CS536 HW5: Computing Solutions

Student Name: Xuenan Wang

NetID: xw336

1. Linear Regression

Consider data generated in the following way:

- X_1 through X_{10} and X_{16} through X_{20} are i.i.d. standard normals.
- $X_{11} = X_1 + X_2 + N(\mu = 0, \sigma^2 = 0.1)$
- $X_{12} = X_3 + X_4 + N(\mu = 0, \sigma^2 = 0.1)$
- $X_{13} = X_4 + X_5 + N(\mu = 0, \sigma^2 = 0.1)$
- $X_{14} = 0.1X_7 + N(\mu = 0, \sigma^2 = 0.1)$
- $X_{15} = 2X_2 - 10 + N(\mu = 0, \sigma^2 = 0.1)$

The values Y are generated according to the following linear model:

$$Y = 10 + \sum_{i=1}^{10} (0.6)^i X_i + N(\mu = 0, \sigma^2 = 0.1). \quad (1)$$

Note, the variables X_{11} through X_{20} are technically irrelevant.

1) Generate a data set of size $m = 1000$. Solve the naive least squares regression model for the weights and bias that minimize the training error - how do they compare to the true weights and biases? What did your model conclude as the most significant and least significant features - was it able to prune anything? Simulate a large test set of data and estimate the 'true' error of your solved model.

I generated one data point as following:

```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
Data.X: [1, -0.974483613080833, -0.06709793649975916, 0.23039487031539413, 0.9456932547002465, -0.450
1682400711009, -1.2363977273491071, 0.3446942975178719, 0.8126632338945884, 2.3381475440020414, -1.02
05879497782748, -0.6773686518233528, 1.339759967364761, 0.19034491047673802, 0.10115356223514418, -10
.251377471583606, -1.3235691795939541, -1.2211031092577933, -0.42864741423033753, 0.41125624187504733
, -0.5891256059532733]
Data.Y: 8.684574947362133
```

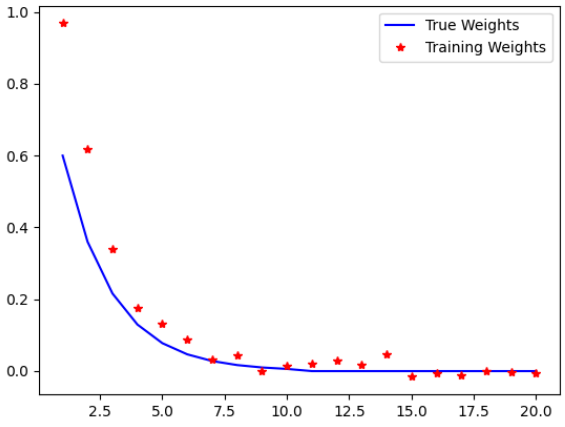
Naïve linear regression is solved by $w = [X^T X]^{-1} X^T y$ when $X^T X$ is reversable and $w = [X^T X + \lambda I]^{-1} X^T y$ when $X^T X$ is not reversable. I set $\lambda = 0.1$ here.

With training data size of 1000, I solve naive least squares regression model as following:

```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
--------Training with data size of 1000 --------
model.w: [ 9.84493956e+00  9.68488280e-01  6.18237503e-01  3.39439353e-01
  1.76593561e-01  1.32159349e-01  8.76135650e-02  3.29396700e-02
  4.40241538e-02 -1.41162441e-04  1.55698646e-02  2.06843486e-02
  2.79121861e-02  1.68308725e-02  4.74536834e-02 -1.58239410e-02
 -5.37224286e-03 -1.13699922e-02 -1.13066584e-03 -1.82874219e-03
 -4.56195019e-03]
model.bias: 10.067345281787926
trainError: 0.08805431917404738
--------Testing with data size of 1000000 --------
testError: 0.10196749486048333
```

The true(theoretical) bias is 10, and my experimental result is 9.84.

To further demonstrate the difference between training weights and real weights(theoretical), I plotted it out:



We can see that the most significant feature is X_1 and least significant feature is X_{15}. This makes sense because we know that the model is mostly affected by X_1 and for features X_{11}-X_{20}, they are irrelevant, so essentially, it's just noise.

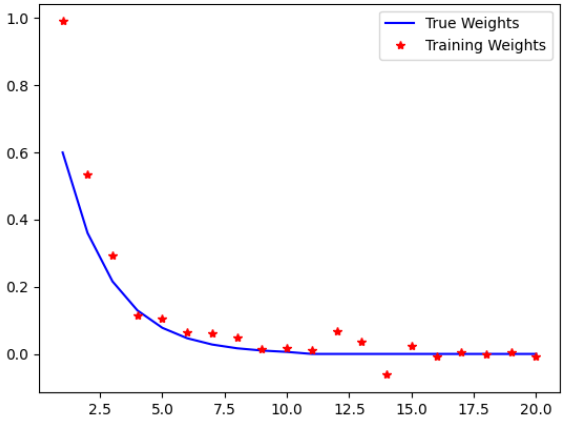
To prune this model, I think it's reasonable to pick features with smaller indexes, which means we focus more on X_1 and less focus on X_{11}-X_{20}. It's like PCA, we pick the principal features. By doing this, we are abandoning those features with less contribution to our final results. However, this idea is basically lasso regression where we set unimportant features' weights to zeros. So it's reasonable to say that following experiments are the pruning implementation I have in mind.

2) Write a program to take a data set of size m and a parameter λ, and solve for the ridge regression model for that data. Write another program to take the solved model and estimate the true error by evaluating that model on a large test data set. For data sets of size $m = 1000$, plot estimated true error of the ridge regression model as a function of λ. What is the optimal λ to minimize testing error? What are the weights and biases ridge regression gives at this λ, and how do they compare to the true weights? What did your model conclude as the most significant and least significant features - was it able to prune anything? How does the optimal ridge regression model compare to the naive least squares model?

Ridge regression is solved by $w = [X^T X + \lambda I]^{-1} X^T y$. It's essentially a naïve least squares regression with a L2-norm penalty.

When I set $\lambda = 0$, which means there are no penalties, the ridge regression is same as naive linear regression.

