exercisesweek37

September 12, 2025

1 Exercises week 37

Implementing gradient descent for Ridge and ordinary Least Squares Regression

Date: **September 8-12, 2025**

1.1 Learning goals

After having completed these exercises you will have:

1. Your own code for the implementation of the simplest gradient descent approach applied to ordinary least squares (OLS) and Ridge regression
2. Be able to compare the analytical expressions for OLS and Ridge regression with the gradient descent approach
3. Explore the role of the learning rate in the gradient descent approach and the hyperparameter λ in Ridge regression
4. Scale the data properly

1.2 Simple one-dimensional second-order polynomial

We start with a very simple function

$$f(x) = 2 - x + 5x^2,$$

defined for $x \in [-2, 2]$. You can add noise if you wish.

We are going to fit this function with a polynomial ansatz. The easiest thing is to set up a second-order polynomial and see if you can fit the above function. Feel free to play around with higher-order polynomials.

[1]:
```
# Important Libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

1.3 Exercise 1, scale your data

Before fitting a regression model, it is good practice to normalize or standardize the features. This ensures all features are on a comparable scale, which is especially important when using regularization. Here we will perform standardization, scaling each feature to have mean 0 and standard deviation 1.

1.3.1 1a)

Compute the mean and standard deviation of each column (feature) in your design/feature matrix X. Subtract the mean and divide by the standard deviation for each feature.

We will also center the target y to mean 0. Centering y (and each feature) means the model does not require a separate intercept term, the data is shifted such that the intercept is effectively 0 . (In practice, one could include an intercept in the model and not penalize it, but here we simplify by centering.) Choose $n = 100$ data points and set up x, y and the design matrix X.

[2]:
```
# Generating the synthetic dataset
p_features = 2
n = 100
x = np.linspace(-3, 3, n)
y = 5 * x**2  - x + 2 + np.random.normal(0,0.2)
y = y.reshape(-1, 1)
```

[3]:
```
def polynomial_features(x, p, intercept=False):
    n = len(x)
    k = 0
    if intercept:
        X = np.zeros((n, p + 1))
        X[:, 0] = 1
        k += 1
    else:
        X = np.zeros((n, p))

    for i in range(1, p +1):
        X[:, i + k-1] = x**i
    return X
```

[4]:
```
X = polynomial_features(x, p=p_features, intercept=False)

# Standardize features (zero mean, unit variance for each feature)
X_mean = X.mean(axis=0)
X_std = X.std(axis=0)

if np.any(X_std == 0):
    X_std[X_std == 0] = 1  # safeguard to avoid division by zero for constant␣
 ↪features

X_norm = (X - X_mean) / X_std
```

```
X = X_norm
# Center the target to zero mean (optional, to simplify intercept handling)
y_mean = y.mean(axis=0)
y_centered = y - y_mean
y = y_centered
```

Fill in the necessary details. Do we need to center the y-values?

After this preprocessing, each column of X_{norm} has mean zero and standard deviation 1 and y_{centered} has mean 0. This makes the optimization landscape nicer and ensures the regularization penalty $\lambda \sum_j \theta_j^2$ in Ridge regression treats each coefficient fairly (since features are on the same scale).

[5]:
```
# Just to check that it is done correctly
print("X_norm: \t",X_norm.mean(axis=0), X_norm.std(axis=0))
print("y_centered: \t",y_centered.mean(axis=0))
```

```
X_norm:          [5.77315973e-17 7.54951657e-17] [1. 1.]
y_centered:      [2.84217094e-16]
```

1.4 Exercise 2, calculate the gradients

Find the gradients for OLS and Ridge regression using the mean-squared error as cost/loss function.

$$\nabla_\theta C(\theta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{100} (\theta_0 + \theta_1 x_i - y_i) \\ \sum_{i=1}^{100} (x_i(\theta_0 + \theta_1 x_i) - y_i x_i) \end{bmatrix} = \frac{2}{n} X^T (X\theta - \mathbf{y}),$$

[ ]:
```
def Gradient_OLS(X, y,theta, eta=0.01, n=100):
    return (2.0/n)*X.T @ (X @ theta-y)
```

$$\nabla_\theta C_{\text{ridge}}(\theta) = \frac{2}{n}\begin{bmatrix} \sum_{i=1}^{100} (\theta_0 + \theta_1 x_i - y_i) \\ \sum_{i=1}^{100} (x_i(\theta_0 + \theta_1 x_i) - y_i x_i) \end{bmatrix} + 2\lambda \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = 2(\frac{1}{n} X^T (X\theta - \mathbf{y}) + \lambda\theta).$$

[ ]:
```
def Gradient_Ridge(X, y, theta, eta=0.01, lambda_param=1.0,n=100):

