1 CS 536 : Computing Solutions

2 Fatima AlSaadeh (fya7)

3 Linear Regression

[3]:
```
import numpy as np
import math
import pandas as pd
```

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.

[4]:
```
# generating the data based on the questions instructions
def generate_data(m):
    all_x = []
    all_y = []
    true_b = 0
    for i in range (m):
        x = np.random.normal(0, 1,21)
        sigma = np.sqrt(0.1)
        # adding at x0 1 to carry the bias value in the model calculation
        x[0]=1
        x[11] = x[1]+x[2]+np.random.normal(0, sigma)
        x[12] = x[3]+x[4]+np.random.normal(0, sigma)
        x[13] = x[4]+x[5]+np.random.normal(0, sigma)
        x[14] = (0.1 * x[7])+np.random.normal(0, sigma)
        x[15] = 2* x[2]-10+np.random.normal(0, sigma)
        y = 10+np.random.normal(0, sigma)
        true_b += y
        true_w = [0]
        for i in range(1,11):
            true_w.append(.6**i)
            y+= (.6**i)*x[i]
        for i in range(11,21):
            true_w.append(0)
        all_x.append(x)
        all_y.append(y)
    true_b =  true_b/m
    return np.array(all_x),np.array(all_y),np.array(true_w), np.array(true_b)
```

[5]:
```
# generating the columns labels
columns = []
for i in range(20):
    s = "X"+str(i+1)
    columns.append(s)
```

[6]:
```
# naive least squares regression model
def naive_regression(X,Y):
    Xt = X.transpose()
    XtX_mult = np.matmul(Xt, X)
    inv= np.linalg.inv(XtX_mult)
    res = np.matmul(inv, Xt)
    w = np.matmul(res, Y)
    return w
# naive least squares regression model predictions
def predict(x,w):
    y = np.matmul(x,w)
    return y
# calculate the model error
def true_error(y,pred_y):
    return sum(((y-pred_y)**2))/len(y)
```

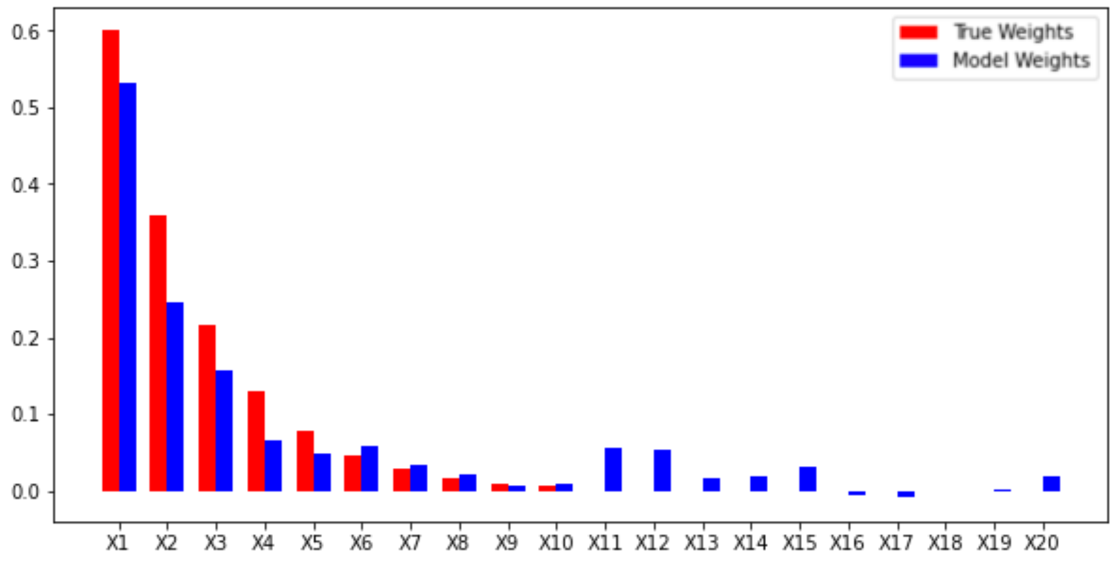
[7]:
```
#bar plot function
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (10,5)
def bar_plot(data1, data2, labels, legend, title):
    X = np.arange(len(labels))
    fig, ax = plt.subplots()
    ax.bar(X - (0.35/2), data1, color = 'r', width = 0.35, label='True'+legend)
    ax.bar(X + (0.35/2), data2, color = 'b', width = 0.35, label='Naive'+legend)
    ax.set_xticks(X)
    ax.set_xticklabels(labels)
    ax.legend()
    fig.suptitle(title)
    fig.show()
```

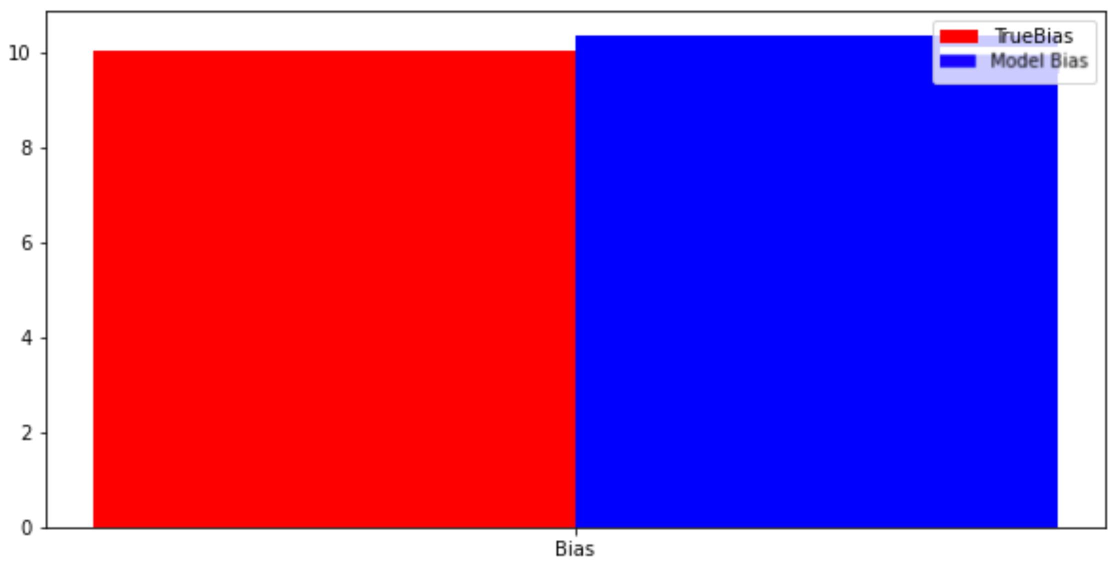
[29]:
```
# results function that will make call to the models based
# on the model param value model = 0,1,2 = naive, ridge, lasso respectively
# The function generate data and split 80% training and 20% testing
# l is lambda value for ridge or lasso
def get_results(model, m=1000,l = .1, plotit = True,num_iterations=100):
    size = int(m/.8)
    x,y,true_w,true_b= generate_data(size)
    x_tr = x[:int(.8*len(x)),]
    x_te = x[int(.8*len(x)):,]
    y_tr = y[:int(.8*len(y)),]
    y_te = y[int(.8*len(y)):,]
    if model == 0: w = naive_regression(x_tr,y_tr)
    elif model==1: w = ridge_regression(x_tr,y_tr,l)
    elif model==2: w = lasso_regression(x_tr,y_tr,l,num_iterations)
    if plotit:
        bar_plot(true_w[1:21], w[1:21], columns,'Weights',"")
        bar_plot(true_b, w[0], ["Bias"],'Bias',"")
    pred_y = predict(x_te,w)
    t_err = true_error(y_te,pred_y)
    if plotit:
        print("true error = " + str(t_err))
    return t_err,w
```

[9]:
```
# for model naive on m = 1000 generate training data of size
# 1000 and testing data of size 250 and plot the results
get_results(0,m=1000)
```

```
true error = 0.10993665201704796
```

[9]:
```
(0.10993665201704796,
 array([ 1.03519111e+01,  5.31811428e-01,  2.46435926e-01,  1.56641466e-01,
         6.70043963e-02,  4.84879774e-02,  5.79174936e-02,  3.39061986e-02,
         2.22905136e-02,  6.51393045e-03,  8.26475583e-03,  5.65809758e-02,
         5.37471523e-02,  1.61582661e-02,  1.86045682e-02,  3.24353742e-02,
        -4.64209725e-03, -9.25974667e-03, -8.12523776e-04,  9.75450483e-04,
         1.83568868e-02]))
```