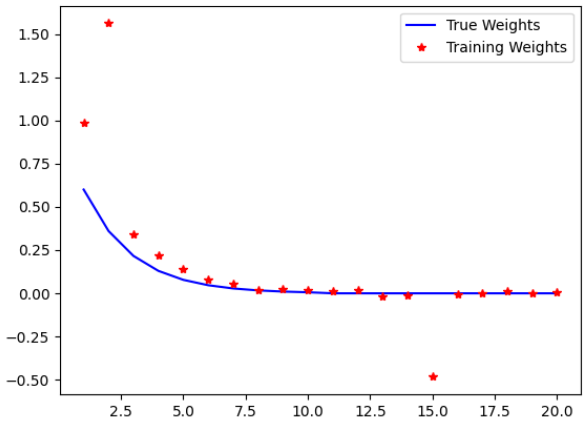
```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
--------Training with data size of 1000 and lambda of 0 --------
model.w: [ 1.02235250e+01  9.90722053e-01  5.34971888e-01  2.94385553e-01
  1.14044591e-01  1.04520754e-01  6.46628419e-02  6.25475805e-02
  4.90030811e-02  1.50877500e-02  1.69024715e-02  1.11269714e-02
  6.63115885e-02  3.76983796e-02 -6.20658416e-02  2.31370112e-02
 -7.76502613e-03  6.19981130e-03 -2.34985757e-03  4.71421444e-03
 -7.11582343e-03]
model.bias: 9.98140880853826
trainError: 0.09618926388548343
--------Testing with data size of 1000000 --------
testError: 0.10199651214309621
```



Here we have experimental bias as 10.22 and the real bias is 10. Test error is 0.101 and train error is 0.096.

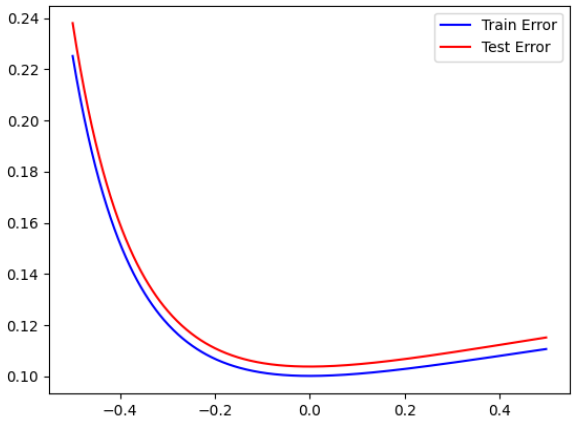
Now, if I set $\lambda = 1$, the result will be different because in this time, we have a penalty.

```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
--------Training with data size of 1000 and lambda of 1 --------
model.w: [ 5.16455654e+00  9.87848835e-01  1.56147468e+00  3.38186172e-01
  2.16822325e-01  1.42561918e-01  8.09655332e-02  5.25038102e-02
  1.63087303e-02  2.61423577e-02  1.83357013e-02  1.28387428e-02
  2.06485769e-02 -1.62925644e-02 -1.36601892e-02 -4.82653654e-01
 -4.10554906e-03  4.86676004e-04  1.08926893e-02 -2.46588770e-03
  8.29717961e-03]
model.bias: 9.97867517569882
trainError: 0.11867642675249124
--------Testing with data size of 1000000 --------
testError: 0.12437630264963717
```



We can see that comparing to naïve model, weights from ridge regression are closer to true weights. For those important features, the weights are more significant. Although I cannot explain the high value of feature #2 and low value of feature #15. This is wired because I have repeated this experiment multiple times and they all looks like this one.

The experimental bias here is 5.25 when the real bias is still 10.




```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
minTrainError is 0.10020061509034037 at lambda= 4.440892098500626e-16
minTestError is 0.10385163853100274 at lambda= 4.440892098500626e-16
```

Above is the plot of errors over lambda. I tried plot this from 0 to 5 with step of 0.2 and I found that the best lambda we can choose was 0. After that, I also tried multiple selections and they all gave me lambda equals to 0.

After some confusing time, I read the discussion board on Canvas and it really helped me out. I adjust my lambda to -0.5 to 0.5 with step of 0.002. Above is what I got.

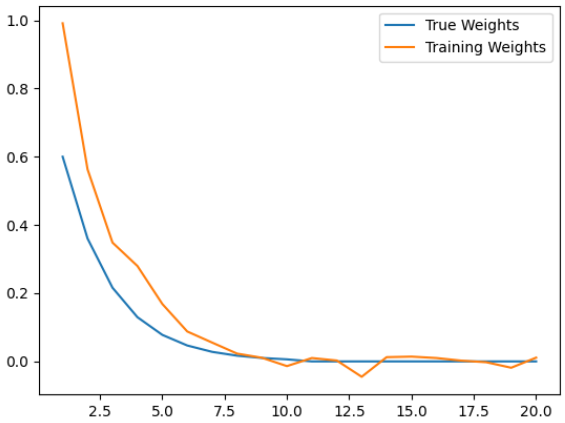
We can see that the optimal lambda (4.44×10^{-16}) is extremely close to zero. And both training error and test error reached their minimal value at same lambda.

3) Write a program to take a data set of size m and a parameter λ, and solve for the Lasso regression model for that data. For a data set of size $m = 1000$, show that as λ increases, features are effectively eliminated from the model until all weights are set to zero.

Lasso regression can be solved by $w = [X^T X + \lambda I]^{-1}[X^T y]$. It's similar to ridge regression but with a L1-norm penalty.

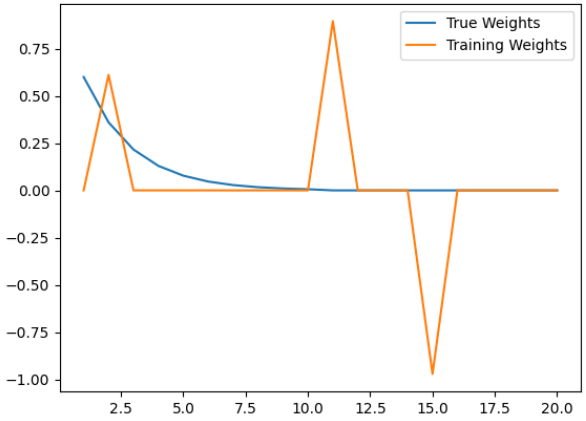
Firstly, with $\lambda = 0$, lasso regression should be same as naïve linear regression.

```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
--------Training with data size of 1000 and lambda of 0 --------
model.w: [ 1.01316553e+01  9.91127301e-01  5.62711924e-01  3.48298233e-01
  2.79652680e-01  1.68309105e-01  8.77119232e-02  5.51499843e-02
  2.28928444e-02  1.06069873e-02 -1.37716114e-02  9.90648963e-03
  2.60912313e-03 -4.50307081e-02  1.24561213e-02  1.40454535e-02
  9.92778759e-03  2.25057186e-03 -2.39454422e-03 -1.85623541e-02
  1.10167948e-02]
model.bias: 9.944395476755068
trainError: 0.24547864663673324
--------Testing with data size of 1000000 --------
testError: 0.25520540940249553
```



