$$A = \begin{bmatrix} 9 & -1 & 1 & -1 \\ -1 & 9 & -1 & 1 \\ 1 & -1 & 9 & -1 \\ -1 & 1 & -1 & 9 \end{bmatrix} \qquad \text{B}= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} 11/96 & 1/96 & -1/96 & 1/96 \\ 1/96 & 11/96 & 1/96 & -1/96 \\ -1/96 & 1/96 & 11/96 & 1/96 \\ 1/96 & -1/96 & 1/96 & 11/96 \end{bmatrix}$$

$$X = A^{-1}B = \begin{bmatrix} 11/96 & 1/96 & -1/96 & 1/96 \\ 1/96 & 11/96 & 1/96 & -1/96 \\ -1/96 & 1/96 & 11/96 & 1/96 \\ 1/96 & -1/96 & 1/96 & 11/96 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/8 \\ 1/8 \\ 1/8 \\ 1/8 \end{bmatrix}$$

Therefore, $\alpha_A = \alpha_B = \alpha_C = \alpha_D = 1/8$

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 $Obj\ Func(\underline{\alpha}) = 1/4$

Hence,

$$\frac{1}{2}\|\underline{w}^*\|^2 = \frac{1}{4}$$

or,

$$\|\underline{w}^*\| = \frac{1}{\sqrt{2}}$$

$$\underline{w}^* = \frac{1}{8}\left[-\varphi\left(\underline{y}_A\right) + \varphi\left(\underline{y}_B\right) - \varphi\left(\underline{y}_C\right) + \varphi\left(\underline{y}_D\right)\right]$$

Referencing

$$\varphi\left(\underline{y}\right) = [1, y_1^2, \sqrt{2}y_1y_2, y_2^2, \sqrt{2}y_1, \sqrt{2}y_2]^T$$

and

A = ((1, 1), -1)
B = ((-1, 1), 1)
C = ((-1, -1), -1)
D = ((1, -1), 1)

$$\underline{w}^* = \frac{1}{8}\left[-\begin{bmatrix} 1 \\ 1 \\ \sqrt{2} \\ 1 \\ \sqrt{2} \\ \sqrt{2} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ -\sqrt{2} \\ 1 \\ -\sqrt{2} \\ \sqrt{2} \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ \sqrt{2} \\ 1 \\ -\sqrt{2} \\ -\sqrt{2} \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ -\sqrt{2} \\ 1 \\ \sqrt{2} \\ -\sqrt{2} \end{bmatrix}\right]$$

or,

$$\underline{w}^* = \frac{1}{8}\begin{bmatrix} 0 \\ 0 \\ -4\sqrt{2} \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\sqrt{2}/2 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1/\sqrt{2} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The first element of w* is the bias:

$$bias = w_1^* = 0$$

The separating function is

$$\underline{w}^{*T}\varphi(\underline{x}) = 0$$

or,

$$\begin{bmatrix} 0 & 0 & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ {x_1}^2 \\ \sqrt{2}x_1x_2 \\ {x_2}^2 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \end{bmatrix} = 0$$

or,

$$-x_1x_2 = 0$$

or,

$$classify\ (\underline{x}) = sign(-\boldsymbol{x_1} * \boldsymbol{x_2})$$

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

***