    return (2.0/n)*X.T @ (X @ theta-y) + 2*lambda_param*theta
```

1.5 Exercise 3, using the analytical formulae for OLS and Ridge regression to find the optimal paramters θ

[9]:
```
# Set regularization parameter, either a single value or a vector of values
# Note that lambda is a python keyword. The lambda keyword is used to create␣
 ↪small, single-expression functions without a formal name. These are often␣
 ↪called "anonymous functions" or "lambda functions."
lam = 0.01
theta = np.random.randn(2,1)
```

```
# Analytical form for OLS and Ridge solution: theta_Ridge = (X^T X + lambda *␣
↪I)^{-1} X^T y and theta_OLS = (X^T X)^{-1} X^T y
I = np.eye(2)
theta_closed_formRidge = np.linalg.inv(X.T @ X + lam * np.identity(len(X.T))) @␣
↪X.T @ y
theta_closed_formOLS = np.linalg.inv(X.T @ X) @ X.T @ y

print("Closed-form Ridge coefficients:", theta_closed_formRidge)
print("Closed-form OLS coefficients:", theta_closed_formOLS)
```

```
Closed-form Ridge coefficients: [[-1.74928386]
 [13.68402453]]
Closed-form OLS coefficients: [[-1.74945879]
 [13.68539293]]
```

This computes the Ridge and OLS regression coefficients directly. The identity matrix I has the same size as $X^T X$. It adds λ to the diagonal of $X^T X$ for Ridge regression. We then invert this matrix and multiply by $X^T y$. The result for θ is a NumPy array of shape (n_features,) containing the fitted parameters θ.

1.5.1 3a)

Finalize, in the above code, the OLS and Ridge regression determination of the optimal parameters θ.

[11]:
```
print("Optimal Parameters (Closed-form)")
print("Closed-form Ridge coefficients:\n", theta_closed_formRidge)
print("Closed-form OLS coefficients:\n", theta_closed_formOLS)
```

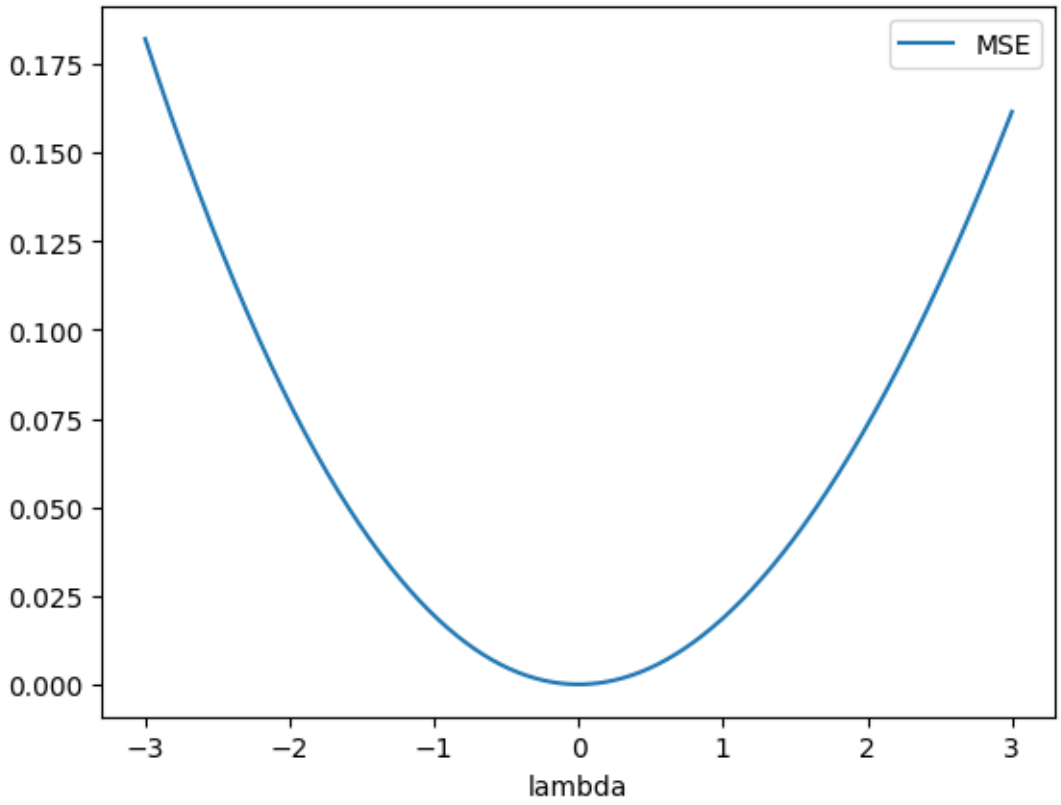
```
Optimal Parameters (Closed-form)
Closed-form Ridge coefficients:
 [[-1.74928386]
 [13.68402453]]
Closed-form OLS coefficients:
 [[-1.74945879]
 [13.68539293]]
```

1.5.2 3b)

Explore the results as function of different values of the hyperparameter λ. See for example exercise 4 from week 36.

[31]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

plot_predict = []
points = 1000

lam =  np.linspace(-3, 3, points)
```

```
for i in range(points):
    theta = np.random.randn(2,1)
    # Analytical form for OLS and Ridge solution: theta_Ridge = (X^T X + lambda
 ↪* I)^{-1} X^T y and theta_OLS = (X^T X)^{-1} X^T y
    I = np.eye(2)
    theta_closed_formRidge = np.linalg.inv(X.T @ X + lam[i] * np.identity(len(X.
 ↪T))) @ X.T @ y
    mse_predict = mean_squared_error(y, X @ theta_closed_formRidge)
    plot_predict.append(mse_predict)



plt.plot(lam, plot_predict, label="MSE")
plt.xlabel("lambda")
plt.legend()
plt.show()
```