With $\lambda = 1$, I'm having a L1-norm penalty here. The results are as following:

```
xw336@ilab1:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
--------Training with data size of 1000 and lambda of 1 --------
model.w: [ 0.          0.          0.61122437  0.          0.          0.
  0.          0.          0.          0.          0.          0.89582218
  0.          0.          0.         -0.97146424  0.          0.
  0.          0.          0.        ]
model.bias: 10.00733097726535
trainError: 1.011016744733839
--------Testing with data size of 1000000 --------
testError: 1.0232243591348766
```



We can see that with lasso regression and $\lambda \neq 0$, the non-important features are pushing down to zeros. To magnify this, I plot the weights change over λ. Following is the result:

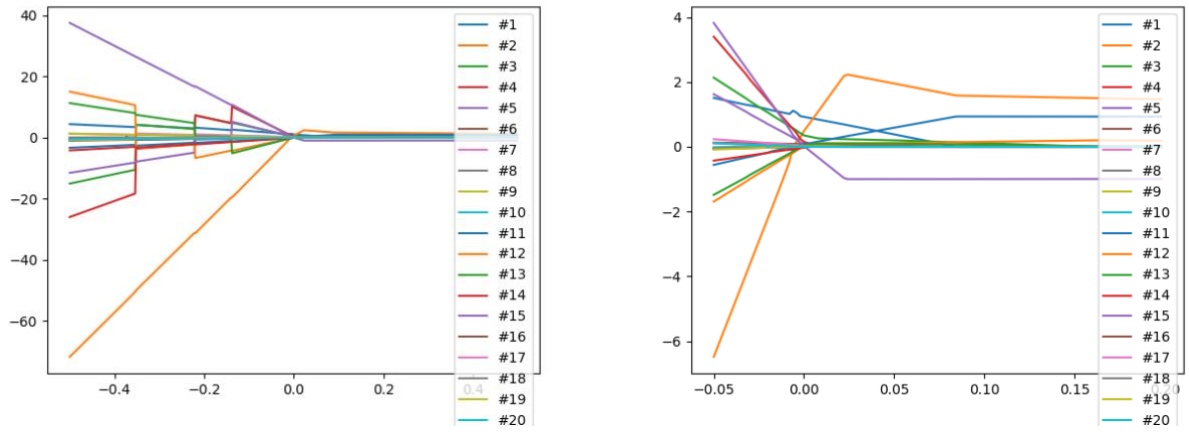
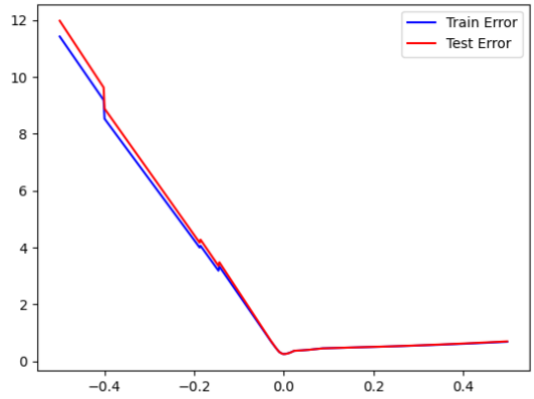


Fig. Weights over λ, start:-0.5 stop:0.5 step: 0.002(left) and start:-0.05 stop:0.2 step: 0.002 (right)

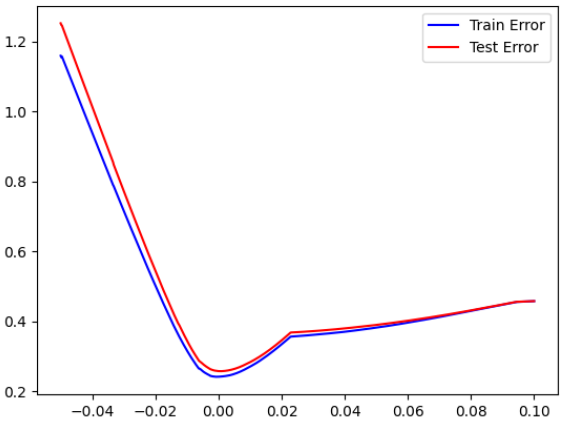
We can clearly see the trend that weights are pushing down to zeros.

4) For data sets of size $m = 1000$, plot estimated true error of the lasso regression model as a function of λ. What is the optimal λ to minimize testing error? What are the weights and biases lasso regression gives at this λ, and how do they compare to the true weights? What did your model conclude as the most significant and least signficiant features - was it able to prune anything? How does the optimal regression model compare tot he naive least squares model?

Similar as the ridge regression question, I computer lambda over [-0.5, 0.5] with step of 0.002. Following are my results:



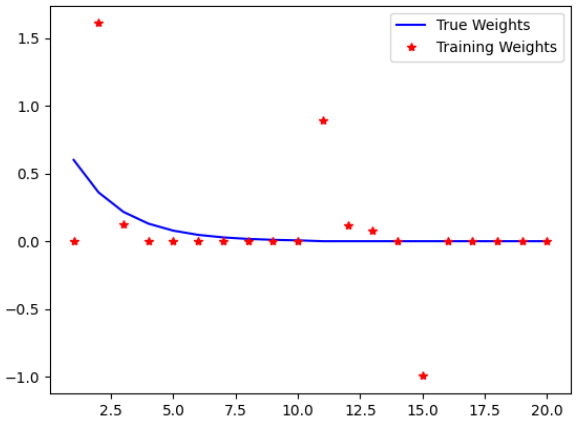
It's not clear to see. So I changed the range to [-0.05, 0.1] with step of 0.0002. Following are the results:




```
minTrainError is 0.24151841350596248 at lambda= -0.0006000000000003017
minTestError is 0.2575178462320199 at lambda= 0.0005999999999996911
```

My result shows that at $\lambda = 0.00059$ we have a minimal test error of 0.257.

