Linear Regression

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.

The comparison graph and the calculation result are at the end of the answer. The overall trend of the difference between true and predicted weights and bias is similar. For X11-16, the true weight should be 0, there are still some weight in the training dataset. The most significant feature is X1, and least significant feature in this graph is X16, but during the time when I run the model several time, the least significant feature can also be X12, X13, X17, X18, X19… I don't think it is able to prune anything. The true error is 0.0940 (at the end of the answer).

```
import numpy as np
import pandas as pd
import seaborn as sns
```

```
def generate_x(m):
    x1_10 = np.random.normal(0,1,(m,11))
    x11 = np.asarray([x1_10[i][1] + x1_10[i][2] + np.random.normal(0,np.sqrt(0.1)) for i in range(m)]).reshape(-1,1)
    x12 = np.asarray([x1_10[i][3] + x1_10[i][6] + np.random.normal(0,np.sqrt(0.1)) for i in range(m)]).reshape(-1,1)
    x13 = np.asarray([x1_10[i][5] + x1_10[i][7] + np.random.normal(0,np.sqrt(0.1)) for i in range(m)]).reshape(-1,1)
    x14 = np.asarray([0.1*x1_10[i][7] + np.random.normal(0,np.sqrt(0.1)) for i in range(m)]).reshape(-1,1)
    x15 = np.asarray([2*x1_10[i][2] - 10 + np.random.normal(0,np.sqrt(0.1)) for i in range(m)]).reshape(-1,1)
    x16_20 = np.random.normal(0, 1, (m, 5))
    return np.concatenate((x1_10, x11, x12, x13,x14,x15,x16_20),1)
def generate_y(x, m):
    y = []
    for i in range(m):
        yvalue = 10
        for j in range(1, 11):
            yvalue = yvalue + ((0.6)**j)*x[i][j]
        yvalue += np.random.normal(0, np.sqrt(0.1))
        y.append(yvalue)
    return np.asarray(y)
def generate_dataset(m):
    x = generate_x(m)
    y = generate_y(x,m).reshape(m,1)
    dataset = pd.DataFrame(np.append(x, y, 1), columns=["X" + str(i) for i in range(21)]+['Y'])
    dataset['X0'] = 1
    return dataset
```

```
my_dataset = generate_dataset(1000)
my_dataset
```

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16	X17	X18
0	1	-0.293429	-1.330103	-0.889855	-0.765400	-0.716090	0.173252	-0.454633	0.421021	1.982509	-0.772769	-1.538833	-0.566281	-0.644171	0.125378	-12.570553	-0.963178	-2.023012	0.508198
1	1	-0.121993	2.017380	0.243652	0.351938	-0.362628	-0.546303	1.619385	-0.233606	-0.228036	0.835890	1.927235	-0.886450	1.408115	0.030113	-6.072722	-1.614086	-0.293879	-0.161044
2	1	1.065010	-2.072970	0.384060	0.311389	-1.524842	0.853919	-0.445736	-0.687275	-0.163766	0.842565	-0.640522	1.207413	-1.769250	0.416463	-14.189273	0.372150	-0.049385	-0.435955
3	1	-1.101008	-0.158912	3.443621	1.381127	-1.617270	1.205451	1.165205	-0.409871	-0.296569	1.517601	-1.047669	4.917646	-0.362483	0.413419	-10.699989	-0.974494	0.750298	1.291267
4	1	-0.314309	1.659284	-0.308988	-0.255199	-1.258616	-0.673033	-0.281478	1.092556	1.434613	1.665551	1.091998	-1.138471	-1.249284	0.274641	-6.576240	0.294184	-1.196703	1.410690
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
995	1	-0.003639	-0.796097	0.140745	0.764042	1.412627	0.085378	-0.370955	0.458949	0.308199	-0.923484	-0.959436	0.220086	1.401935	-0.308849	-11.478772	2.020664	-0.430650	1.639644
996	1	-0.268258	0.891519	0.335976	0.309578	0.432452	-0.526724	1.005247	-0.743668	-0.455817	-0.381893	1.297587	0.073542	1.467465	0.263108	-8.099310	-1.010114	1.137012	0.372465
997	1	0.876383	1.473384	-1.320148	0.090301	-1.305654	1.460729	0.956350	0.165804	1.893421	1.250099	2.304419	0.036562	0.004931	0.094618	-6.302386	0.190249	-1.631577	-0.640871
998	1	-0.457969	1.220347	-0.133718	-1.058599	-0.964940	-0.460591	-0.493911	0.753334	0.321669	-0.392294	0.671562	-0.845975	-1.551231	-0.262001	-7.700791	0.185614	0.050037	-0.490663
999	1	-1.382234	1.372084	-0.330885	1.618197	-0.991546	-0.724578	-0.280915	0.466291	0.916831	-0.179375	-0.001235	-0.717855	-0.259372	0.240776	-6.904329	-0.919651	-1.117563	1.019635