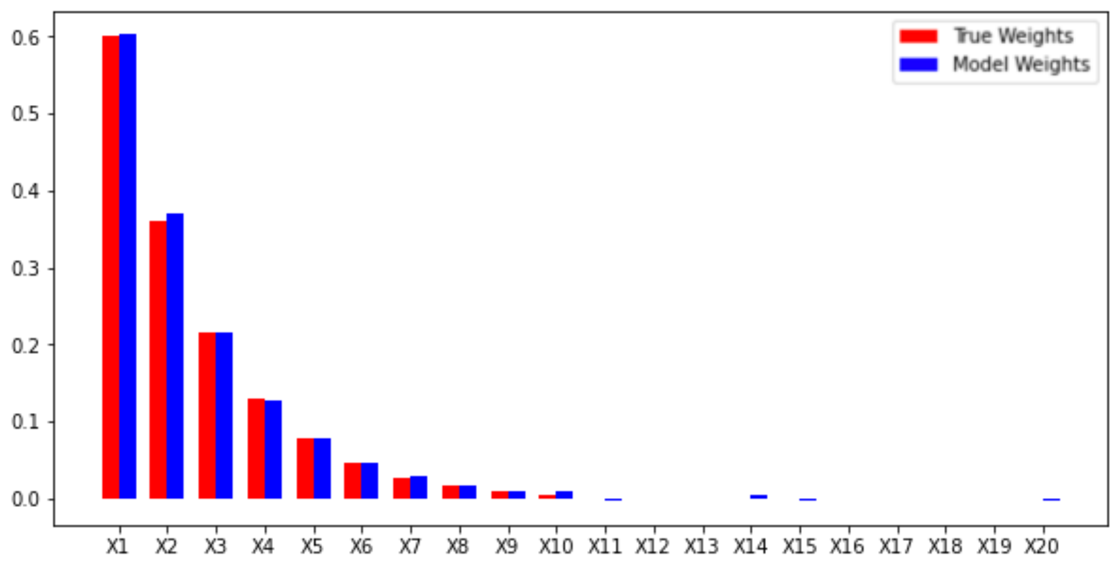


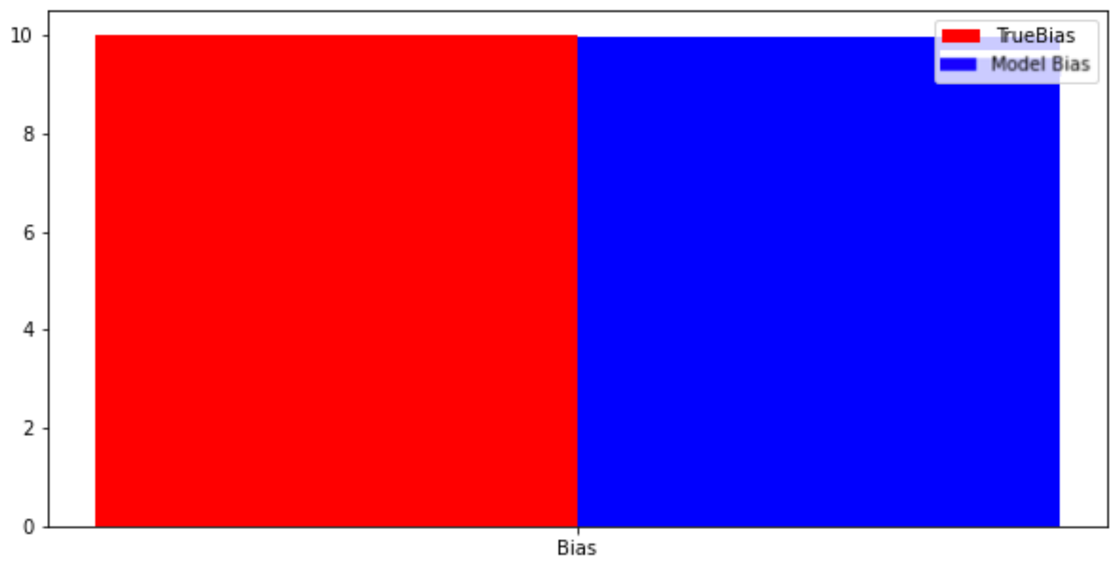


```
[10]: # for model naive on m = 30000 generate training data of size
      # 30000 and testing data of size 7500 and plot the results
      get_results(0,m=30000)
```

```
true error = 0.09812168934507037
```

```
[10]: (0.09812168934507037,
       array([ 9.97680706e+00,  6.02348129e-01,  3.70139349e-01,  2.16562619e-01,
               1.27316472e-01,  7.76283102e-02,  4.55254820e-02,  2.91109243e-02,
               1.82519414e-02,  1.05700105e-02,  9.13935258e-03, -3.58169570e-03,
               6.03529292e-04,  9.02982657e-04,  6.00521952e-03, -2.46756579e-03,
               7.19918568e-04, -8.54932630e-04,  6.41316254e-04,  5.12020749e-04,
              -2.44281077e-03]))
```





Analysis: The calculated weights and biases appear to be close to the true ones, On m=1000 (The function generate data and split 80% training =1000 data point and 20% testing = 250) the model concluded that X1 is the most significant features and the features X19 is the least significant feature and in most cases some features between X11-X20 were pruned (X18 is pruned in this case) and the true error was 0.10993665201704796. on m with 300000 training data points and 7500 testing data points the true error was 0.0981216893450703 the features X12,X13,X16,X17,X18,X19 were pruned and X11,X15,X20 were the least significant. Note: to calculate the true error (testing error) in all sections for this question I used MSE (mean squared error).

2) Write a program to take a data set of size m and a parameter , and solve for the ridge re-

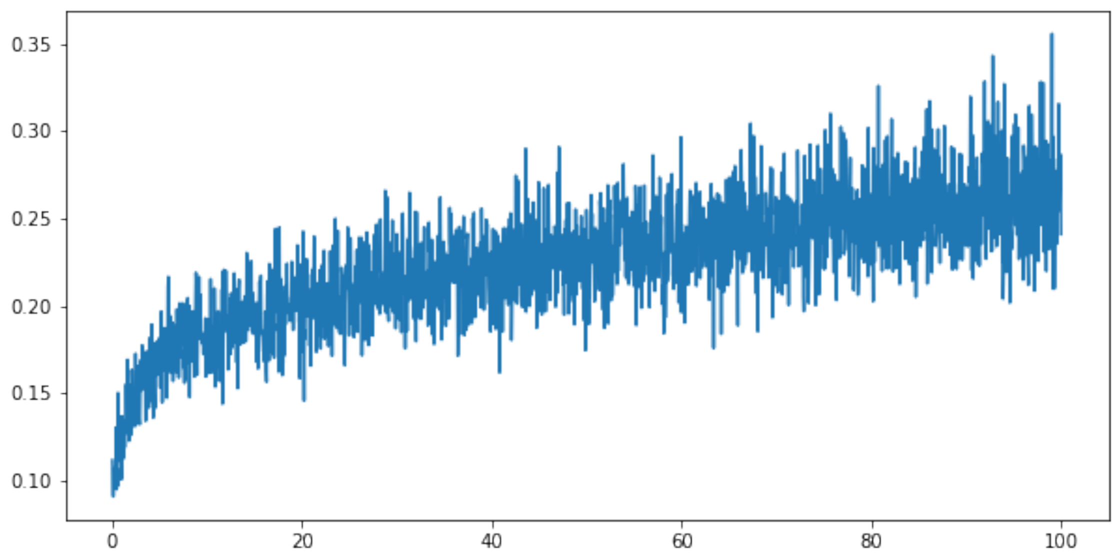
gression model for that data. Write another program to take the solved model and estimate the true error by evaluating that model on a large test data set. For data sets of size m = 1000, plot estimated true error of the ridge regression model as a function of . What is the optimal to minimize testing error? What are the weights and biases ridge regression gives at this , 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?

[11]:
```
# ridge regression model
def ridge_regression(X,Y, l):
    Xt = X.transpose()
    XtX_mult = np.matmul(Xt, X)
    Ident = l*np.identity(len(X[0]), dtype = float)
    addit = XtX_mult+ Ident
    inv= np.linalg.inv(addit)
    res = np.matmul(inv, Xt)
    w = np.matmul(res, Y)
    return w
```

[12]:
```
# on m = 10000 we are fitting the ridge_regression on the training data
# and calculating the true error on testing data
l = np.arange(.01, 100, .05)
true_err = []
for i in l:
    t_err, w = get_results(1,1000,i, False)
    true_err.append(t_err)
```