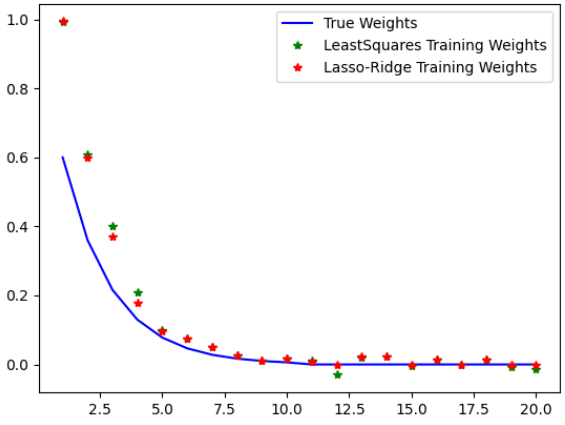
The weights are as following:



Based on my experiment, the most significant feature is X_2 and least significant feature is X_{15}. Comparing to least squares regression, the result I have here have less features contributing to the final result. I think it's basically good, but the low value of X_1 and high value of X_{11} are confusing.

5) Consider using lasso as a means for feature selection: on a data set of size $m = 1000$, run lasso regression with the optimal regularization constant from the previous problems, and identify the set of relevant features; then run ridge regression to fit a model to only those features. *How can you determine a good ridge regression regularization constant to use here?* How does the resulting lasso-ridge combination model compare to the naive least squares model? What features does it conclude are significant or relatively insignificant? How do the testing errors of these two models compare?

```
xw336@ilab2:~/MachineLearning$ /usr/bin/python3 /ilab/users/xw336/MachineLearning/LinearRegression.py
--------Training LeastSquares with data size of 1000 --------
LeastSquares train error: 0.10026361281833239
LeastSquares.w: [ 9.93726098e+00  9.90967029e-01  6.08012822e-01  4.02133189e-01
  2.09102243e-01  1.00298243e-01  7.40442489e-02  4.96436169e-02
  2.50893856e-02  1.18190753e-02  1.73410797e-02  1.09624305e-02
 -2.99039838e-02  2.15456713e-02  2.22489321e-02 -4.83824720e-03
  1.26923241e-02 -1.96299189e-03  1.27532150e-02 -7.20552816e-03
 -1.36142835e-02]
--------Training Lasso with data size of 1000 and lambda of 0.0005999999999996911 --------
Lasso train error: 0.25465250133277556
Lasso.w: [ 9.68412267e+00  9.86444901e-01  6.53146439e-01  3.80919239e-01
  1.80946435e-01  9.22068643e-02  7.35188862e-02  4.99479472e-02
  2.44355238e-02  1.10883927e-02  1.62003854e-02  1.55479502e-02
 -9.24208192e-03  2.85420947e-02  1.54865121e-02 -3.00540111e-02
  1.23341835e-02 -1.38953193e-03  1.18647372e-02 -6.70847616e-03
 -1.34279751e-02]
Significant features: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 16, 18]
--------Training Ridge with data size of 1000 and lambda of 4.440892098500626e-16 --------
Ridge train error: 0.10060576498008239
Ridge.w: [9.98545715e+00 9.94316556e-01 6.00436370e-01 3.71540734e-01
 1.77255148e-01 9.74420209e-02 7.38242961e-02 4.98422269e-02
 2.51667103e-02 1.14461890e-02 1.78115784e-02 8.37782527e-03
 0.00000000e+00 2.38358827e-02 2.29204247e-02 0.00000000e+00
 1.27191108e-02 0.00000000e+00 1.34841644e-02 0.00000000e+00
 0.00000000e+00]
--------Testing LeastSquares with data size of 1000000 --------
testError: 0.10166895097614252
--------Testing Lasso-Ridge with data size of 1000000 --------
testError: 0.10136061866055446
```



I picked the optimal lambda value (4.440892098500626e-16) from previous ridge model experiment as this ridge lambda value.

For least squares regression, the bias is 9.937, train error is 0.10;

For lasso-ridge regression, the bias is 9.985, train error is 0.10.

Test error for least squares regression is 0.1016, and for lasso-ridge regression is 0.1013. Lasso-ridge regression is better.

From above plot, we can clearly see that lasso-ridge regression is closer to true weights, which shows that my lasso-ridge model improved over least squares regression. The most significant feature is X_1, and least significant feature is X_{20}. That's exactly right.

2. SVMs

1) Implement a barrier-method dual SVM solver. How can you (easily!) generate an initial feasible $\underline{\alpha}$ solution away from the boundaries of the constraint region? How can you ensure that you do not step outside the constraint region in any update step? How do you choose your ϵ_t? Be sure to return all α_i including α_1 in the final answer.

I started by initial all α equals to 1. And I set a target loss and max iteration steps as the termination condition. To choose a ϵ, I started by random pick and see how it goes. After a few trials, I found that 0.01 is a reasonable pick. I will elaborate more on experiment in the third question.

2) Use your SVM solver to compute the dual SVM solution for the XOR data using the kernel function $K(\underline{x}, \underline{y}) = (1 + \underline{x}.\underline{y})^2$. Solve the dual SVM by hand to check your work.

We have $K\left(\underline{x}, \underline{y}\right) = \left(1 + \underline{x} \cdot \underline{y}\right)^2 = \phi(\underline{x}) \cdot \phi\left(\underline{y}\right)$. For clarity, I replace $\underline{x}, \underline{y}$ with $\underline{a}, \underline{b}$. Then we got:

$$\phi(\underline{a}) = \phi(x_1, x_2) = \left[1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2\right]^T$$

The objective function is $L_D(\alpha) = \alpha_1 + \alpha_2 + \alpha_3 + \alpha_4 - \frac{1}{2}\sum_{i=1}^{4}\sum_{j=1}^{4}\alpha_i\alpha_j y_i y_j k_{ij}$ where $k_{ij} = K\left(\underline{x}_i, \underline{x}_j\right) = \left(1 + \underline{x}_i^T \cdot \underline{x}_j\right)^2 = 1 + x_{i1}^2x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2x_{j2}^2 + 2x_{i1}x_{j1} + 2x_{i2}x_{j2}$ with constraints $\sum_{i=0}^{n}\alpha_i y_i = 0$ and $\forall\, i, \alpha_i \geq 0$.