We see that the model errors blows up quite fast once λ has i higher absolute value.

1.6 Exercise 4, Implementing the simplest form for gradient descent

Alternatively, we can fit the ridge regression model using gradient descent. This is useful to visualize the iterative convergence and is necessary if n and p are so large that the closed-form might be too slow or memory-intensive. We derive the gradients from the cost functions defined above. Use the gradients of the Ridge and OLS cost functions with respect to the parameters θ and set up (using the template below) your own gradient descent code for OLS and Ridge regression.

Below is a template code for gradient descent implementation of ridge:

[35]:
```
# Gradient descent parameters, learning rate eta first
eta = 0.0001
lam = 0.001
# Then number of iterations
num_iters = 100000

# Initialize weights for gradient descent
theta_gdOLS = np.random.randn(2,1)
theta_gdRidge = np.random.randn(2,1)

# Gradient descent loop
for t in range(num_iters):
    # Compute gradients for OSL and Ridge
    grad_OLS = Gradient_OLS(X_norm, y_centered, eta=eta,theta=theta_gdOLS)
    theta_gdOLS -= eta * grad_OLS
    grad_Ridge= Gradient_Ridge(X_norm, y_centered, eta=eta,␣
 ↪lambda_param=lam,theta=theta_gdRidge)
    theta_gdRidge -= eta * grad_Ridge

# After the loop, theta contains the fitted coefficients

print("Gradient Descent OLS coefficients:", theta_gdOLS)
print("Gradient Descent Ridge coefficients:", theta_gdRidge)
```

```
Gradient Descent OLS coefficients: [[-1.74945879]
 [13.6853929 ]]
Gradient Descent Ridge coefficients: [[-1.74771108]
 [13.67172118]]
```

1.6.1 4a)

Write first a gradient descent code for OLS only using the above template. Discuss the results as function of the learning rate parameters and the number of iterations