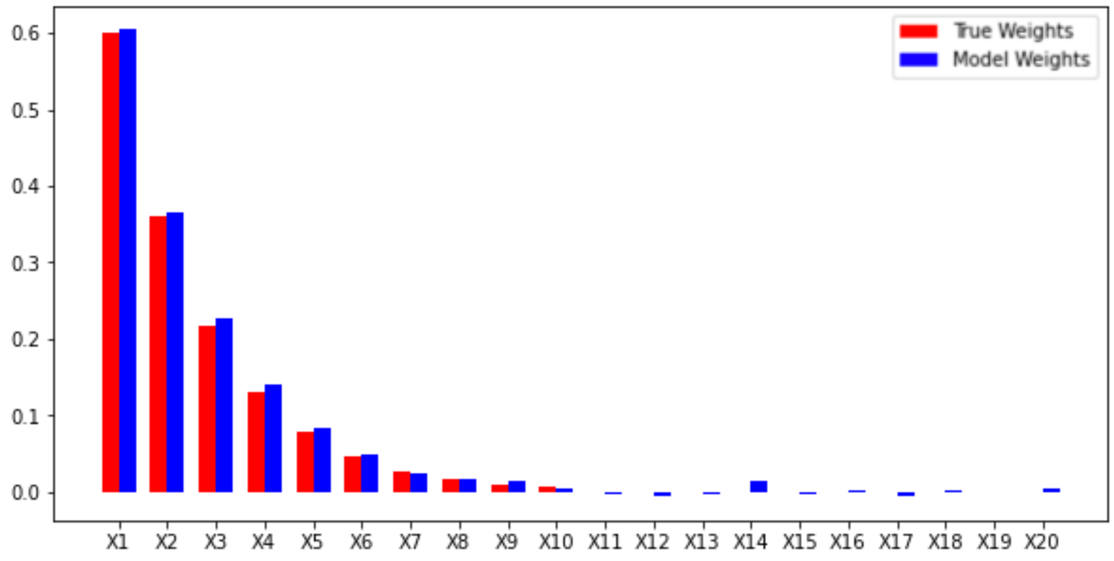
[13]:
```
# plo
plt.plot(l,true_err)
best_l =l[true_err.index(min(true_err))]
print(l[true_err.index(min(true_err))])
```

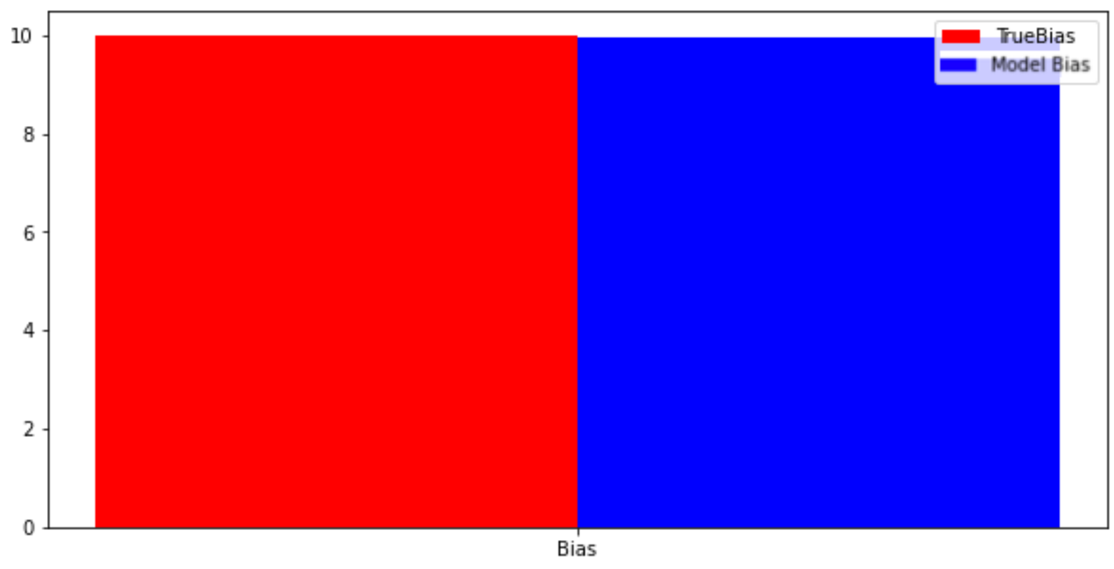
```
0.11
```



```
[14]: t_err,w = get_results(1,10000,best_l, True)
      true_err.append(t_err)
```

```
true error = 0.09971132725953837
```





[15]:
```
print("Weights:")
print(w[1:])
```

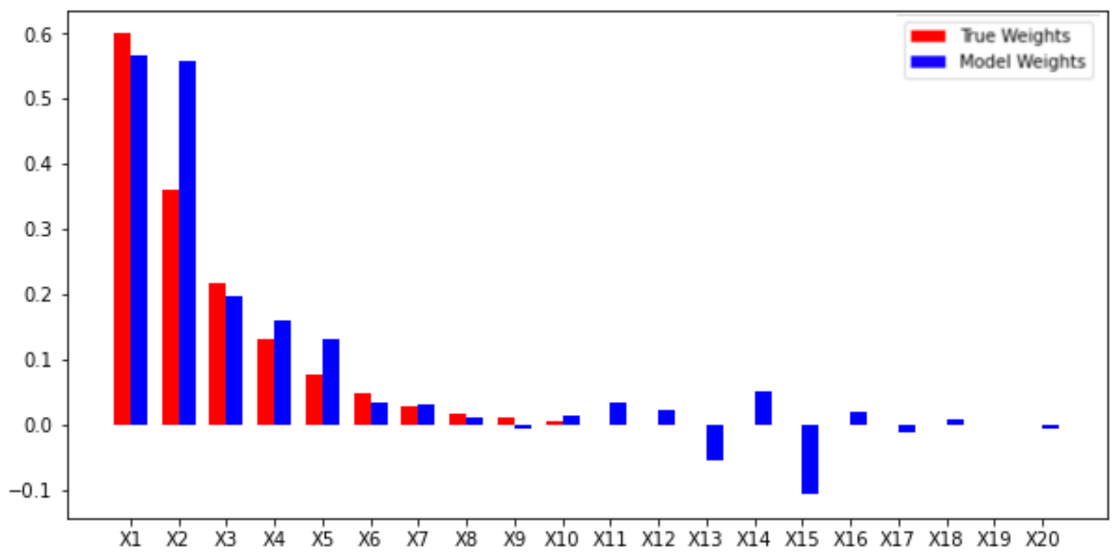
```
Weights:
[ 0.60413981  0.36601815  0.22758706  0.14136323  0.08301341  0.04869998
  0.02563887  0.01778709  0.01444125  0.00513201 -0.00321291 -0.00625562
 -0.00239028  0.01541516 -0.00372036  0.00343057 -0.00428563  0.00126597
  0.00093525  0.00464683]
```

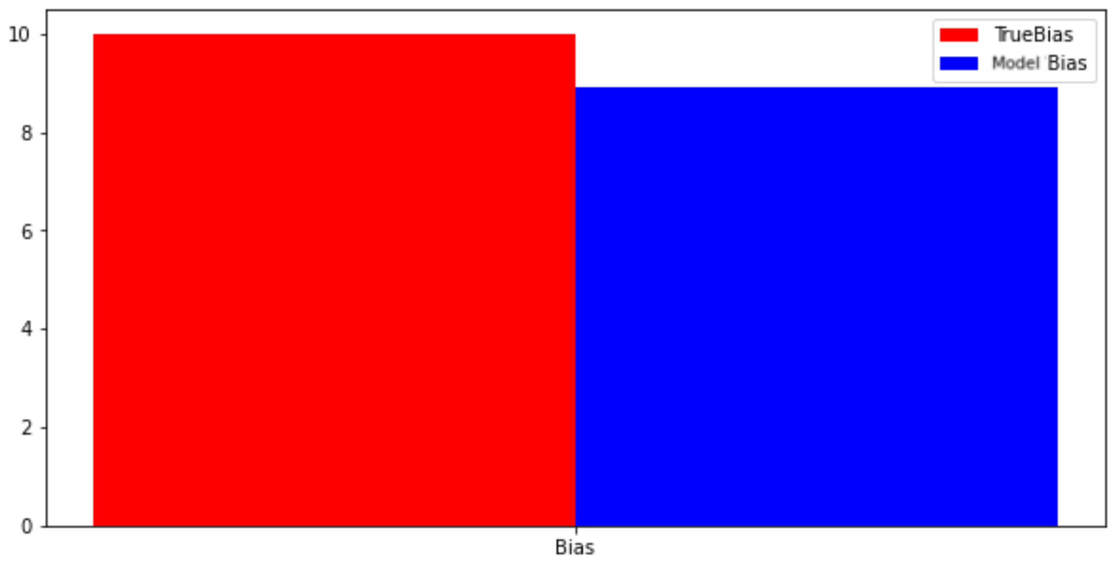
[16]:
```
print("Bias")
print(w[0])
```

```
Bias
9.956294968318847
```

[206]:
```
t_err,w = get_results(1,1000,best_l, True)
```

```
true error = 0.10084901734682271
```