It's easy to calculate that:

$$K = \begin{bmatrix} 9 & 1 & 1 & 1 \\ 1 & 9 & 1 & 1 \\ 1 & 1 & 9 & 1 \\ 1 & 1 & 1 & 9 \end{bmatrix}$$

We can computer partial derivative of objective function and set to 0:

$$\frac{\partial L_D(\alpha)}{\partial \alpha_1} = 1 - (9\alpha_1 - \alpha_2 - \alpha_3 + \alpha_4) = 0$$

$$\frac{\partial L_D(\alpha)}{\partial \alpha_2} = 1 - (-\alpha_1 + 9\alpha_2 + \alpha_3 - \alpha_4) = 0$$

$$\frac{\partial L_D(\alpha)}{\partial \alpha_3} = 1 - (-\alpha_1 + \alpha_2 + 9\alpha_3 - \alpha_4) = 0$$

$$\frac{\partial L_D(\alpha)}{\partial \alpha_4} = 1 - (\alpha_1 - \alpha_2 - \alpha_3 + 9\alpha_4) = 0$$

And the result can be easily calculated: $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.125$

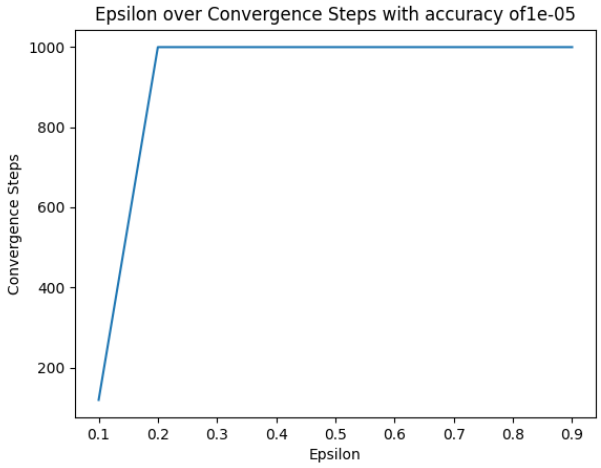
3) Given the solution your SVM solver returns, reconstruct the primal classifier and show that it correctly classifies the XOR data.

My dual SVM implementation of XOR data is as following:

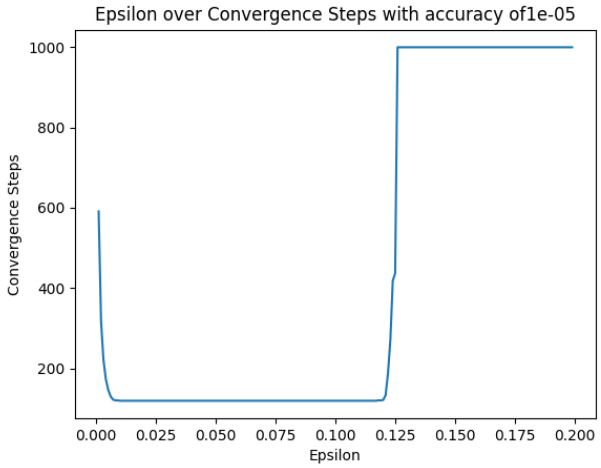
```
xw336@ilab3:~/MachineLearning$ python3 SVM.py
--------Training DualSVMs--------
Training Data:
X:
 [[-1 -1]
 [-1  1]
 [ 1  1]
 [ 1 -1]]
Y:
 [-1  1 -1  1]
alpha: [0.125114682692284, 0.125137772051570, 0.125098380027310, 0.125075290668024]
convergence step: 119
```

The α I got is basically [0.125, 0.125, 0.125, 0.125], which is exactly the solution I want.

To find an optimal ϵ, I did some experiments. From previous guess trails of ϵ, I already know that 1 is not an option, therefore plotted convergence steps over ϵ in the range of [0.1, 1] with step of 0.01. Following is my result:



The maximum iteration steps I set is 1000, so the max convergence steps on the above figure is 1000. Note that it doesn't mean the step of convergence is it touch the max line. Afterwards, I magnified the [0.1, 0.2] range. This time, I plot in the range of [0.001, 0.2] with step of 0.001. Following is the result:



It's clear that roughly in the range of [0.02, 0.11], the convergence steps are all the same. After further measurement, this convergence step is 119.

Source Code (Python 3)