1000 rows × 22 columns

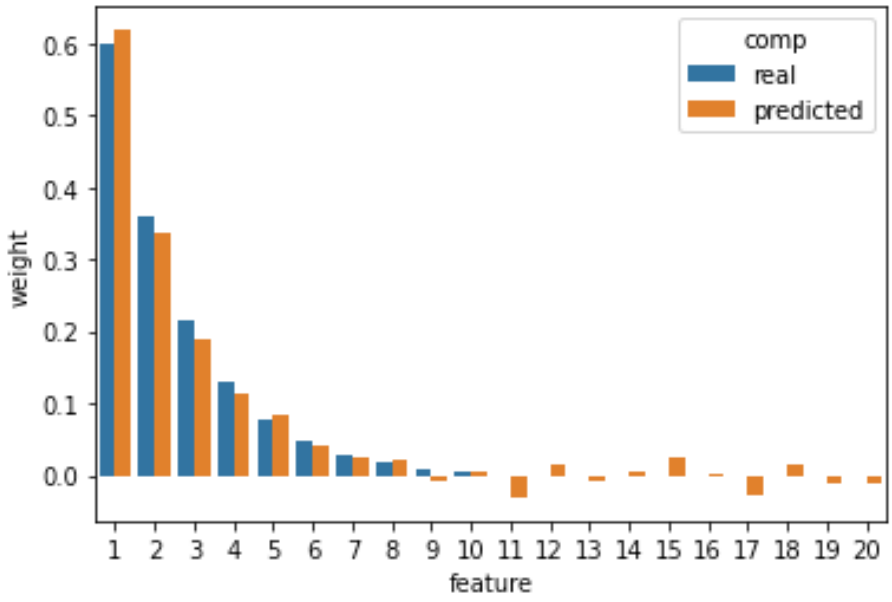
```
class Regression():

    def naive_regression(self, x, y):
        z, Xs = X.shape
        self.w = np.zeros(shape=(Xs,1))
        self.w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), y)
        return self.w
    def predict(self, x, w):
        return np.dot(x, w)
    def error(self, x, y, w):
        h_x = self.predict(x, w)
        error = 0
        for i in range(len(y)):
            error += (y[i] - h_x[i])**2
        error /= len(y)
        return error
```

```
    def compare_plot(real, predicted, X):
        data_real = pd.DataFrame(pd.Series(real), columns=['weight'])
        data_real['comp'] = pd.Series(['real']*len(real))
        data_real['feature'] = pd.Series(list(range(X)))
        data_real = data_real.iloc[1:]

        data_pred = pd.DataFrame(pd.Series(predicted), columns=['weight'])
        data_pred['comp'] = pd.Series(['predicted']*len(predicted))
        data_pred['feature'] = pd.Series(list(range(X)))
        data_pred = data_pred.iloc[1:]

        data = pd.concat([data_real, data_pred])

        sns.barplot(x="feature", y="weight", hue="comp", data=data)
```

```
def true_error(w):
    true_error = 0
    for i in (range(10)):
        data = generate_dataset(5000)
        x = np.asarray(data.iloc[:,:-1])
        y = np.asarray(data.iloc[:,-1:])
        true_error += lin_reg.error(x, y, w)
    return float(true_error/10)
```

```
lin_reg = Regression()
X = np.asarray(my_dataset.iloc[:,:-1])
y = np.asarray(my_dataset.iloc[:,-1:])
w_real = [10]
for i in range(1, 21):
    if i <= 10:
        w_real.append((0.6)**i)
    else:
        w_real.append(0)
w_pred = lin_reg.naive_regression(X, y)
features = X.shape[1]
compare_plot(w_real, w_pred.flatten(), features)
print("error: ", float(lin_reg.error(X, y, w_pred)))
print("bias: ", w_pred[0])
true_err = true_error(w_pred)
print("true error: ", true_err)
```

```
error:  0.10550087780297747
bias:  [10.25052643]
true error:  0.09405497458670264
```

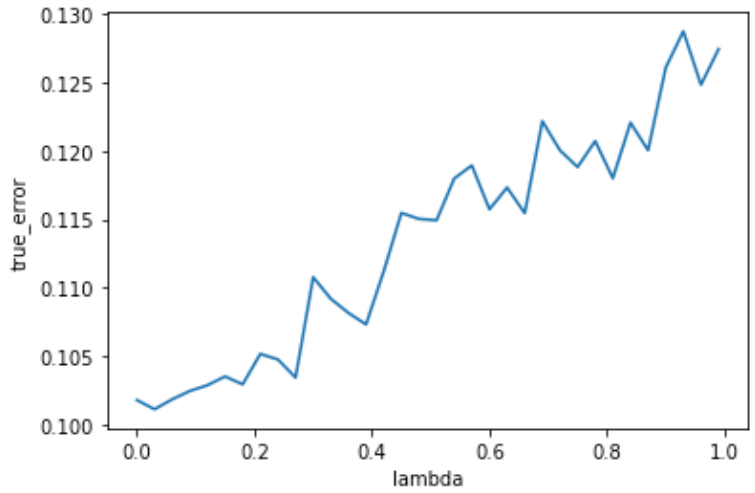


2) Write a program to take a data set of size m and a parameter λ, and solve for the ridge regression model for that data. Write another program to take the solved model and estimate the true error by evaluating that model on a large test data set. For data sets of size m = 1000, **plot estimated true 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?**

The plot is at the end of the answer. The optimal lambda to minimize testing error is about 0.03. Using lambda = 0.03, the weights and bias are similar to the real ones. In my graph, the most significant feature is X1, the least significant feature is X17. I don't think it is able to prune anything. Comparing to the naive least squares model, the testing error decreases only a little, but it is relatively more accurate.