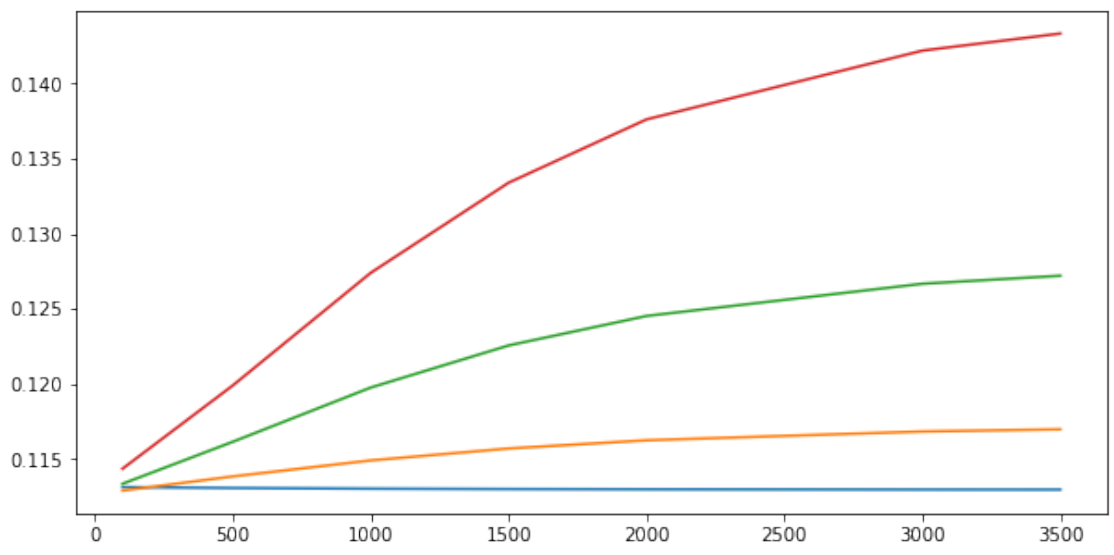


Analysis: on m of size 1000, and lambda in range [.01, 100] with .05 step the minimum true error was at lambda 0.11, trying this lambda on dataset of size 10,000 for training and 2500 for testing the true error was 0.09971132725953837 with weights and bias as shown in the above two sections, they are relatively close to the true weights with an error. the most significant feature is X1 and X11,X13,X15,X18 least significant where X19 was pruned. The naive least squares model on m=1000 the results were with true error of 0.109936652017047969, while the ridge regression for m = 1000 the true error of 0.10084901734682271, both of them were able to prune one feature, the results was close for the two models on m =1000.

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

[73]:
```
# lasso regression model takes x,y, lambda, and number of iterations
def lasso_regression(x,y, l, num_iter=100):
    #(xi.T (y-X.w) - l/2)/((xi.T).Xi)
    w = [0]*21
    all_w =[]
    for i in range(num_iter):
        old_weights = w
        for i in range(len(w)):
            temp1 = (np.matmul(x[:,i].T,(y - np.matmul(x,w))) - (l/2))/((np.
 ↪matmul(x[:,i].T, x[:,i])))
            temp2 = (np.matmul(x[:,i].T,(y - np.matmul(x,w))) + (l/2))/((np.
 ↪matmul(x[:,i].T, x[:,i])))
            if  w[i]> -1*temp1 :
                w[i] = w[i] + temp1
            elif w[i] < -1*temp2 :
                w[i] = w[i] + temp2
            else:
                w[i] = 0
    return w
```

[20]:
```
# find best num iterations
# try different number of iteratioms on different values of lambdas and pick the␣
 ↪number of iteration that
# always gave the minimum true error
size = int(1000/.8)
x,y,true_w,true_b= generate_data(size)
x_tr = x[:int(.8*len(x)),]
x_te = x[int(.8*len(x)):,]
y_tr = y[:int(.8*len(y)),]
y_te = y[int(.8*len(y)):,]
l = [0,5,10,30,40,50,90,100,500,1000,1500,2000,3000,3500]
j = np.arange(.01, 20, 5)
all_err = []
for d in j:
    all_err = []
    for i in l:
        w = lasso_regression(x_tr,y_tr,d,i)
        pred_y = predict(x_te,w)
        t_err = true_error(y_te,pred_y)
        all_err.append(t_err)
    plt.plot(l,all_err)
```

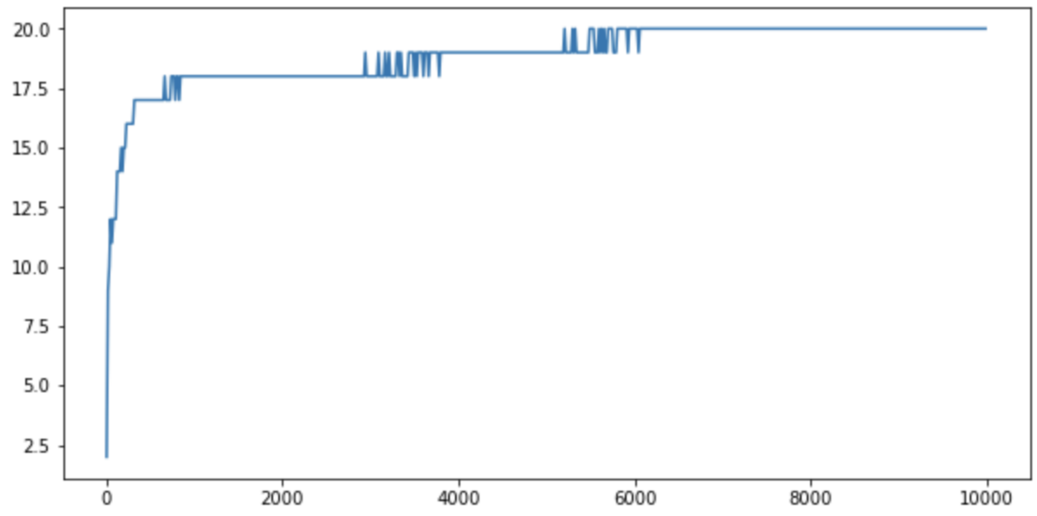


```
[26]: best_num_iteration =l[all_err.index(min(all_err))]
      print(best_num_iteration)
```

```
100
```

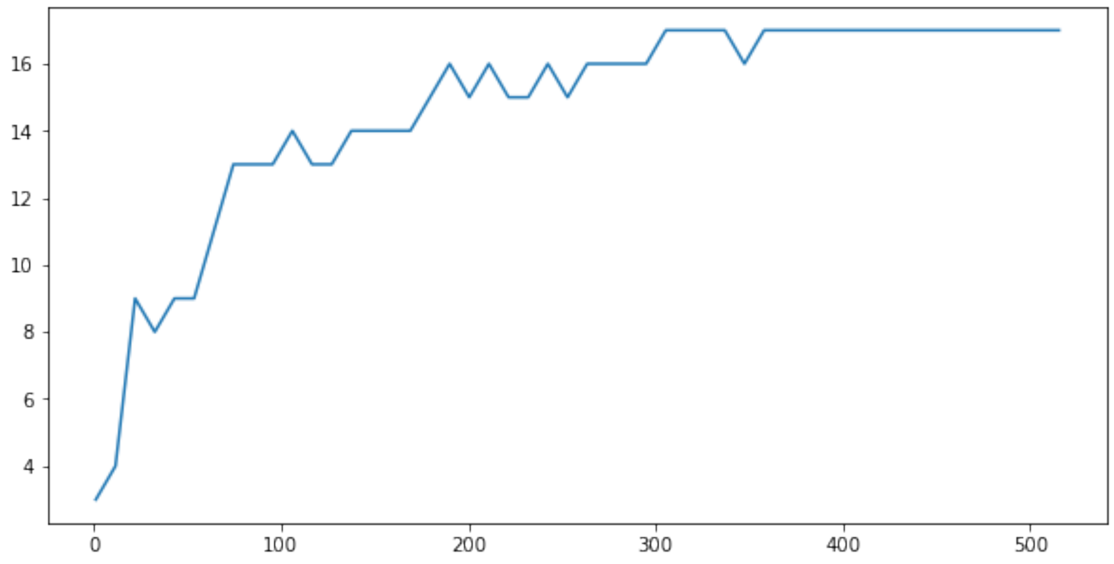
```
[32]: l = np.arange(1, 10000, 15)
      num_eliminated = []
      for i in range(len(l)):
          t_err, w = get_results(2, 1000,l[i], False,best_num_iteration)
          num_eliminated.append(w.count(0))
      plt.plot(l,num_eliminated)
```

```
[32]: [<matplotlib.lines.Line2D at 0x7f62bc2c30a0>]
```



[38]:
```
l = np.arange(1, 520, 10.5)
num_eliminated = []
for i in range(len(l)):
    t_err, w = get_results(2, 1000,l[i], False,best_num_iteration)
    num_eliminated.append(w.count(0))
plt.plot(l,num_eliminated)
```