LinearRegression.py

```
import numpy as np
from math import sqrt
from matplotlib import pyplot as plt
import itertools
import heapq

class Data:

    def __init__(self):
        self.X = [0 for _ in range(20)]
        self.X[:10] = list(np.random.standard_normal(10))
        self.X[10] = self.X[0] + self.X[1] + np.random.normal(0, sqrt(0.1))
        self.X[11] = self.X[2] + self.X[3] + np.random.normal(0, sqrt(0.1))
        self.X[12] = self.X[3] + self.X[4] + np.random.normal(0, sqrt(0.1))
        self.X[13] = 0.1 * self.X[6] + np.random.normal(0, sqrt(0.1))
        self.X[14] = 2 * self.X[1] - 10 + np.random.normal(0, sqrt(0.1))
        self.X[15:] = list(np.random.standard_normal(5))
        self.Y = 10 + sum([0.6 ** i * self.X[i] for i in range(10)]) + np.random.normal(0, sqrt(0.1))
        self.X = [1] + self.X


class DataSet:

    def __init__(self, m):
        self.datas = [Data() for _ in range(m)]
        self.X = np.array([d.X for d in self.datas])
        self.Y = np.array([d.Y for d in self.datas])


class LinearRegression:

    def __init__(self, X, Y):
```

```
        self.w = np.dot(np.dot(self.inverse(X), X.T), Y)

    def inverse(self, X):
        matrix = np.dot(X.T, X)
        try:
            inversedMatrix = np.linalg.inv(matrix)
        except np.linalg.LinAlgError:
            inversedMatrix = inverse(matrix + np.eye(matrix.shape[0]) * 0.1)
        return inversedMatrix

    def predict(self, X):
        return np.dot(X, self.w)


class RidgeRegression:

    def __init__(self, X, Y, paramLambda):
        self.w = np.array(np.dot(np.dot(np.matrix(np.dot(X.T, X) +\
            np.eye(np.dot(X.T, X).shape[0]) * paramLambda).I, X.T), Y))[0]

    def predict(self, X):
        return np.dot(X, self.w)


# class LassoRegression:

#     def __init__(self, X, Y, paramLambda):
#         self.w = np.array(np.dot(np.matrix(np.dot(X.T, X) + np.eye(np.dot(X.T, X).shape[0]) *\
#             paramLambda).I, np.dot(X.T, Y)))[0]

#     def predict(self, X):
#         return np.dot(X, self.w)


class LassoRegression:
```

```
    def __init__(self, X, Y, paramLambda):
        self.paramLambda = paramLambda
        self.trainRound = 1000
        self.w = self.fit(X, Y)

    def softThreshold(self, arg1, arg2):
        if (arg1 < -arg2):
            return arg1 + arg2
        elif (arg1 > arg2):
            return arg1 - arg2
        else:
            return 0

    def fit(self, X, Y):
        beta = np.zeros(X.shape[1])
        for i in range(self.trainRound):
            for j in range(len(beta)):
                beta_tmp = beta.copy()
                beta_tmp[j] = 0.0
                r_j = Y - np.dot(X, beta_tmp)
                arg1 = np.dot(X[:, j], r_j)
                arg2 = self.paramLambda * X.shape[0]
                beta[j] = self.softThreshold(arg1, arg2) / (X[:, j]**2).sum()
        return beta

    def predict(self, X):
        return np.dot(X, self.w)


if __name__ == "__main__":
    # # Generate data
    # data = Data()
    # print("Data.X:", data.X)
    # print("Data.Y:", data.Y)

    # # Question 1-1
```

```
# trainSize = 1000
# testSize = 1000000
# print("--------Training with data size of", trainSize, "--------")
# trainData = DataSet(trainSize)
# model = LinearRegression(trainData.X, trainData.Y)
# trainError = 0
# for i in range(trainSize):
#     trainError += (model.predict(trainData.X[i])-trainData.Y[i]) ** 2
# trainError /= trainSize
# print("model.w:", model.w)
# print("model.bias:", np.mean(trainData.Y))
# print("trainError:", trainError)
# trueWeights = [0.6 ** i for i in range(1, 11)]
# trueWeights += [0 for _ in range(10)]
# scalex = list(range(1, 21))
# plt.plot(scalex, trueWeights, 'b', label="True Weights")
# plt.plot(scalex, model.w[1:], 'r*', label="Training Weights")
# plt.legend()
# plt.savefig("LinearRegression-1.png")
# print("--------Testing with data size of", testSize, "--------")
# testData = DataSet(testSize)
# testError = 0
# for i in range(testSize):
#     testError += (model.predict(testData.X[i])-testData.Y[i]) ** 2
# testError /= testSize
# print("testError:", testError)

# # Question 1-2
# trainSize = 1000
# testSize = 1000000
# paramLambda = 1
# print("--------Training with data size of", trainSize, "and lambda of", paramLambda, "--------")
# trainData = DataSet(trainSize)
# model = RidgeRegression(trainData.X, trainData.Y, paramLambda)
# trainError = 0
# for i in range(trainSize):
```

```
#     trainError += (model.predict(trainData.X[i])-trainData.Y[i]) ** 2
# trainError /= trainSize
# print("model.w:", model.w)
# print("model.bias:", np.mean(trainData.Y))
# print("trainError:", trainError)
# trueWeights = [0.6 ** i for i in range(1, 11)]
# trueWeights += [0 for _ in range(10)]
# scalex = list(range(1, 21))
# plt.plot(scalex, trueWeights, 'b', label="True Weights")
# plt.plot(scalex, model.w[1:], 'r*', label="Training Weights")
# plt.legend()
# figName = "RidgeRegression-" + str(trainSize) + "-" + str(paramLambda) + ".png"
# plt.savefig(figName)
# print("--------Testing with data size of", testSize, "--------")
# testData = DataSet(testSize)
# testError = 0
# for i in range(testSize):
#     testError += (model.predict(testData.X[i])-testData.Y[i]) ** 2
# testError /= testSize
# print("testError:", testError)

# trainSize = 1000
# testSize = 10000
# testErrorArray = []
# trainErrorArray = []
# lambdaArray = []
# trainData = DataSet(trainSize)
# testData = DataSet(testSize)
# minTrainError = [1000, 0]
# minTestError = [1000, 0]
# startLambda = -0.5
# stopLambda = 0.5
# stepLambda = 0.002
# for cur in np.arange(startLambda, stopLambda, stepLambda):
#     model = RidgeRegression(trainData.X, trainData.Y, cur)
#     lambdaArray.append(cur)
```

```
#     trainError = 0
#     for i in range(trainSize):
#         trainError += (model.predict(trainData.X[i])-trainData.Y[i]) ** 2
#     trainError /= trainSize
#     if minTrainError[0] >= trainError:
#         minTrainError = [trainError, cur]
#     trainErrorArray.append(trainError)
#     testError = 0
#     for i in range(testSize):
#         testError += (model.predict(testData.X[i])-testData.Y[i]) ** 2
#     testError /= testSize
#     if minTestError[0] >= testError:
#         minTestError = [testError, cur]
#     testErrorArray.append(testError)
# # print("lambdaArray:", lambdaArray)
# # print("testErrorArray:", testErrorArray)
# # print("trainErrorArray:", trainErrorArray)
# print("minTrainError is", minTrainError[0], "at lambda=", minTrainError[1])
# print("minTestError is", minTestError[0], "at lambda=", minTestError[1])
# plt.plot(lambdaArray, trainErrorArray, "b", label="Train Error")
# plt.plot(lambdaArray, testErrorArray, "r", label="Test Error")
# plt.legend()
# fileName = "RidgeRegression-TestError-TrainError-train" + str(trainSize) + "-test" +\
#       str(testSize) + "start" + str(startLambda) + "stop" + str(stopLambda) + "step" +\
#           str(stepLambda) + ".png"
# plt.savefig(fileName)

# # Question 1-3
# trainSize = 1000
# testSize = 1000000
# paramLambda = 1
# print("--------Training with data size of", trainSize, "and lambda of", paramLambda, "--------")
# trainData = DataSet(trainSize)
# model = LassoRegression(trainData.X, trainData.Y, paramLambda)
# trainError = 0
# for i in range(trainSize):
```