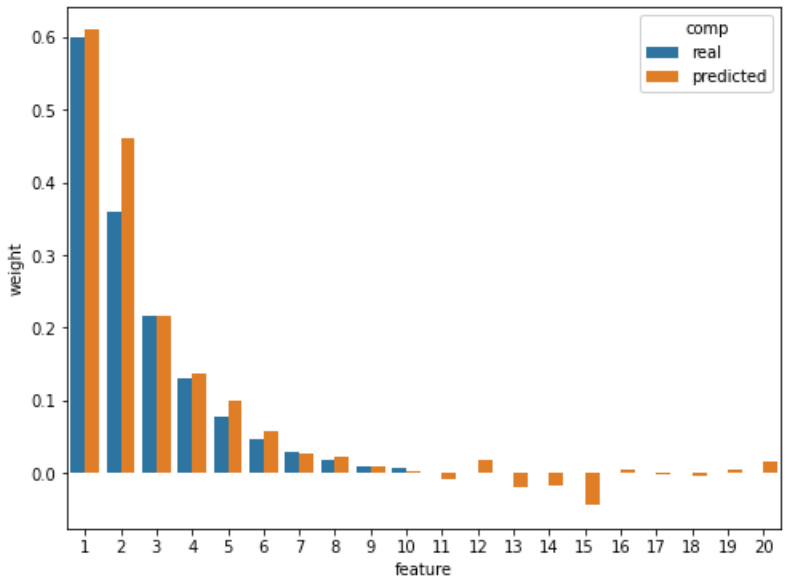
```
    def ridge_regression(self, x, y, lmbda):
        z, Xs = X.shape
        self.w = np.zeros(shape=(Xs,1))
        self.w = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X) + lmbda*np.identity(Xs)), X.T), y)
        return self.w
```

```
def lambda_increase(m):
    lmbda = np.arange(0, 1, 0.03)
    true_error = []
    for i in lmbda:
        error = 0
        data = generate_data(m)
        X = np.asarray(data.iloc[:,:-1])
        y = np.asarray(data.iloc[:,-1:])
        lin_reg = Regression()
        w = lin_reg.ridge_regression(X, y, i)
        true_error.append(error(w))

    plt.plot(lmbda, true_error)
    plt.xlabel("lambda")
    plt.ylabel("true_error")
    plt.show()

lambda_increase(1000)
```




```
Error:  0.09955034685375735
```

```
w_lambda = lin_reg.ridge_regression(X, y, 0.03)
compare_plot(w_real, w_lambda.flatten(), features)
```



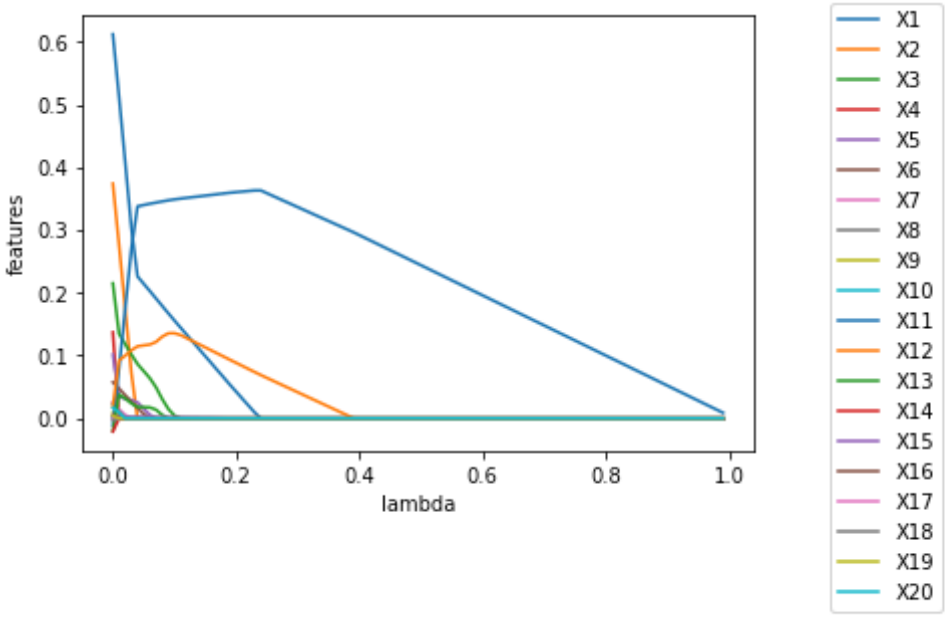
3)Write a program to take a data set of size m and a parameter λ, and solve for the Lasso regression model for that data. For a data set of size m = 1000, **show that as λ increases, features are effectively eliminated from the model until all weights are set to zero.**

From the graph, we can see that as lambda increases, the weight of each data is decreasing to 0. So as lambda increases more, all features will be effectively eliminated.

```
    def thresh(self, support, lmbda):
        if support > 0.0 and lmbda < abs(support):
            return (support - lmbda)
        elif support < 0.0 and lmbda < abs(support):
            return (support + lmbda)
        else:
            return 0.0
    def lasso_regression(self, X, y, lmbda, m):
        z, features = X.shape
        self.w = np.zeros(shape=(features,1))
        self.w[0] = np.sum(y - np.dot(X[:, 1:], self.w[1:]))/z
        for i in range(m):
            for j in range(1, features):
                copy_w = self.w.copy()
                copy_w[j] = 0.0
                residue = y - np.dot(X, copy_w)
                a1 = np.dot(X[:, j], residue)
                a2 = lmbda*z
                self.w[j] = self.thresh(a1, a2)/(X[:, j]**2).sum()
        return self.w
```

```
lmbda = np.arange(0, 1.0, 0.01)
weights = []
lin_reg = Regression()
for i in lmbda:
    weight = lin_reg.lasso_regression(X, y, i, 1000)
    weights.append(weight.flatten())
weights_lasso = np.stack(weights).T
weights_lasso[1:].shape
```

