Ridge Regression

Problem:

Generate a predictor vector **X** of length **n = 100** (random vector **X**), as well as a noise vector **ε** of length **n = 100**. Generate a response vector **Y** of length **n = 100** according to the following model:

$$Y_i = \beta_0 + \beta_1 X_i + \beta_2 X_i^2 + \beta_3 X_i^3 + \epsilon_i$$

where:

- $\beta_0 = 50$
- $\beta_1 = 10$
- $\beta_2 = -20$
- $\beta_3 = 0.1$

Perform ridge regression using **X**, $\mathbf{X^2}$, $\mathbf{X^3}$, and $\mathbf{X^4}$ as predictors. Choose any two different values of **λ** (different from 0 and ∞). With each **λ**, perform ridge regression both **with** and **without** standardizing the predictors. Then, compare the results.

Note: No built-in functions are allowed.

In [1]:
```
import numpy as np
```

In [2]:
```
# set random seed for reproducibility
np.random.seed(6)
# define parameters
n = 100
x = np.random.rand(n)
epsilon = np.random.normal(0, 1, n)
b0 = 50
b1 = 10
b2 = -20
b3 = 0.1
```

In [3]:
```
# standardizing the data
x_mean = np.mean(x)
x_std = np.std(x)
x_standardized = (x - x_mean) / x_std
```

In [4]:
```
# generate Y
y = b0 + b1*x + b2*x**2 + b3*x**3 + epsilon
```

In [5]:
```
def ridge_regression(X, y, lmd):
    n, p = X.shape
    I = np.eye(p)
    beta_hat = np.linalg.inv(X.T @ X + lmd * I) @ X.T @ y
    return beta_hat
```

In [6]:
```
# create X matrix
X = np.column_stack((np.ones(n), x, x**2, x**3, x**4))
X_standardized = np.column_stack((np.ones(n), x_standardized, x_standardized**2, x_sta
```

In [7]:
```
lmd0, lmd1 = 0.1, 0.01
```

In [8]:
```
# fit the model

beta_hat = ridge_regression(X, y, lmd0)
# predict and calculate the mean squared error
y_hat = X @ beta_hat
mse = np.mean((y - y_hat)**2)
print(f"Not standardized & lambda = {lmd0}, MSE = {mse}")

beta_hat1 = ridge_regression(X, y, lmd1)
y_hat1 = X @ beta_hat1
mse1 = np.mean((y - y_hat1)**2)
print(f"Not standardized & lambda = {lmd1}, MSE = {mse1}")

beta_hat_ = ridge_regression(X_standardized, y, lmd0)
y_hat_ = X_standardized @ beta_hat_
mse_ = np.mean((y - y_hat_)**2)
print(f"Standardized & lambda = {lmd0}, MSE = {mse_}")

beta_hat1_ = ridge_regression(X_standardized, y, lmd1)
y_hat1_ = X_standardized @ beta_hat1_
mse1_ = np.mean((y - y_hat1_)**2)
print(f"Standardized & lambda = {lmd1}, MSE = {mse1_}")
```

```
Not standardized & lambda = 0.1, MSE = 0.794941936608605
Not standardized & lambda = 0.01, MSE = 0.7387468119546439
Standardized & lambda = 0.1, MSE = 0.7420973167905006
Standardized & lambda = 0.01, MSE = 0.7329252504766618
```

Conclusion

1. With both λ 0.1 and 0.01, the mse after standardizing X has dropped, indicating an improved perforamce of ridge regression with standarization.

2. By dropping λ from 0.1 to 0.01, the mse decreased, indicating a lower penalty on coefficients with less regularization, introducing smaller bias.

Lasso Regression

Problem:

Use the dataset you generated in Problem 1 and fit the model for the same set of predictors using **Lasso regression**.

Choose any two different values of **λ** (different from 0 and ∞).

With each **λ**, perform **Lasso regression** without standardizing the predictors. Then perform **Lasso regression** standardizing the predictors.

Questions:

- What can you conclude from these experiments?

Note: No built-in functions are allowed.