[38]:
```
[<matplotlib.lines.Line2D at 0x7f62bc192a60>]
```

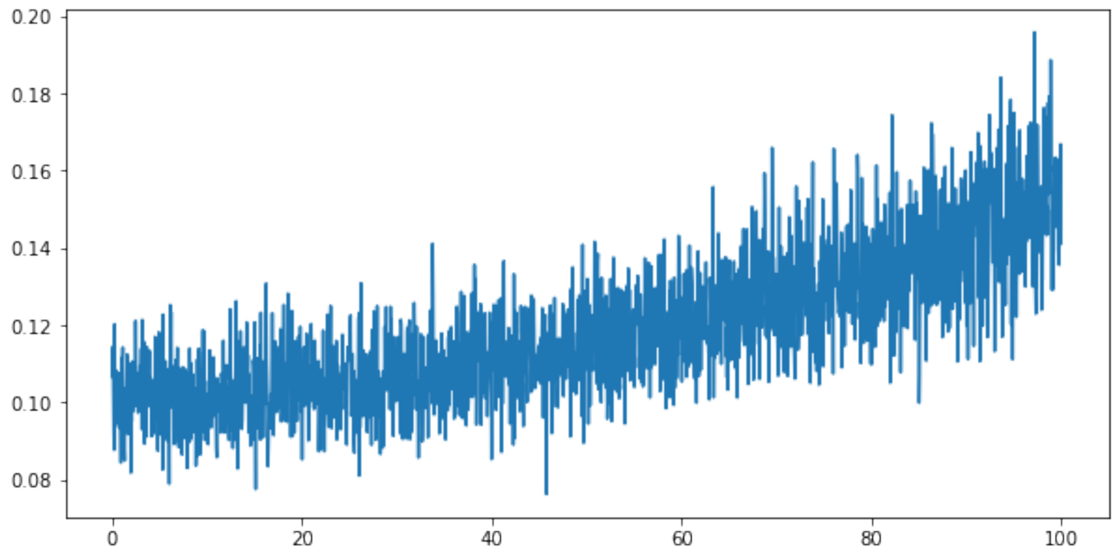


Analysis: I first did grid search to find the best number of iterations to use by looping over different number of iterations and in each iteration we used different number of lambdas then I picked the number of iterations with minimum true error, then running the model on these parameters The two plots above shows that as lambda increases, features are effectively eliminated from the model until all weights are set to zero in the first plots it shows how the number of eliminated features are approaching to 20 on large lambda value. The second plot is to show closely how the elimination process is working on smaller range of lambda values

4) For data sets of size m = 1000, plot estimated true error of the lasso regression model as a function of . What is the optimal to minimize testing error? What are the weights and biases lasso regression gives at this , 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 regression model compare tot he naive least squares model?

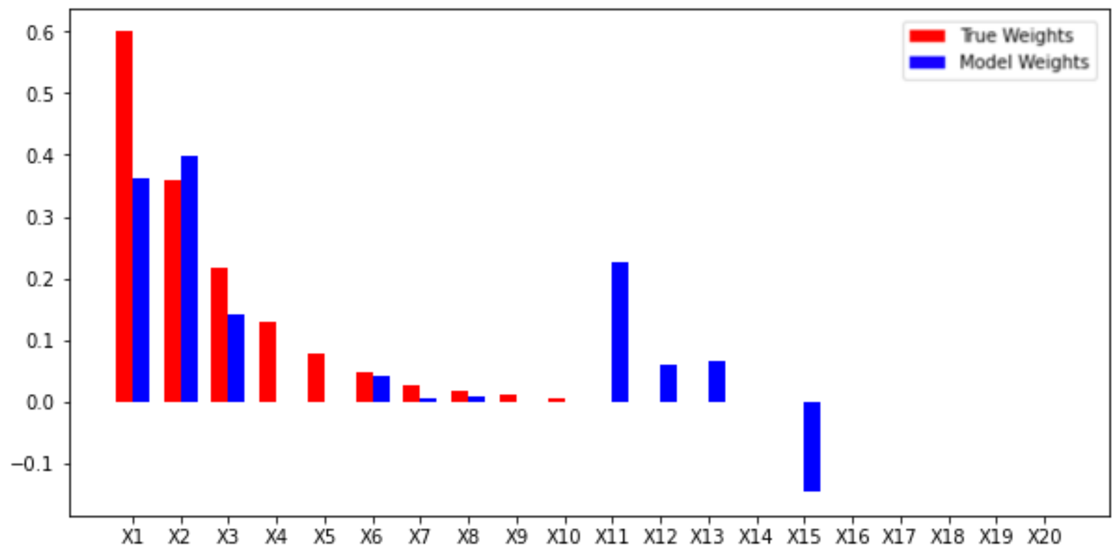
[12]:
```
# from the plot above the best lambda range is between (0,30) after that␣
 ↪relevant features
# might get pruned
l = np.arange(.01, 100, .05)
err = []
for i in l:
    true_err, w = get_results(2, 1000,l = i, plotit = False)
    err.append(true_err)
plt.plot(l,err)
best_l = l[err.index(min(err))]
print("best lambda value= " + str(best_l))
```

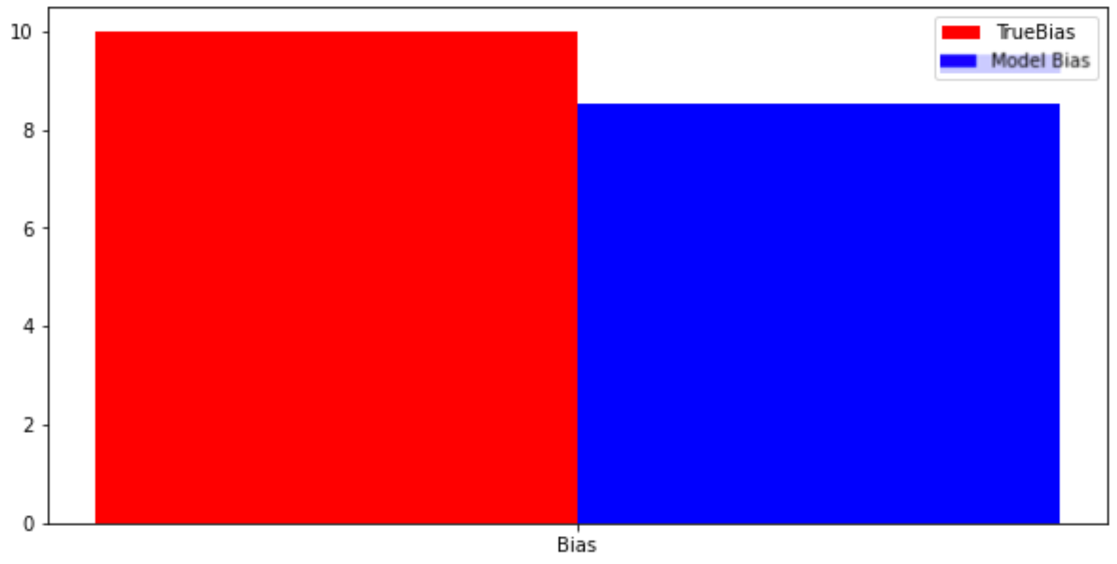
```
best lambda value= 45.76
```



[64]:
```
t, w =get_results(2, 1000,l = best_l, plotit = True)
```

```
true error = 0.10830274283289912
```





[65]:
```
print("Weights")
print(w[1:])
```

```
Weights
[0.3621979421508774, 0.3989980555365777, 0.14103636916948434, 0, 0,
0.04078030537218125, 0.00628058429324458, 0.008863048894971918, 0, 0,
0.22699538772687033, 0.0605940235622239, 0.06448201772721444, 0,
-0.14711547498543018, 0, 0, 0, 0, 0]
```