```
#     trainError += abs(model.predict(trainData.X[i])-trainData.Y[i])
# trainError /= trainSize
# print("model.w:", model.w)
# print("model.bias:", np.mean(trainData.Y))
# print("trainError:", trainError)
# trueWeights = [0.6 ** i for i in range(1, 11)]
# trueWeights += [0 for _ in range(10)]
# scalex = list(range(1, 21))
# plt.plot(scalex, trueWeights, label="True Weights")
# plt.plot(scalex, model.w[1:], label="Training Weights")
# plt.legend()
# figName = "LassoRegression-" + str(trainSize) + "-" + str(paramLambda) + ".png"
# plt.savefig(figName)
# print("--------Testing with data size of", testSize, "--------")
# testData = DataSet(testSize)
# testError = 0
# for i in range(testSize):
#     testError += abs(model.predict(testData.X[i])-testData.Y[i])
# testError /= testSize
# print("testError:", testError)

# trainSize = 1000
# testSize = 10000
# testErrorArray = []
# trainErrorArray = []
# lambdaArray = []
# startLambda = -0.05
# stopLambda = 0.2
# stepLambda = 0.002
# trainData = DataSet(trainSize)
# weights = [[] for _ in range(20)]
# for cur in np.arange(startLambda, stopLambda, stepLambda):
#     model = LassoRegression(trainData.X, trainData.Y, cur)
#     lambdaArray.append(cur)
#     for k in range(20):
#         weights[k].append(model.w[k+1])
```

```
# print("lambdaArray:", lambdaArray)
# for k in range(20):
#     plt.plot(lambdaArray, weights[k], label="#"+str(k+1))
# plt.legend()
# fileName = "LassoRegression-FeaturesOverLambda-train" + str(trainSize) + "-test" +\
#      str(testSize) + "start" + str(startLambda) + "stop" + str(stopLambda) +\
#          "step" + str(stepLambda) + ".png"
# plt.savefig(fileName)

# # Question 1-4
# trainSize = 1000
# testSize = 100000
# testErrorArray = []
# trainErrorArray = []
# lambdaArray = []
# startLambda = -0.05
# stopLambda = 0.1
# stepLambda = 0.0002
# minTrainError = [1000, 0]
# minTestError = [1000, 0]
# trainData = DataSet(trainSize)
# testData = DataSet(testSize)
# for cur in np.arange(startLambda, stopLambda, stepLambda):
#     model = LassoRegression(trainData.X, trainData.Y, cur)
#     lambdaArray.append(cur)
#     trainError = 0
#     for i in range(trainSize):
#         trainError += abs(model.predict(trainData.X[i])-trainData.Y[i])
#     trainError /= trainSize
#     if minTrainError[0] >= trainError:
#         minTrainError = [trainError, cur]
#     trainErrorArray.append(trainError)
#     testError = 0
#     for i in range(testSize):
#         testError += abs(model.predict(testData.X[i])-testData.Y[i])
#     testError /= testSize
```

```
#     if minTestError[0] >= testError:
#         minTestError = [testError, cur]
#     testErrorArray.append(testError)
# # print("lambdaArray:", lambdaArray)
# # print("testErrorArray:", testErrorArray)
# # print("trainErrorArray:", trainErrorArray)
# print("minTrainError is", minTrainError[0], "at lambda=", minTrainError[1])
# print("minTestError is", minTestError[0], "at lambda=", minTestError[1])
# plt.plot(lambdaArray, trainErrorArray, "b", label="Train Error")
# plt.plot(lambdaArray, testErrorArray, "r", label="Test Error")
# plt.legend()
# fileName = "LassoRegression-TestError-TrainError-train" + str(trainSize) + "-test" +\
#      str(testSize) + "start" + str(startLambda) + "stop" + str(stopLambda) +\
#          "step" + str(stepLambda) + ".png"
# plt.savefig(fileName)

# trueWeights = [0.6 ** i for i in range(1, 11)]
# trueWeights += [0 for _ in range(10)]
# scalex = list(range(1, 21))
# plt.clf()
# plt.plot(scalex, trueWeights, 'b', label="True Weights")
# plt.plot(scalex, model.w[1:], 'r*', label="Training Weights")
# plt.legend()
# figName = "LassoRegression-" + "lambda" + str(minTestError[1]) + "-" + str(trainSize) + ".png"
# plt.savefig(figName)

# Question 1-5
trainSize = 1000
testSize = 1000000
lassoLambda = 0.0005999999999999911
trainData = DataSet(trainSize)

print("--------Training LeastSquares with data size of", trainSize, "--------")
leastSquares = LinearRegression(trainData.X, trainData.Y)
LSTrainError = 0
for i in range(trainSize):
```

```
    LSTrainError += (leastSquares.predict(trainData.X[i])-trainData.Y[i]) ** 2
LSTrainError /= trainSize
print("LeastSquares train error:", LSTrainError)
print("LeastSquares.w:", leastSquares.w)

print("--------Training Lasso with data size of", trainSize, "and lambda of", lassoLambda, "--------")
lasso = LassoRegression(trainData.X, trainData.Y, lassoLambda)
lassoTrainError = 0
for i in range(trainSize):
    lassoTrainError += abs(lasso.predict(trainData.X[i])-trainData.Y[i])
lassoTrainError /= trainSize
print("Lasso train error:", lassoTrainError)
print("Lasso.w:", lasso.w)

features = []
for i in range(1, len(lasso.w)):
    if lasso.w[i] > 0:
        features.append(i)
print("Significant features:", features)
selectedX = []
selectedY = []
for i in range(trainSize):
    tmpX = [1]
    for j in range(1, 21):
        if j in features:
            tmpX.append(trainData.X[i][j])
        else:
            tmpX.append(0)
    selectedX.append(np.array(tmpX))
    selectedY.append(trainData.Y[i])
selectedX = np.array(selectedX)
selectedY = np.array(selectedY)

ridgeLambda = 4.440892098500626e-16
print("--------Training Ridge with data size of", trainSize, "and lambda of", ridgeLambda, "--------")
ridge = RidgeRegression(selectedX, selectedY, ridgeLambda)
```

```
ridgeTrainError = 0
for i in range(trainSize):
    ridgeTrainError += (ridge.predict(selectedX[i])-selectedY[i]) ** 2
ridgeTrainError /= trainSize
print("Ridge train error:", ridgeTrainError)
print("Ridge.w:", ridge.w)

trueWeights = [0.6 ** i for i in range(1, 11)]
trueWeights += [0 for _ in range(10)]
scalex = list(range(1, 21))
plt.plot(scalex, trueWeights, 'b', label="True Weights")
plt.plot(scalex, leastSquares.w[1:], 'g*', label="LeastSquares Training Weights")
plt.plot(scalex, ridge.w[1:], 'r*', label="Lasso-Ridge Training Weights")
plt.legend()
figName = "Lasso-RidgeRegression lassoLambda" + str(lassoLambda) + "ridgeLambda" + str(ridgeLambda) +\
    "trainSize" + str(trainSize) + ".png"
plt.savefig(figName)

testData = DataSet(testSize)
print("--------Testing LeastSquares with data size of", testSize, "--------")
LSTestError = 0
for i in range(testSize):
    LSTestError += (leastSquares.predict(testData.X[i])-testData.Y[i]) ** 2
LSTestError /= testSize
print("testError:", LSTestError)
print("--------Testing Lasso-Ridge with data size of", testSize, "--------")
ridgeTestError = 0
for i in range(testSize):
    ridgeTestError += (ridge.predict(testData.X[i])-testData.Y[i]) ** 2
ridgeTestError /= testSize
print("testError:", ridgeTestError)
```