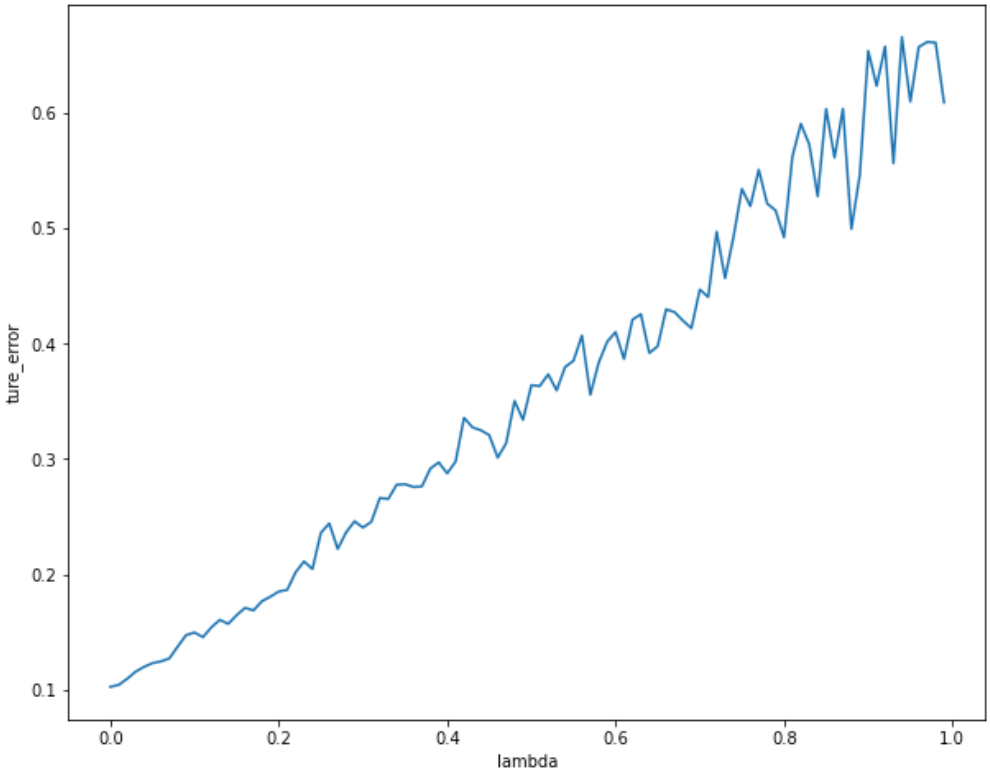
```
n = weights_lasso[1:].shape
for i in range(n):
    plt.plot(lmbda, weights_lasso[1:][i], label = my_dataset.columns[1:][i])
plt.xlabel('lambda')
plt.ylabel('features')
plt.legend(bbox_to_anchor=(1.1, 1.05))
plt.show()
```



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 significant features - was it able to prune anything? How does the optimal regression model compare to the naive least squares model?**

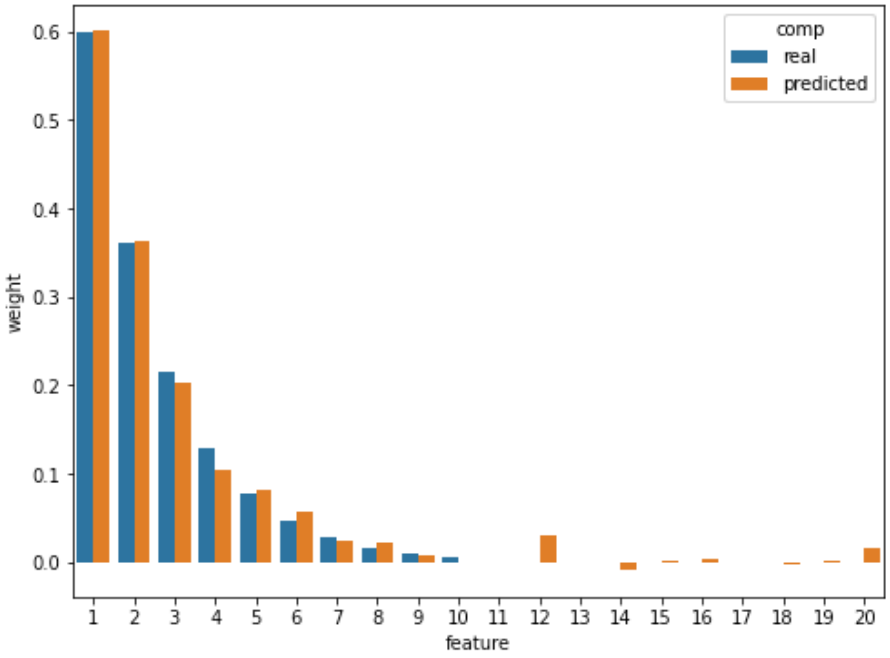
The plot is at the end of the answer. Since the trend of the true error graph is increasing between 0-0.1, the optimal lambda should be 0 to 0.001. The weights and biases are more similar to the true ones comparing to other to regression methods. The most significant feature is X1, and the least significant features are X11, X13, X17. The features after X10 except X12 and X20 all have a very low weight. It is able to prune some of the technically irrelevant variables. Comparing to the naive least squares model, the true error decreases a little, and the weight of each features are more similar to the true weight.

```
def lambda_lasso_increase(m):
    lmbda = np.arange(0, 1, 0.01)
    true_error = []
    for i in lmbda:
        data = generate_dataset(m)
        X = np.asarray(data.iloc[:,:-1])
        y = np.asarray(data.iloc[:,-1:])
        lin_reg = Regression()
        w = lin_reg.lasso_regression(X, y, i, 100)
        true_error.append(true_error(w))
    plt.plot(lmbda, true_error)
    plt.xlabel("lambda")
    plt.ylabel("ture_error")
    plt.show()
lambda_lasso_increase(1000)
```




```
optimal_weight = lin_reg.lasso_regression(X, y, 0.001, 1000)
compare_plot(w_real, optimal_weight.flatten(), features)
print("error: ", float(lin_reg.error(X, y, optimal_weight)))
```

```
Error:  0.0995878880287535
```