[66]:
```python
print("Bias")
print(w[0])
```

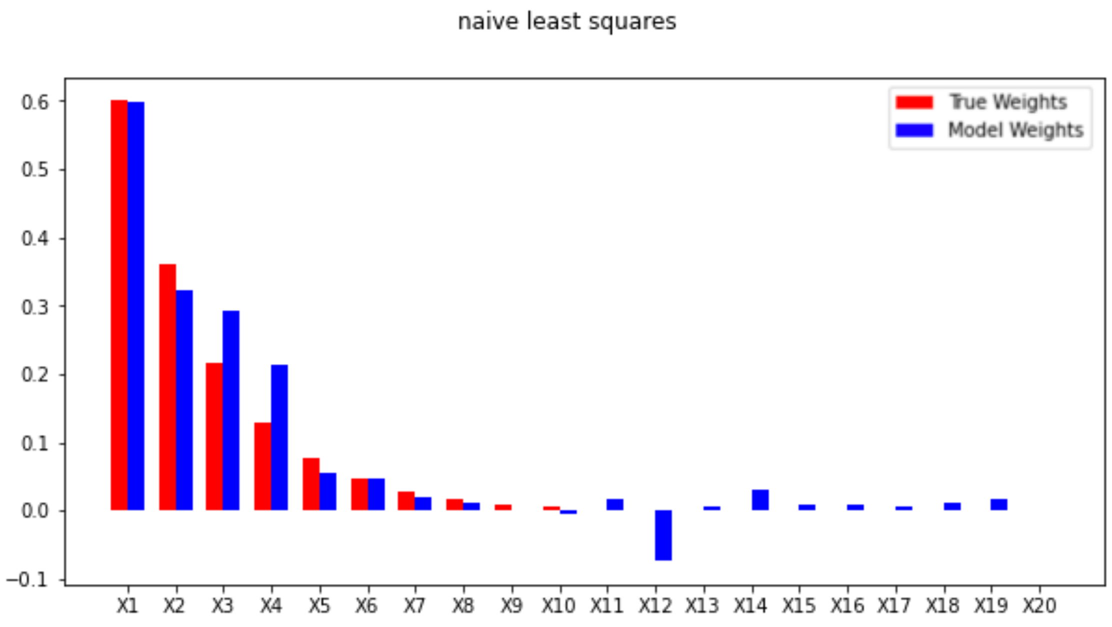
```
Bias
8.507686709910713
```

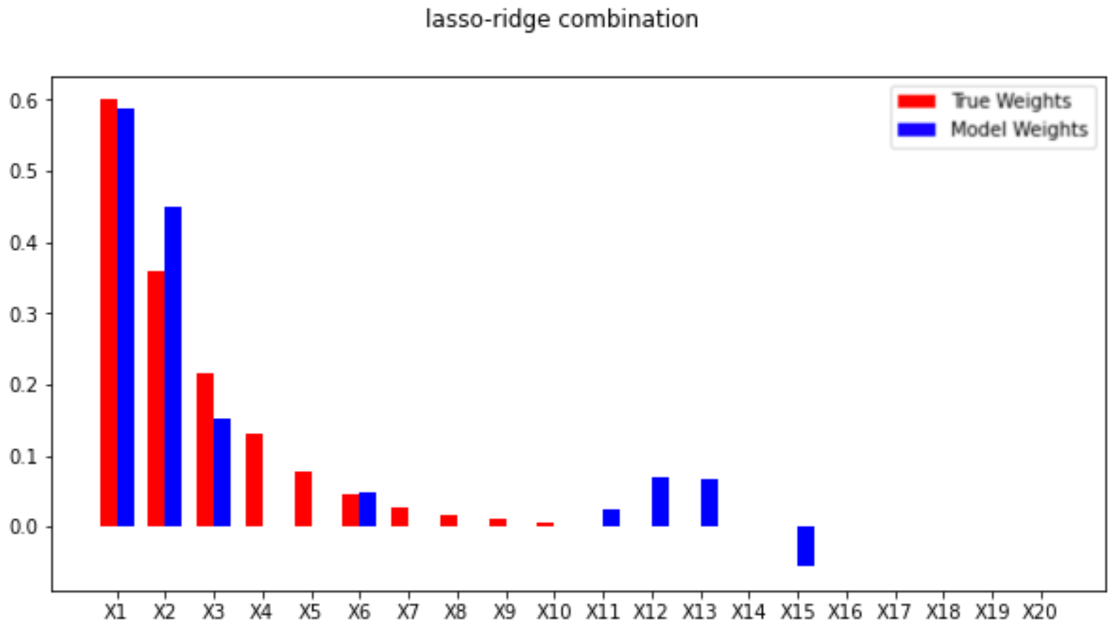
Analysis: For data sets of size m = 1000 the optimal lambda to minimize testing error was 45.76 for a range between [.01, 100] with .05 and error 0.10830274283289912 and weights and biases at this lambda shown in the two sections above, they appear to be less than the true weights and bias with the most significant features from X1 and X2. the X4,X5,X9,X10,X14,X17,X18,X19,X20 are pruned. From the previous results it appears that the lasso regression is performing better than the naive least squares model, but it pruned two significant true weights (X4,X5).

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?

[74]:
```python
# combined lasso ridge model
# run lasso regression with the optimal regularization constant found in␣
↪previous sections
# identify the set of relevant features
# run ridge regression to fit a model to only those features
def get_results_lasso_ridge(m=1000,l = .1, plotit = True):
    size = int(m/.8)
    # generate training and testing data
    x,y,true_w,true_b= generate_data(size)
    x_tr = x[:int(.8*len(x)),]
    x_te = x[int(.8*len(x)):,]
    y_tr = y[:int(.8*len(y)),]
    y_te = y[int(.8*len(y)):,]
    # run naive regression and get the true error
    w_naive = naive_regression(x_tr,y_tr)
    pred_y_naive = predict(x_te,w_naive)
    t_err_naive = true_error(y_te,pred_y_naive)
    # plot the wights and bias for naive
    if plotit:
        bar_plot(true_w[1:21], w_naive[1:21], columns,'Weights',"naive least␣
↪squares")
    # run lasso regression and get the true error
    w = lasso_regression(x_tr,y_tr,l)
    pred_y = predict(x_te,w)
    t_err = true_error(y_te,pred_y)
    before = t_err
    # set the irrelevant features weights to 0
    size = len(w[1:])
    for i in range(size):
        if w[i] ==0:
            x_tr[:, i] = 0
    idx = np.argwhere(np.all(x_tr[..., :] == 0, axis=0))
    for i in idx:
        x_te[:,i] = 0
    lambdas = np.arange(0.05, 10, 0.5)
    errors = []
    ws = []
    #  search to find the best ridge constant
    for i in lambdas:
        w = ridge_regression(x_tr,y_tr,i)
        pred_y = predict(x_te,w)
        t_err = true_error(y_te,pred_y)
        ws.append(w)
        errors.append(t_err)
    best_l_r = lambdas[errors.index(min(errors))]
    best_w_r = ws[errors.index(min(errors))]
    best_err = min(errors)
    if plotit:
        bar_plot(true_w[1:21], best_w_r[1:21], columns,'Weights',"lasso-ridge␣
 ↪combination")
    return best_l_r, l,best_w_r, w, best_err, t_err_naive
last_all_naive = []
lass_all_combination = []
for i in range(1000):
    plotit = False
    if i == 9: plotit = True
    best_l_r, l,best_w_r, w, best_err,t_err_naive =␣
 ↪get_results_lasso_ridge(1000,best_l, plotit)
    lass_all_combination.append(best_err)
    last_all_naive.append(t_err_naive)
print("Naive Error: " + str(np.mean(last_all_naive)))
print("lasso-ridge combination Error: " + str(np.mean(lass_all_combination)))
```

```
Naive Error: 0.10262680786263052
lasso-ridge combination Error: 0.10337009361254357
```





Analysis: to determine a good ridge regression regularization constant to use here, I ran the ridge regression on the relevant features on different lambda values in the range [0.05, 10] with 0.5 step and the best lambda was .05. The resulting lasso-ridge appeared to be worse than the naive least squares model with error. The lasso-ridge combination model considered as significant features X1, X2 and pruned X4,X5,X7,X8,X9,X10,X14,X16,X17,X18,X19,X20. The results shows that for m=1000 and same training and testing data the Naive Error= 0.10262680786263052 and lasso-ridge combination Error = 0.10337009361254357

4 SVMs

1) Implement a barrier-method dual SVM solver. How can you (easily!) generate an initial feasible 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 t? Be sure to return all i including 1 in the final answer.

[233]:
```
import numpy as np
import matplotlib.pyplot as plt

# find the values of y predicted by the dual svm using barrier method
# and classify using the sign function
# return the classifications
def classify(x,y, w,b,alpha):
    k = kernel(x)
    predictions = 0
    for i in range(x.shape[0]):
        predictions+=(alpha[i]*y[i]*k[i])
    predictions+=b
    y_pred = np.sign(predictions)
    return y_pred

# find the error of the model
# error = sum(y!=y_predicted)/number of predictions
def svm_true_error(x,y, w,b,alpha):
    y_pred = classify(x,y, w,b,alpha)
    return 1- np.mean(y_pred==y),y_pred

# ploynomial kernel of degree 2
def kernel(x):
    return (1 + x.dot(x.T)) ** 2

# generate alpha within the constraints range mentioned in the question
def generate_alpha(x,y):
    alpha = np.random.random(x.shape[0])
    f = -1.0*np.sum(alpha[1:]*y[1:]*y[0])
    # if the values of alpha outside the constraints region regenerate
    # until the constraints satisfied
    while f < 0 and any(i < 0 for i in alpha[1:]):
        alpha = np.random.random(x.shape[0])
        f = -1.0*np.sum(alpha[1:]*y[1:]*y[0])
    alpha[0] = f
    return alpha

# The modified objective function as the question mentioned
def objective_function(alpha,y,k):
        m = len(y)
        f = -1*np.sum(alpha[1:]*y[1:]*y[0])
        b = np.sum(alpha[1:m])
        c = f**2 * k[1][1]
        dsum =0
        for i in range(1,m):
            dsum+=alpha[i]*y[i]*k[i][1]
        d = f * y[1]* dsum
        e = 0
        for i in range(1,m):
            for j in range(1,m):
                e+=(alpha[i]*y[i]*k[i][j]*y[j]*alpha[j])
        obj_function = (f+b) - 0.5*(c+ (2*d)+e)
        return obj_function
#dual svm optimized using the barrier method with epsilon array of values
def svm_dual_barrier(x,y,iterations, is_squareds=False, plt_it=True):
    b = 0
    w = 0
    err =0
    object_funs = []
    eps = np.arange(0, 100, .01)
    eps = eps[:iterations+1]
    eps=eps[::-1]
    k = kernel(x)
    alpha = generate_alpha(x,y)
    obj_fun = objective_function(alpha,y,k)
    for iteration in range(1,iterations):
        f = -1*np.sum(alpha[1:]*y[1:]*y[0])
        if is_squareds:
            picked_eps = 1/((iteration)**2)
        else:
            picked_eps = 1/((iteration))
        maximizer = obj_fun + picked_eps*(np.sum(np.log(alpha[1:]))+ np.log(f))
        alpha = alpha + maximizer
        f = -1*np.sum(alpha[1:]*y[1:]*y[0])
        while f<0 or any(i < 0.0 for i in alpha[1:]):
            alpha = generate_alpha(x,y)
            f = -1*np.sum(alpha[1:]*y[1:]*y[0])
        alpha[0] = f
        obj_fun = objective_function(alpha,y,k)
        object_funs.append(obj_fun)
    if plt_it:
        plt.plot(object_funs)
        plt.title("Objective Function Values")
    for i in range(x.shape[0]):
        w+=alpha[i]*y[i]*x[i]
    b = y[0] - w.dot(x[0])
    return alpha, w, b
```

```