In [9]:
```
X.shape, y.shape, X_standardized.shape
```

Out[9]:
```
((100, 5), (100,), (100, 5))
```

In [10]:
```
class lasso_reg:
    def __init__(self, lmd, tol=1e-6, max_iter=1000):
        self.lmd = lmd
        self.tol = tol
        self.max_iter = max_iter
        self.coef_ = None

    def fit(self, X, Y):
        n, p = X.shape
        b = np.zeros(p)
        b_old = np.zeros(p)

        X_t_X = X.T @ X

        for _ in range(self.max_iter):
            for j in range(p):
                # compute the partial residual
                residual = Y - X @ b + X[:, j] * b[j]

                # update coefficient using soft-thresholding
                rho = X[:, j].T @ residual
                b[j] = self._soft_threshold(rho / X_t_X[j, j], self.lmd)

            # early stop if converge
            if np.linalg.norm(b - b_old, ord=2) < self.tol:
                break

            b_old = b.copy()

        self.coef_ = b

    def _soft_threshold(self, rho, lmd):
        if rho > lmd:
            return rho - lmd
        elif rho < -lmd:
            return rho + lmd
        else:
            return 0

    def predict(self, X):
        return X @ self.coef_

    def cal_mse(self, X, Y):
        Y_hat = self.predict(X)
```

```
        return np.mean((Y - Y_hat)**2)
```

In [11]:
```
lmd0, lmd1 = 0.1, 0.01
```

In [13]:
```
# train-test split
x_train = x[:80]
x_test = x[80:]
y_train = y[:80]
y_test = y[80:]
# standardizing the data
x_train_standardized = (x_train - np.mean(x_train)) / np.std(x_train)
x_test_standardized = (x_test - np.mean(x_train)) / np.std(x_train)
```

In [14]:
```
X_train = np.column_stack((np.ones(80), x_train, x_train**2, x_train**3, x_train**4))
X_train_standardized = np.column_stack((np.ones(80), x_train_standardized, x_train_sta
X_test = np.column_stack((np.ones(20), x_test, x_test**2, x_test**3, x_test**4))
X_test_standardized = np.column_stack((np.ones(20), x_test_standardized, x_test_standa
```

In [15]:
```
# case without standardizing the data
lasso0 = lasso_reg(lmd0)
lasso0.fit(X_train, y_train)
# calcualte train and test MSE
mse_train0 = lasso0.cal_mse(X_train, y_train)
mse_test0 = lasso0.cal_mse(X_test, y_test)
print(f"Not standardized & lambda = {lmd0}, Train MSE = {mse_train0}, Test MSE = {mse_

lasso1 = lasso_reg(lmd1)
lasso1.fit(X_train, y_train)
# calcualte train and test MSE
mse_train1 = lasso1.cal_mse(X_train, y_train)
mse_test1 = lasso1.cal_mse(X_test, y_test)
print(f"Not standardized & lambda = {lmd1}, Train MSE = {mse_train1}, Test MSE = {mse_

# case with standardizing the data
lasso0_ = lasso_reg(lmd0)
lasso0_.fit(X_train_standardized, y_train)
# calcualte train and test MSE
mse_train0_ = lasso0_.cal_mse(X_train_standardized, y_train)
mse_test0_ = lasso0_.cal_mse(X_test_standardized, y_test)
print(f"Standardized & lambda = {lmd0}, Train MSE = {mse_train0_}, Test MSE = {mse_tes

lasso1_ = lasso_reg(lmd1)
lasso1_.fit(X_train_standardized, y_train)
# calcualte train and test MSE
mse_train1_ = lasso1_.cal_mse(X_train_standardized, y_train)
mse_test1_ = lasso1_.cal_mse(X_test_standardized, y_test)
print(f"Standardized & lambda = {lmd1}, Train MSE = {mse_train1_}, Test MSE = {mse_tes
```

```
Not standardized & lambda = 0.1, Train MSE = 0.9795324278919608, Test MSE = 0.3642848
4774834435
Not standardized & lambda = 0.01, Train MSE = 0.8430831966174044, Test MSE = 0.420033
8173817821
Standardized & lambda = 0.1, Train MSE = 0.9319593820315231, Test MSE = 0.41624378101
88807
Standardized & lambda = 0.01, Train MSE = 0.7911449728278032, Test MSE = 0.5121604035
42953
```

Conclusion

1. With both λ 0.1 and 0.01, either test and train set, the mse after standardizing X has dropped, indicating an improved perforamce of lasso regression with standarization.
2. By dropping λ from 0.1 to 0.01, the mse deviate differnetly in tain and test sets. MSE would decrease in train set and increase in test set when λ drops from 0.1 to 0.01. Indicating decrease λ would increase the overfitting problem.