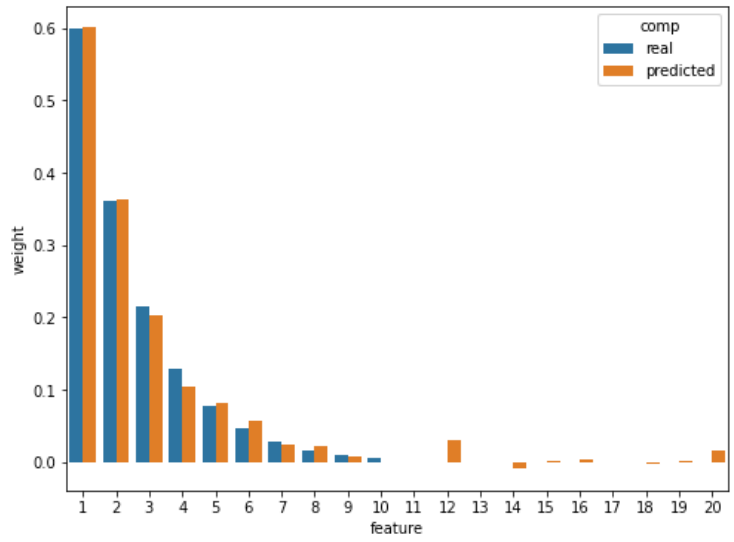


5) **Identify the set of relevant features; then run ridge regression to fit a model to only those features. How can you determine a good ridge regression regularization constant to use here? How does the resulting lasso-ridge combination model compare to the naive least squares model? What features does it conclude are significant or relatively insignificant? How do the testing errors of these two models compare?**

From question 4, we know that the irrelevant features are X10, X11, X13, X16, X17, X18. Other features are relevant features. The ridge regression regularization constant does not affect result very much, lambda = 0.01 is relatively good to use here. Comparing to the naive least square model, this eliminates a lot of irrelevant features. X1 is concluded as the most significant, features after X6 are relatively insignificant. The testing errors of these two models are similar.

```
data = generate_dataset(1000)
linreg = Regresssion()
X = np.asarray(data.iloc[:,:-1])
y = np.asarray(data.iloc[:,-1:])
compare_plt(w_real, optimal_weight.flatten(), features)
print("Error: ", float(lin_reg.error(X, y, optimal_weight)))
```