```

Analysis: To initialize the values we generate randomly alpha values, check if it's within the constraint region if yes we start working with it, if no we move to generate new random variables and repeat.for any update step we did a check on our calculated values if it step outside the constraint region we recalculate our alpha values until it's within the constraints region. I tried two approches to chose epsilon at time t, the first approach is epsilon = 1/iteration while iterations [1,iterations_value] The second approach is epsilon = 1/(iterationˆ2) while iterations [1,iterations_value], then compared the performance for the two approaches by calculating the accuracy when doing the svm prediction over 100 iterations. when epsilon = 1/iteration it performed better by giving higher accuracy and score of 1 show in the upcoming sections (the third section below compares the accuracy of the two approaches when we repeat the prediction 10 times each time over 2500 iterations)

Question 2 and 3 are solved in the same section

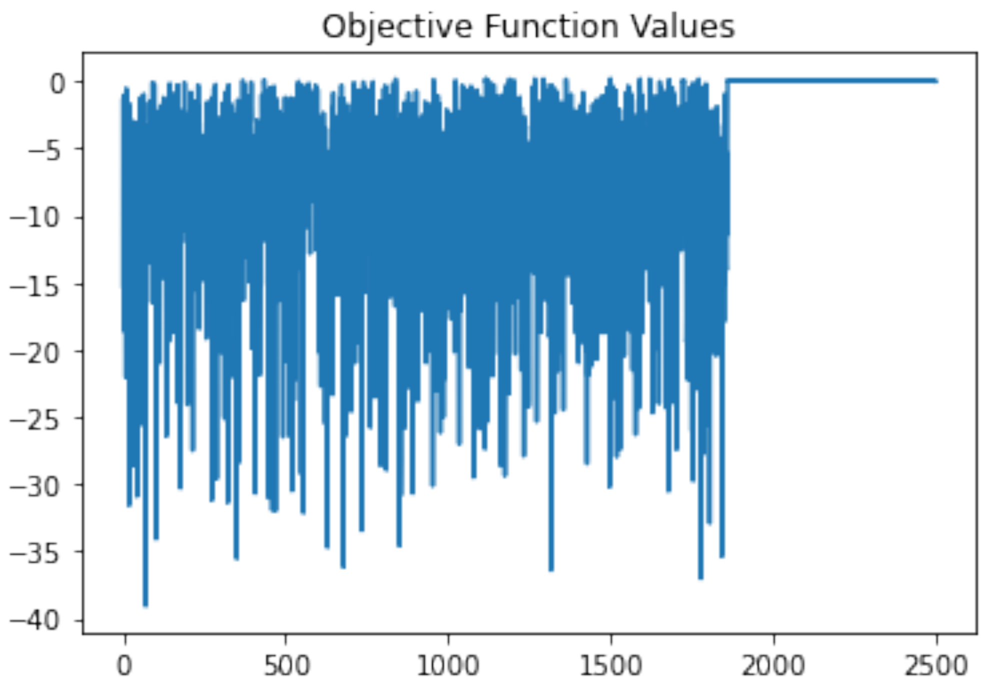
2) Use your SVM solver to compute the dual SVM solution for the XOR data using the kernel function K(x, y). Solve the dual SVM by hand to check your work.

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

4.0.1 first section:

[247]:
```
# XOR data using the kernel function and with chosing epsilon in order
# in the range of [0,5] with step .01
x = np.array([[-1,-1],[-1,1],[1,-1],[1,1]])
y = np.array([-1,1,1,-1])
y.reshape(-1,1)
alpha, w, b= svm_dual_barrier(x,y,2500,is_squareds = False)
print("alpha values returned:")
print(alpha)
err, pred = svm_true_error(x,y, w,b,alpha)
print("the error of the classification :")
print(err)
print("the score of the classification :")
print(1-err)
print("the predictions made : ")
print(pred)
```

```
alpha values returned:
[0.18617442 0.10661936 0.12032244 0.04076738]
the error of the classification :
0.0
the score of the classification :
1.0
the predictions made :
[-1.  1.  1. -1.]
```

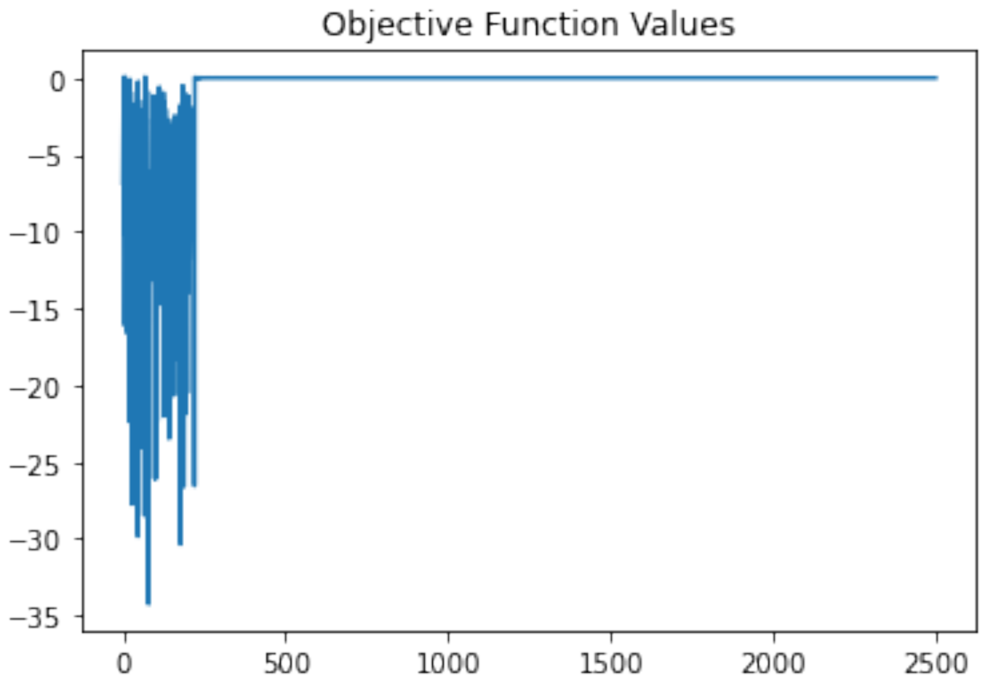


4.0.2 second section:

```
[250]: # solving XOR data using the kernel function using dual svm and the barrier␣
        ↪method
       #and with chosing epsilon that will maximize the objective function at time t
       x = np.array([[-1,-1],[-1,1],[1,-1],[1,1]])
       y = np.array([-1,1,1,-1])
       y.reshape(-1,1)
       alpha, w, b = svm_dual_barrier(x,y,2500,is_squareds = True)
       print("alpha values returned:")
       print(alpha)
       err, pred = svm_true_error(x,y, w,b,alpha)
       print("the error of the classification :")
       print(err)
       print("the score of the classification :")
       print(1-err)
       print("the predictions made : ")
       print(pred)
```

```
alpha values returned:
[0.15961589 0.11867385 0.07547059 0.03452856]
the error of the classification :
0.25
the score of the classification :
```

```
0.75
the predictions made :
[-1.  1. -1. -1.]
```



4.0.3 third section

```
[287]: x = np.array([[-1,-1],[-1,1],[1,-1],[1,1]])
y = np.array([-1,1,1,-1])
y.reshape(-1,1)
total_sq = 0
total = 0
for i in range(10):
    alpha, w, b = svm_dual_barrier(x,y,2500,is_squareds = True,plt_it=False)
    err, pred = svm_true_error(x,y, w,b,alpha)
    if 1-err == 1:
        total_sq+=1
    alpha, w, b = svm_dual_barrier(x,y,2500,is_squareds = False,plt_it=False)
    err, pred = svm_true_error(x,y, w,b,alpha)
    if 1-err == 1:
        total+=1
print("Accuracy 1/(t**2):",total_sq/10)
print("Accuracy 1/t:",total/10)
```

```
Accuracy 1/(t**2): 0.2
Accuracy 1/t: 0.6
```

Analysis: On epsilon = 1/iteration and with 2501 iterations in the first section the alpha values returned are [0.18617442 0.10661936 0.12032244 0.04076738] the model was able to predict the values with 0 error and y predicted is [-1., 1., 1., -1.] with chosing the epsilon = 1/(iterationˆ2) with 2501 iterations the alpha values returned [0.15961589 0.11867385 0.07547059 0.03452856] the model was able to predict the values with 0.25 error and y predicted is [-1. 1. -1. -1.] as shown in the second section above. The results of solving the svm by hand was close to the results of the first approach, the work by hand is below:

```
[ ]: Next Page
```