[ ]:
```
n_iter = 1000
etas =  np.logspace(-3, 1, 3)

iterations = [10, 1000]
```

```
for eta in etas:
    theta_gdOLS = np.random.randn(2,1)
    for n_iter in iterations:
        for i in range(n_iter):
            grad_OLS = Gradient_OLS(X_norm, y_centered,␣
 ↪eta=eta,theta=theta_gdOLS)
            theta_gdOLS -= eta * grad_OLS
        print("Eta:", eta, ", iterations:", n_iter)
        print("Theta OLS:", theta_gdOLS[0], theta_gdOLS[1])
        print("\n")
```

```
Eta: 0.001 , iterations: 10
Theta OLS: [0.64795865] [0.6027791]


Eta: 0.001 , iterations: 1000
Theta OLS: [-1.42565275] [11.91839594]


Eta: 0.1 , iterations: 10
Theta OLS: [-1.4916276] [12.216093]


Eta: 0.1 , iterations: 1000
Theta OLS: [-1.74945879] [13.68539293]


Eta: 10.0 , iterations: 10
Theta OLS: [6.0253295e+12] [-9.00636199e+13]


Eta: 10.0 , iterations: 1000
Theta OLS: [nan] [nan]



C:\Users\Anton\AppData\Local\Temp\ipykernel_20124\871024197.py:2:
RuntimeWarning: overflow encountered in matmul
  gradient = (2.0/n)*X.T @ (X @ theta-y)
C:\Users\Anton\AppData\Local\Temp\ipykernel_20124\871024197.py:2:
RuntimeWarning: invalid value encountered in matmul
  gradient = (2.0/n)*X.T @ (X @ theta-y)
```

Again we see that the when eta becomes too large the model blows up, while when eta is too small it takes a long time to learn what the optimal parameters are. Meaning that the cost for finding the optimal solution is higher.

1.6.2 4b)

Write then a similar code for Ridge regression using the above template. Try to add a stopping parameter as function of the number iterations and the difference between the new and old θ values. How would you define a stopping criterion?

[37]:
```
n_iter = 1000
etas =  np.logspace(-5, 1, 3)

lam_params = np.logspace(-5, 0, 3)
iterations = [10, 10000]

for eta in etas:
    theta = np.random.randn(2,1)
    for lam in lam_params:
        for n_iter in iterations:
            for i in range(n_iter):
                grad = Gradient_Ridge(X_norm, y_centered, eta=eta,theta=theta,␣
 ↪lambda_param=lam)
                if np.absolute(eta*grad).sum() < 1e-8:
                    print("Converged after", i, "iterations")
                    break
                theta -= eta * grad

            print("Eta:", eta,", lambda",lam, ", iterations:", n_iter)
            print("Theta Ridge:", theta[0], theta[1])
            print("\n")
```

```
Eta: 1e-05 , lambda 1e-05 , iterations: 10
Theta Ridge: [-1.22159502] [2.0798811]


Eta: 1e-05 , lambda 1e-05 , iterations: 10000
Theta Ridge: [-1.31727904] [4.18361669]


Eta: 1e-05 , lambda 0.0031622776601683794 , iterations: 10
Theta Ridge: [-1.31736464] [4.18551423]


Eta: 1e-05 , lambda 0.0031622776601683794 , iterations: 10000
Theta Ridge: [-1.39491187] [5.90464085]


Eta: 1e-05 , lambda 1.0 , iterations: 10
Theta Ridge: [-1.39470383] [5.90501601]
```

```
Eta: 1e-05 , lambda 1.0 , iterations: 10000
Theta Ridge: [-1.2232759] [6.21415549]


Eta: 0.01 , lambda 1e-05 , iterations: 10
Theta Ridge: [0.52465994] [2.90308001]


Converged after 846 iterations
Eta: 0.01 , lambda 1e-05 , iterations: 10000
Theta Ridge: [-1.74944121] [13.68525567]


Eta: 0.01 , lambda 0.0031622776601683794 , iterations: 10
Theta Ridge: [-1.74843272] [13.67736652]


Converged after 557 iterations
Eta: 0.01 , lambda 0.0031622776601683794 , iterations: 10000
Theta Ridge: [-1.74394401] [13.64225278]


Eta: 0.01 , lambda 1.0 , iterations: 10
Theta Ridge: [-1.45261164] [11.36326341]


Converged after 413 iterations
Eta: 0.01 , lambda 1.0 , iterations: 10000
Theta Ridge: [-0.87472942] [6.84269668]


Eta: 10.0 , lambda 1e-05 , iterations: 10
Theta Ridge: [1.02897634e+13] [-7.66389246e+13]


Eta: 10.0 , lambda 1e-05 , iterations: 10000
Theta Ridge: [nan] [nan]


Eta: 10.0 , lambda 0.0031622776601683794 , iterations: 10
Theta Ridge: [nan] [nan]



C:\Users\Anton\AppData\Local\Temp\ipykernel_18124\2281413180.py:3:
RuntimeWarning: overflow encountered in matmul
  return (2.0/n)*X.T @ (X @ theta-y) + 2*lambda_param*theta
```

```
C:\Users\Anton\AppData\Local\Temp\ipykernel_18124\2281413180.py:3:
RuntimeWarning: invalid value encountered in matmul
  return (2.0/n)*X.T @ (X @ theta-y) + 2*lambda_param*theta

Eta: 10.0 , lambda 0.0031622776601683794 , iterations: 10000
Theta Ridge: [nan] [nan]


Eta: 10.0 , lambda 1.0 , iterations: 10
Theta Ridge: [nan] [nan]


Eta: 10.0 , lambda 1.0 , iterations: 10000
Theta Ridge: [nan] [nan]
```