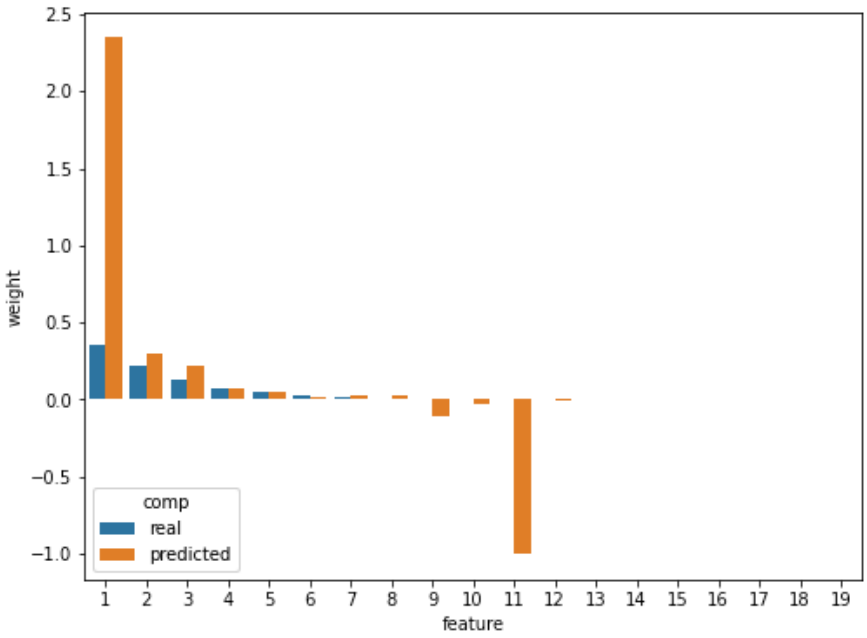
```
Error:  0.09076316848154202
```



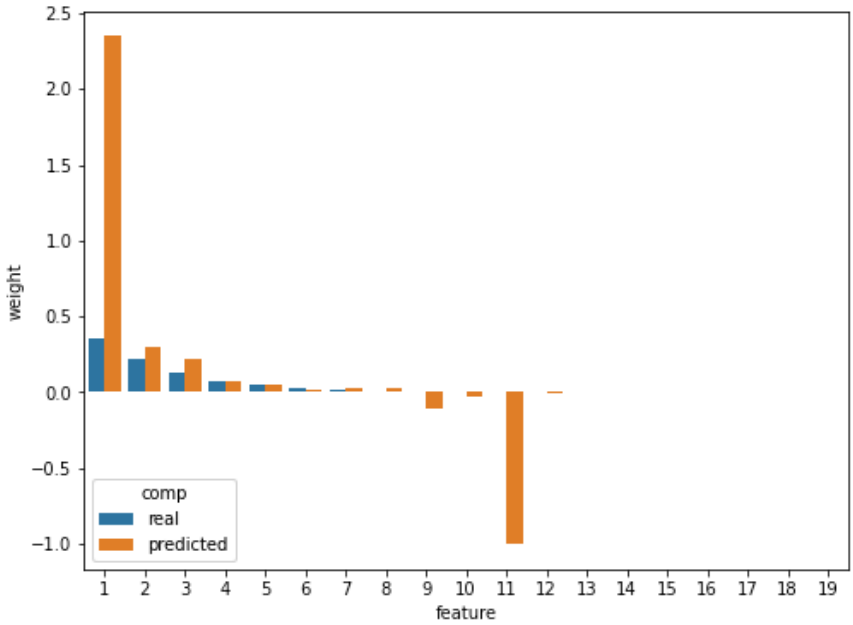
```
relevant_features = ["X1", "X2", "X3", "X4", "X5", "X6", "X7", "X8", "X9", "X12", "X14", "X15", "X19", "X20", "Y"]
rel = [a[1:] for a in relevant_features[:-1]]
new_data = data[relevant_features]
```

```
X_new = np.asarray(new_data.iloc[:,:-1])
y_new = np.asarray(new_data.iloc[:,-1:])
new_true_weights = []
for i in range(1, 21):
    if str(i) in rel:
        new_true_weights.append(w_actual[i])
    else:
```

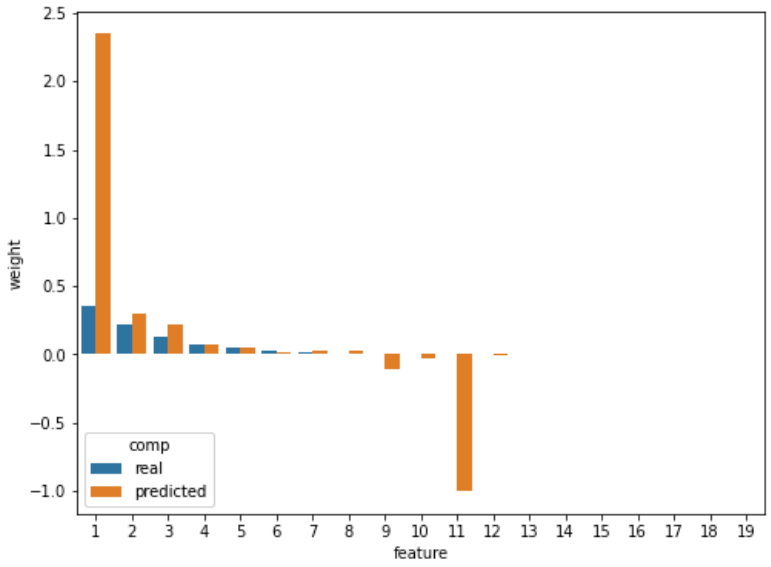
```
new_ridge = Regression()
w_lambda_2 = new_ridge.ridge_regression(X_new, y_new, 0.001)
compare_plot(new_true_weights, w_lambda_2.flatten(), features)
```



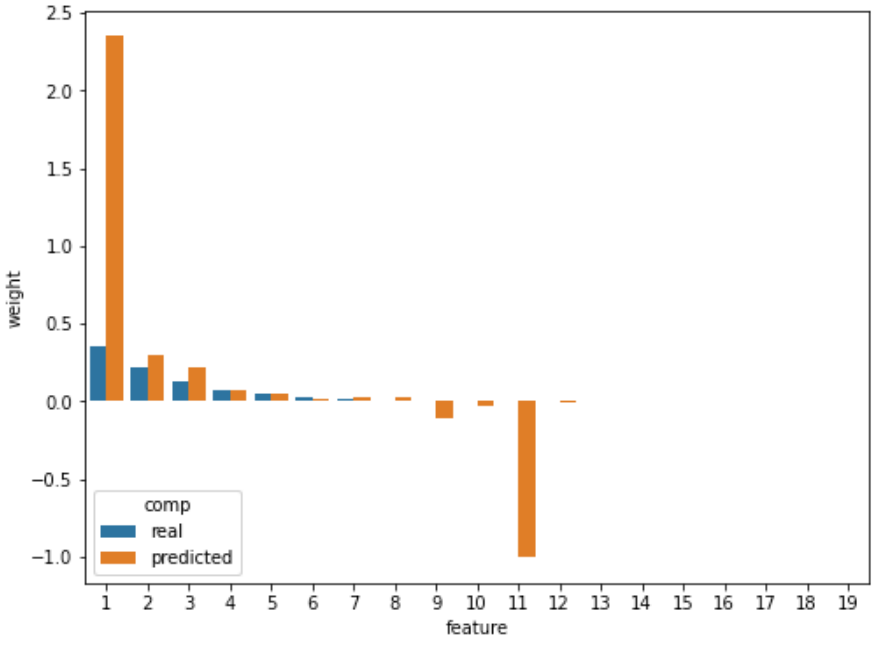

```
w_lambda_2 = new_ridge.ridge_regression(X_new, y_new, 0.01)
compare_plot(new_true_weights, w_lambda_2.flatten(), features)
```