SVM.py

```
import numpy as np
from sympy import *
from matplotlib import pyplot as plt
import sys


class XORData:
    def __init__(self):
        self.X = np.mat([[-1, -1], [-1, 1], [1, 1], [1, -1]])
        self.Y = np.array([-1, 1, -1, 1])


class DualSVMs:
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y
        self.tarLoss = 0.00001
        self.alpha, self.minStep = self.fit(self.X, self.Y)

    def fit(self, X, Y):
        print("--------Training DualSVMs--------")
        print("Training Data:")
        print("X:\n", X)
        print("Y:\n", Y)
        m = X.shape[0]
        alpha = [1 for _ in range(m - 1)]
        alphaTmp = 0
        epsilon = 0.01
        step = 1
        preLoss = 1
        curLoss = 2
        while abs(curLoss - preLoss) > self.tarLoss:
            # print("alpha:", alpha)
            preLoss = self.calcLoss(X, Y, step, alpha)
            for i in range(m - 1):
                curWeights = self.calcWeights(X, Y, i + 1, step, alpha)
```

```
            alpha[i] += epsilon * curWeights
        step += 1
        curLoss = self.calcLoss(X, Y, step, alpha)
    for i in range(m - 1):
        alphaTmp -= alpha[i] * Y[i + 1] * Y[0]
    alpha.append(alphaTmp)
    return alpha, step

def fitStepByEpsilon(self, epsilon):
    X = self.X
    Y = self.Y
    m = X.shape[0]
    alpha = [1 for _ in range(m - 1)]
    step = 1
    preLoss = 1
    curLoss = 2
    while abs(curLoss - preLoss) > self.tarLoss:
        # print("alpha:", alpha)
        preLoss = self.calcLoss(X, Y, step, alpha)
        for i in range(m - 1):
            curWeights = self.calcWeights(X, Y, i + 1, step, alpha)
            alpha[i] += epsilon * curWeights
        step += 1
        curLoss = self.calcLoss(X, Y, step, alpha)
        if step >= 1000:
            return step
    return step

def calcLoss(self, X, Y, step, alpha):
    m = X.shape[0]
    t = var("alpha:4")
    dim = [0 for _ in range(6)]
    for i in range(1, m):
        dim[0] -= t[i] * (Y[i] * Y[0])
        dim[1] += t[i]
        dim[2] += t[i] * Y[i] * self.calcKernel(X[i], X[0])
```

```
        for j in range(1, m):
            dim[3] += t[i] * Y[i] * self.calcKernel(X[i], X[j]) * Y[j] * t[j]
        dim[4] += log(t[i])
    dim[5] = log(dim[0])
    first = dim[0] + dim[1]
    second = (-1 / 2) * (
        pow(dim[0], 2) * self.calcKernel(X[0], X[0])
        + 2 * Y[0] * dim[0] * dim[2]
        + dim[3]
    )
    loss = first + second + 1 / (step ** 2) * (dim[4] + dim[5])
    return loss.subs({alpha1: alpha[0], alpha2: alpha[1], alpha3: alpha[2]})

def calcWeights(self, X, Y, index, step, alpha):
    m = X.shape[0]
    t = var("alpha:4")
    dim = [0 for _ in range(6)]
    for i in range(1, m):
        dim[0] -= t[i] * (Y[i] * Y[0])
        dim[1] += t[i]
        dim[2] += t[i] * Y[i] * self.calcKernel(X[i], X[0])
        for j in range(1, m):
            dim[3] += t[i] * Y[i] * self.calcKernel(X[i], X[j]) * Y[j] * t[j]
        dim[4] += log(t[i])
    dim[5] = log(dim[0])
    first = dim[0] + dim[1]
    second = (-1 / 2) * (
        pow(dim[0], 2) * self.calcKernel(X[0], X[0])
        + 2 * Y[0] * dim[0] * dim[2]
        + dim[3]
    )
    loss = first + second + (1 / step ** 2) * (dim[4] + dim[5])
    return diff(loss, t[index]).subs(
        {alpha1: alpha[0], alpha2: alpha[1], alpha3: alpha[2]}
    )
```

```
    def calcKernel(self, X, Y):
        return int(((1 + X * Y.T) ** 2)[0][0])

    def plotOverEpsilon(self, epsilonStart, epsilonStop, epsilonStep):
        steps = []
        epsilons = []
        for epsilon in np.arange(epsilonStart, epsilonStop, epsilonStep):
            epsilons.append(epsilon)
            steps.append(self.fitStepByEpsilon(epsilon))
        plt.plot(epsilons, steps)
        plt.title("Epsilon over Convergence Steps with accuracy of" + str(self.tarLoss))
        plt.xlabel("Epsilon")
        plt.ylabel("Convergence Steps")
        fileName = "SVM-EpsilonpOverSteps-" + "start" + str(epsilonStart*1000) +  "stop" + str(epsilonStop*1000) + "step" +
str(epsilonStep*1000) +".png"
        plt.savefig(fileName)


if __name__ == "__main__":
    xor = XORData()
    model = DualSVMs(xor.X, xor.Y)
    print("alpha:", model.alpha)
    print("convergence step:", model.minStep)
    # epsilonStart = 0.1022
    # epsilonStop = 0.1026
    # epsilonStep = 0.00001
    # model.plotOverEpsilon(epsilonStart, epsilonStop, epsilonStep)
```