1.7 Exercise 5, Ridge regression and a new Synthetic Dataset

We create a synthetic linear regression dataset with a sparse underlying relationship. This means we have many features but only a few of them actually contribute to the target. In our example, we'll use 10 features with only 3 non-zero weights in the true model. This way, the target is generated as a linear combination of a few features (with known coefficients) plus some random noise. The steps we include are:

Decide on the number of samples and features (e.g. 100 samples, 10 features). Define the **true** coefficient vector with mostly zeros (for sparsity). For example, we set $\hat{\theta} = [5.0, -3.0, 0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 0.0]$, meaning only features 0, 1, and 6 have a real effect on y.

Then we sample feature values for X randomly (e.g. from a normal distribution). We use a normal distribution so features are roughly centered around 0. Then we compute the target values y using the linear combination $X\hat{\theta}$ and add some noise (to simulate measurement error or unexplained variance).

Below is the code to generate the dataset:

[88]:
```python
import numpy as np

# Set random seed for reproducibility
np.random.seed(0)

# Define dataset size
n_samples = 100
n_features = 10

# Define true coefficients (sparse linear relationship)
theta_true = np.array([5.0, -3.0, 0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 0.0])

# Generate feature matrix X (n_samples x n_features) with random values
```

```
X = np.random.randn(n_samples, n_features)  # standard normal distribution

# Generate target values y with a linear combination of X and theta_true, plus␣
 ↪noise
noise = 0.5 * np.random.randn(n_samples)  # Gaussian noise
y = X @ theta_true + noise
y = y.reshape(-1,1)
y_true = X @ theta_true
y_true = y_true.reshape(-1,1)
theta_true = theta_true.reshape(-1,1)
```

[89]:
```
theta_true
```

[89]:
```
array([[ 5.],
       [-3.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 0.],
       [ 2.],
       [ 0.],
       [ 0.],
       [ 0.]])
```

This code produces a dataset where only features 0, 1, and 6 significantly influence y. The rest of the features have zero true coefficient. For example, feature 0 has a true weight of 5.0, feature 1 has -3.0, and feature 6 has 2.0, so the expected relationship is:

$$y \approx 5 \times x_0 \; - \; 3 \times x_1 \; + \; 2 \times x_6 \; + \; \text{noise}.$$

You can remove the noise if you wish to.

Try to fit the above data set using OLS and Ridge regression with the analytical expressions and your own gradient descent codes.

If everything worked correctly, the learned coefficients should be close to the true values [5.0, -3.0, 0.0, ..., 2.0, ...] that we used to generate the data. Keep in mind that due to regularization and noise, the learned values will not exactly equal the true ones, but they should be in the same ballpark. Which method (OLS or Ridge) gives the best results?