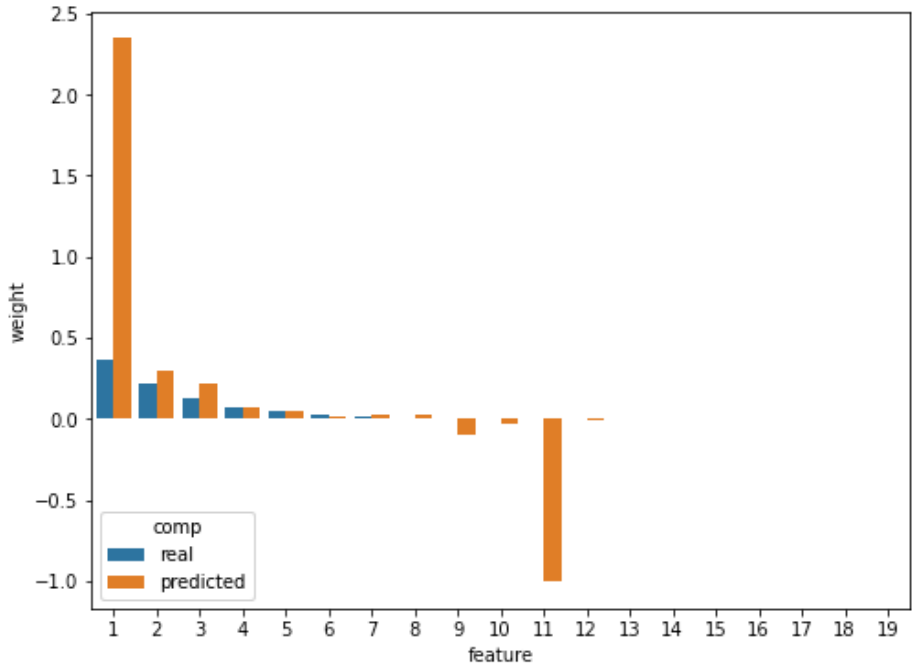
```
w_lambda_2 = new_ridge.ridge_regression(X_new, y_new, 0.1)
compare_plot(new_true_weights, w_lambda_2.flatten(), features)
```



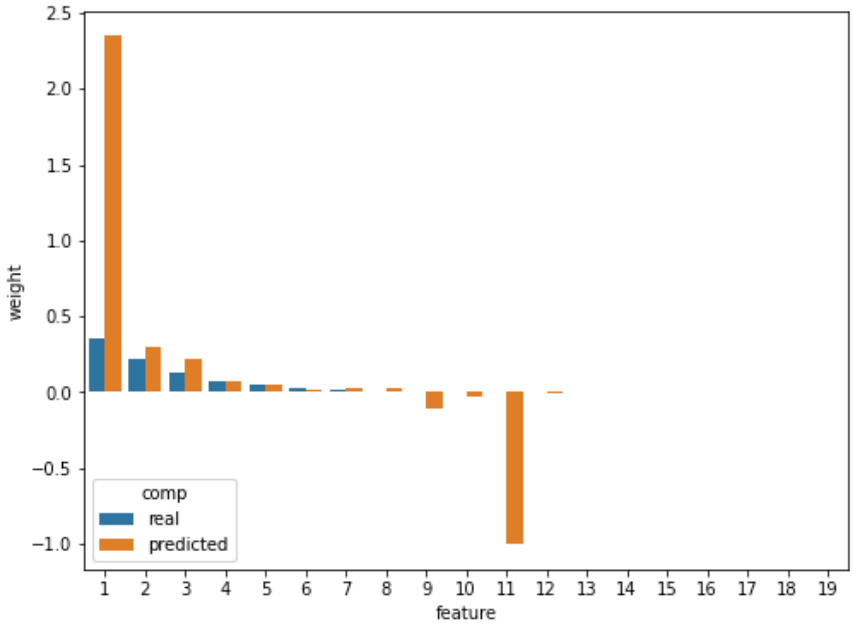

```
w_lambda_2 = new_ridge.ridge_regression(X_new, y_new, 0.5)
compare_plot(new_true_weights, w_lambda_2.flatten(), features)
```



```
w_lambda_2 = new_ridge.ridge_regression(X_new, y_new, 1)
compare_plot(new_true_weights, w_lambda_2.flatten(), features)
```




```
w_lambda_2 = new_ridge.ridge_regression(X_new, y_new)
compare_plot(new_true_weights, w_lambda_2.flatten(), features)
```



SVMs

1) Implement a barrier-method dual SVM solver. How can you (easily!) generate an initial feasible α 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.

This following is a barrier-method dual SVM solver will return all α_i including α_1 in the final answer. The results are in question2. As t increases, ε_t will be more similar to the optimal solution.

```
class SVM():
    def __init__(self):
        self.m = 4
    def fit(self, X, y):
        alpha = [1] * (self.m - 1)
        print("initial alpha: ", alpha)
        alpha_prime = 0
        beta = 0.01
        threshold = 0.00001
        steps = 1
        loss = 1
        updated_loss = 2
        while abs(updated_loss - loss) > threshold:
            loss = self.update_loss(X, y, steps, alpha)
            for i in range(self.m - 1):
                delta = self.update_weight(X, y, i + 1, steps, alpha)
                alpha[i] = alpha[i] + beta * delta
            steps += 1
            updated_loss = self.update_loss(X, y, steps, alpha)
        for i in range(self.m - 1):
            alpha_prime -= alpha[i] * y[i+1] * y[0]
        alpha.append(alpha_prime)
        print("final alpha is: ", alpha)
        return alpha
    def kernel(self, X, y):
        return (1 + X * y.T) ** 2
```