[ ]: By Hand

4) The XOR data represented as :
X1: $(-1,+1),+1$
X2: $(-1,-1),-1$
X3: $(+1,-1),+1$
X4: $(+1,+1),-1$.

we construct x as : $\begin{bmatrix} -1 & -1 \\ -1 & +1 \\ +1 & -1 \\ +1 & +1 \end{bmatrix}$ and y as: $\begin{bmatrix} -1 \\ +1 \\ +1 \\ -1 \end{bmatrix}$ having the kernel function

$$K(x_i, x_j) = (1 + x_i * x_j^T)^2$$

and the dual svm :

$$\max_{\underline{\alpha}} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i y^i K(\underline{x}^i, \underline{x}^j) y^j \alpha_j,$$

$$\text{(s.t.)} \quad \sum_{i=1}^{m} \alpha_i y^i = 0$$

$$\forall i : \alpha_i \geq 0,$$

$$\text{classifier}(\underline{x}) = \text{sign}\left(\sum_i \alpha_i y^i K(\underline{x}^i, \underline{x}) + b \right).$$

$$K1,1 = (1 + \begin{bmatrix} -1 & -1 \end{bmatrix} * \begin{bmatrix} -1 \\ -1 \end{bmatrix})^2 = 9$$

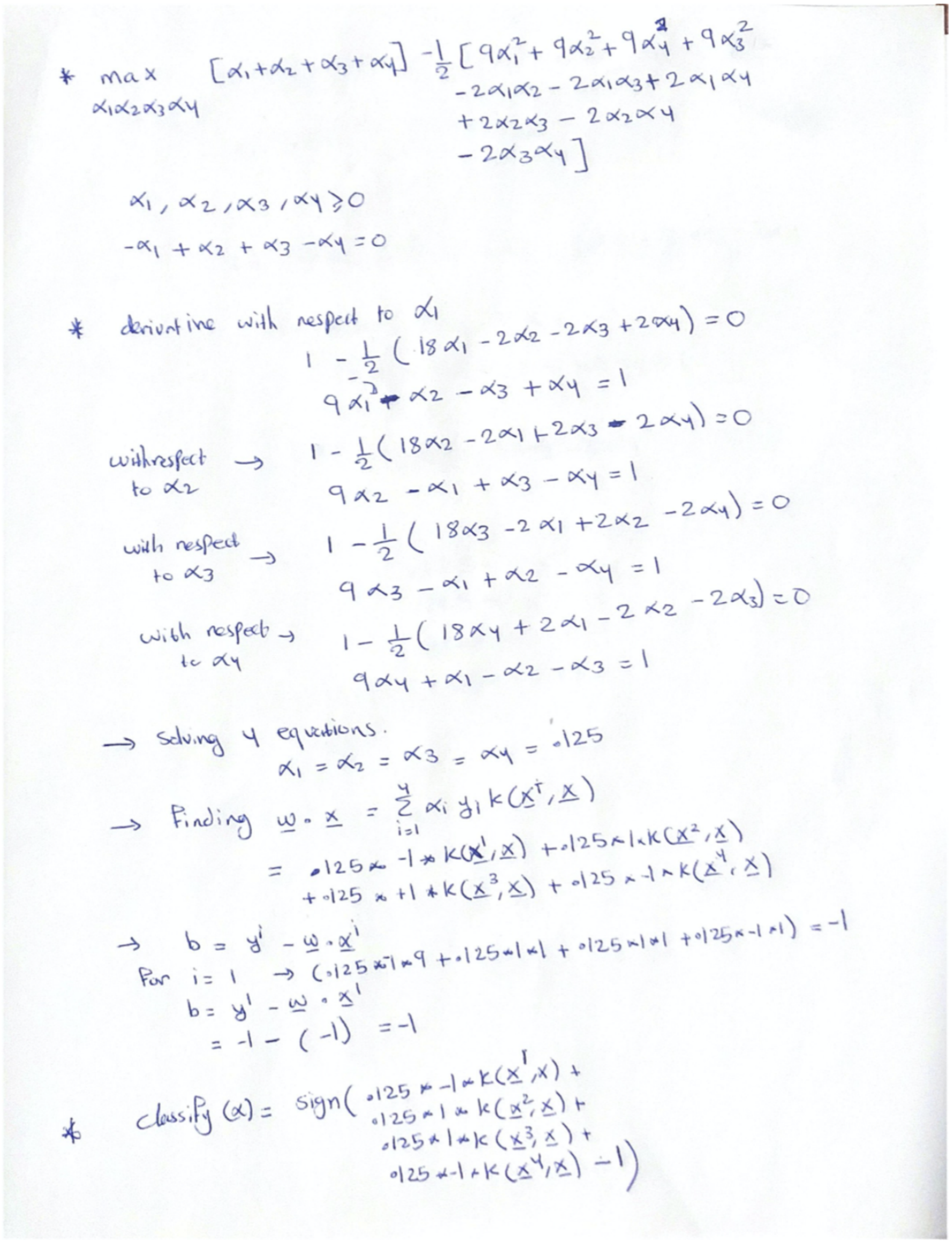
$$K1,2 = (1 + \begin{bmatrix} -1 & -1 \end{bmatrix} * \begin{bmatrix} -1 \\ +1 \end{bmatrix})^2 = 1$$

$$K1,3 = (1 + \begin{bmatrix} -1 & -1 \end{bmatrix} * \begin{bmatrix} +1 \\ -1 \end{bmatrix})^2 = 1$$

$$K1,4 = (1 + \begin{bmatrix} -1 & -1 \end{bmatrix} * \begin{bmatrix} +1 \\ +1 \end{bmatrix})^2 = 1$$

$$K1,1 * y1 * y1 = 9 * +1 * +1 = +9$$

$$K=\begin{bmatrix}9 & 1 & 1 & 1\\ 1 & 9 & 1 & 1\\ 1 & 1 & 9 & 1\\ 1 & 1 & 1 & 9\end{bmatrix} \quad K*yi*yj=\begin{bmatrix}+9 & -1 & -1 & +1\\ -1 & +9 & +1 & -1\\ -1 & +1 & +9 & -1\\ +1 & -1 & -1 & +9\end{bmatrix}$$



* $\max_{\alpha_1\alpha_2\alpha_3\alpha_4} [\alpha_1+\alpha_2+\alpha_3+\alpha_4] - \frac{1}{2}[9\alpha_1^2 + 9\alpha_2^2 + 9\alpha_4^2 + 9\alpha_3^2$
$-2\alpha_1\alpha_2 - 2\alpha_1\alpha_3 + 2\alpha_1\alpha_4$
$+2\alpha_2\alpha_3 - 2\alpha_2\alpha_4$
$-2\alpha_3\alpha_4]$

$\alpha_1, \alpha_2, \alpha_3, \alpha_4 \geq 0$

$-\alpha_1 + \alpha_2 + \alpha_3 - \alpha_4 = 0$

* derivative with respect to α_1 $1 - \frac{1}{2}(18\alpha_1 - 2\alpha_2 - 2\alpha_3 + 2\alpha_4) = 0$

$9\alpha_1 - \alpha_2 - \alpha_3 + \alpha_4 = 1$

with respect to α_2 $\rightarrow$ $1 - \frac{1}{2}(18\alpha_2 - 2\alpha_1 + 2\alpha_3 - 2\alpha_4) = 0$

$9\alpha_2 - \alpha_1 + \alpha_3 - \alpha_4 = 1$

with respect to α_3 $\rightarrow$ $1 - \frac{1}{2}(18\alpha_3 - 2\alpha_1 + 2\alpha_2 - 2\alpha_4) = 0$

$9\alpha_3 - \alpha_1 + \alpha_2 - \alpha_4 = 1$

with respect to α_4 $\rightarrow$ $1 - \frac{1}{2}(18\alpha_4 + 2\alpha_1 - 2\alpha_2 - 2\alpha_3) = 0$

$9\alpha_4 + \alpha_1 - \alpha_2 - \alpha_3 = 1$

$\rightarrow$ Solving 4 equations.
$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = .125$

$\rightarrow$ Finding $\underline{w} \cdot \underline{x} = \sum_{i=1}^{4} \alpha_i y_i k(\underline{x}^i, \underline{x})$
$= .125 * -1 * k(\underline{x}^1, \underline{x}) + .125 * 1 * k(\underline{x}^2, \underline{x}) + .125 * -1 * k(\underline{x}^4, \underline{x})$
$+ .125 * +1 * k(\underline{x}^3, \underline{x})$

$\rightarrow$ $b = y^i - \underline{w} \cdot \underline{x}^i$
For $i = 1$ $\rightarrow (.125 * -1 * 9 + .125 * 1 * 1 + .125 * 1 * 1 + .125 * -1 * 1) = -1$
$b = y^1 - \underline{w} \cdot \underline{x}^1$
$= -1 - (-1) = -1$

* classify $(x) =$ sign$(.125 * -1 * k(\underline{x}^1, x) +$
$.125 * 1 * k(\underline{x}^2, \underline{x}) +$
$.125 * 1 * k(\underline{x}^3, \underline{x}) +$
$.125 * -1 * k(\underline{x}^4, \underline{x}) - 1)$