[90]:
```
lam = 0.1
#Analytical Solution
theta_closed_formOLS = np.linalg.inv(X.T @ X) @ X.T @ y
theta_closed_formRidge = np.linalg.inv(X.T @ X + lam * np.identity(len(X.T))) @␣
 ↪X.T @ y

theta_closed_formOLS_true = np.linalg.inv(X.T @ X) @ X.T @ y_true
```

```
theta_closed_formRidge_true = np.linalg.inv(X.T @ X + lam * np.identity(len(X.
 ↪T))) @ X.T @ y_true

print("Optimal Parameters (Closed-form)")
print("Closed-form OLS coefficients:\n", theta_closed_formOLS)
print("Closed-form Ridge coefficients:\n", theta_closed_formRidge)
print("\n")
print("Difference between true value with noise OLS, Ridge: ",np.
 ↪abs(theta_closed_formOLS - theta_true).sum(), np.abs(theta_closed_formRidge␣
 ↪- theta_true).sum())
print("Difference between true value OLS without noise, Ridge: ",np.
 ↪abs(theta_closed_formOLS_true - theta_true).sum(), np.
 ↪abs(theta_closed_formRidge_true - theta_true).sum())
```

```
Optimal Parameters (Closed-form)
Closed-form OLS coefficients:
 [[ 5.00905318e+00]
 [-3.00383337e+00]
 [-1.62718294e-02]
 [ 1.44819819e-01]
 [-7.16006510e-02]
 [-4.29656382e-02]
 [ 2.05558117e+00]
 [ 1.97583716e-03]
 [ 4.11922237e-02]
 [-5.10225177e-02]]
Closed-form Ridge coefficients:
 [[ 5.00410898e+00]
 [-2.99968373e+00]
 [-1.63065915e-02]
 [ 1.45477818e-01]
 [-7.25435409e-02]
 [-4.37776623e-02]
 [ 2.05285140e+00]
 [ 2.26563823e-03]
 [ 4.02214213e-02]
 [-5.10893259e-02]]


Difference between true value with noise OLS, Ridge:  0.4383162375780215
0.42895864891638424
Difference between true value OLS without noise, Ridge:  2.706168622523819e-15
0.015904259027352978
```

In the analytical case the Ridge seems to give the best values when we include noise but it is still very close, while if we exclude noise OLS gives the best results without a doubt.

[95]:
```
# Gradient descent parameters, learning rate eta first
eta = 0.01
lam = 0.01
# Then number of iterations
num_iters = 1000

# Initialize weights for gradient descent
theta_gdOLS = np.random.randn(10,1)
theta_gdOLS_true = np.random.randn(10,1)
theta_gdRidge = np.random.randn(10,1)
theta_gdRidge_true = np.random.randn(10,1)

# Gradient descent loop
for t in range(num_iters):
    # Compute gradients for OSL and Ridge
    grad_OLS = Gradient_OLS(X, y, eta=eta,theta=theta_gdOLS)
    grad_OLS_true = Gradient_OLS(X, y_true, eta=eta,theta=theta_gdOLS_true)
    theta_gdOLS -= eta * grad_OLS
    theta_gdOLS_true -= eta * grad_OLS_true
    grad_Ridge= Gradient_Ridge(X, y, eta=eta,␣
 ↪lambda_param=lam,theta=theta_gdRidge)
    grad_Ridge_true= Gradient_Ridge(X, y_true, eta=eta,␣
 ↪lambda_param=lam,theta=theta_gdRidge_true)
    theta_gdRidge -= eta * grad_Ridge
    theta_gdRidge_true -= eta * grad_Ridge_true

# After the loop, theta contains the fitted coefficients

print("Gradient Descent OLS coefficients:", theta_gdOLS)
print("Gradient Descent Ridge coefficients:", theta_gdRidge)
print("\n")
print("Difference between true value with noise OLS, Ridge: ",np.
 ↪abs(theta_gdOLS - theta_true).sum(), np.abs(theta_gdRidge - theta_true).
 ↪sum())
print("Difference between true value OLS without noise, Ridge: ",np.
 ↪abs(theta_gdOLS_true - theta_true).sum(), np.abs(theta_gdRidge_true -␣
 ↪theta_true).sum())
```

```
Gradient Descent OLS coefficients: [[ 5.00905252e+00]
 [-3.00382578e+00]
 [-1.62739389e-02]
 [ 1.44827775e-01]
 [-7.16050075e-02]
 [-4.29733199e-02]
 [ 2.05556653e+00]
 [ 1.97484293e-03]
 [ 4.11918375e-02]
```

```
 [-5.10178345e-02]]
Gradient Descent Ridge coefficients: [[ 4.96007396e+00]
 [-2.96287251e+00]
 [-1.66154179e-02]
 [ 1.51238957e-01]
 [-8.08084651e-02]
 [-5.09123527e-02]
 [ 2.02862591e+00]
 [ 4.83525687e-03]
 [ 3.16594815e-02]
 [-5.16761120e-02]]


Difference between true value with noise OLS, Ridge:  0.4383093883217313
0.4934254850344568
Difference between true value OLS without noise, Ridge:  5.538304778910893e-05
0.15685855841264412
```

Which is equivalent to what we found in the analytical case, except there has been done a lot more computation and the solutions are still not as good yet.