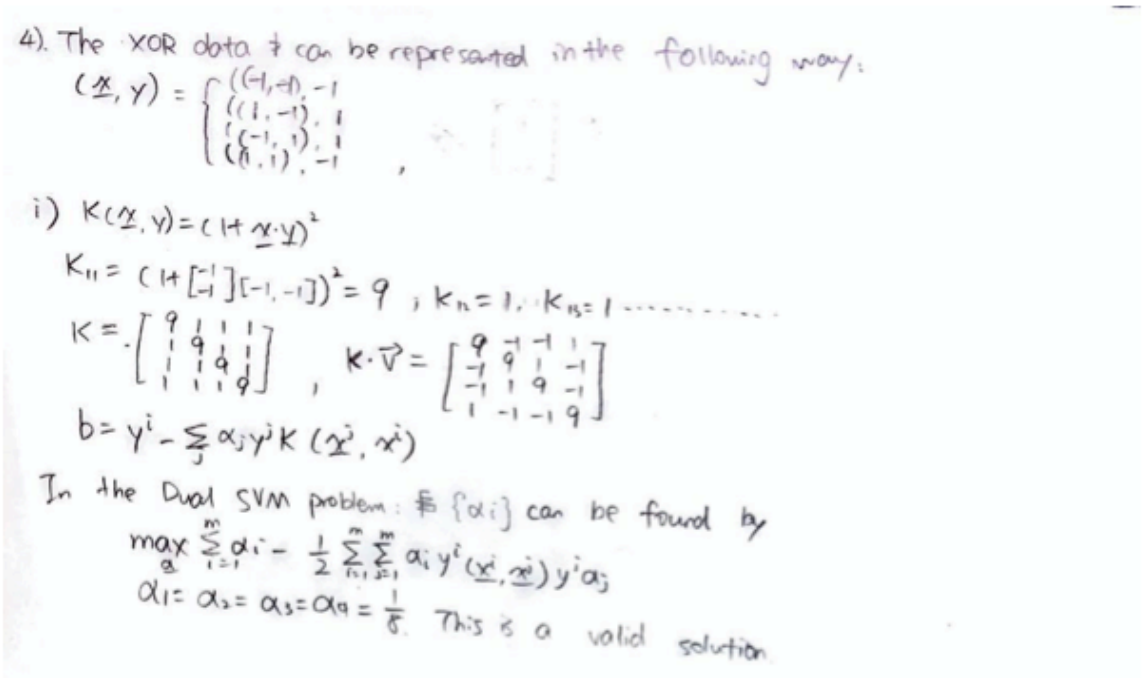
```
    def update_loss(self, X, y, step, alpha):
        n = len(alpha) + 1
        s = 'al:' + str(n)
        t = sp.var(s)
        dim = [0] * 6
        for i in range(1, self.m):
            dim[0] -= t[i] * (y[i] * y[0])
            dim[1] += t[i]
            dim[2] += t[i] * y[i] * self.kernel(X[i], X[0])
            for j in range(1, self.m):
                dim[3] += t[i] * y[i] * self.kernel(X[i], X[j]) * y[j] * t[j]

        D = (-1/2) * (pow(dim[0], 2) * self.kernel(X[0], X[0]) + 2 * y[0] * dim[0] * dim[2] + dim[3])
        for i in range(1, self.m):
            dim[4] += sp.log(t[i])
        dim[5] = sp.log(dim[0])
        loss = dim[0] + dim[1] + D + 1 / (step ** 2) * (dim[4] + dim[5])
        return loss.subs({al1:alpha[0], al2:alpha[1], al3:alpha[2]})
    def update_weight(self, X, y, k, step, alpha):
        n = len(alpha) + 1
        s = 'al:' + str(n)
        t = sp.var(s)
        dim = [0] * 6
        for i in range(1, self.m):
            dim[0] -= t[i] * (y[i] * y[0])
            dim[1] += t[i]
            dim[2] += t[i] * y[i] * self.kernel(X[i], X[0])
            for j in range(1, self.m):
                dim[3] += t[i] * y[i] * self.kernel(X[i], X[j]) * y[j] * t[j]

        D = (-1/2) * (pow(dim[0], 2) * self.kernel(X[0], X[0]) + 2 * y[0] * dim[0] * dim[2] + dim[3])
        for i in range(1, self.m):
            dim[4] += sp.log(t[i])
        dim[5] = sp.log(dim[0])
        loss = dim[0] + dim[1] + D + (1 / step ** 2) * (dim[4] + dim[5])
        diff_loss = sp.diff(loss, t[k])
        return diff_loss.subs({al1:alpha[0], al2:alpha[1], al3:alpha[2]})
```

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

Using my SVM solver, the solution is $\alpha 1 = \alpha 2 = \alpha 3 = \alpha 4 = 0.125$. Solving by hand in homework3, the solution is $\alpha 1 = \alpha 2 = \alpha 3 = \alpha 4 = 1/8$. They are exactly the same.

```
X = np.mat([[1, 1], [-1, -1], [1, -1], [-1, 1]])
y = [-1, -1, 1, 1]
s = SVM()
s.fit(X, y)
```

```
initial alpha:  [1, 1, 1]
final alpha is:  [0.125139850106026, 0.125094843551614, 0.125116862432028, 0.125071855877616]
```



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

Using the solution we just got, we can find that the error of our predicted value and true value is 0. So it correctly classifies the XOR data.

```
XOR_ans = {'A': [0,0,1,1], 'B': [0,1,0,1], 'Y': [0,1,1,0]}
XOR_data = pd.DataFrame(ans)
s2 = SVM()
s2.train(XOR_data.drop('Y', 1).values, XOR_data['Y'].values)
print("the error is: ", s2.predict(XOR_data.drop('Y', 1).values, XOR_data['Y'].values)[1])
```

```
the error is:  0.0
```