Exercises week 37

Implementing gradient descent for Ridge and ordinary Least Squares Regression

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

Code can be found at

https://github.com/viktorbgulbrandsen/fysstk3155/tree/main

Simple one-dimensional second-order polynomial

We start with a very simple function defined for $x \in [-2, 2]$. You can add noise if you wish.

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

In [133…

```python
import numpy as np

def sample_from_function(n=100, noise_level=0.1):
    noise = np.random.normal(0, noise_level, n)
    x = np.linspace(-2, 2, n)
    y = 2 - x + 5 * x**2 + noise
    return x, y

x, y = sample_from_function(n=100, noise_level=0.1)
# print(x)
# print(y)
```

1a)

We first need to create the feature matrix.

In [134…

```python
def polynomial_features(x, p):
    n = len(x)
    X = np.zeros((n, p+1))
    for j in range(p+1):
        X[:, j] = x**j
    return X

X = polynomial_features(x, 2)
# print(X.shape)
```

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

We will also center the target $\boldsymbol{y}$ to mean 0. Centering $\boldsymbol{y}$ (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 $\boldsymbol{x}$, $\boldsymbol{y}$ and the design matrix $\boldsymbol{X}$.

In [135…

```python
X_mean = X.mean(axis=0)
X_std = X.std(axis=0)
X_std[X_std == 0] = 1
X_norm = (X - X_mean) / X_std

y_mean = y.mean()
y_centered = y - y_mean

def check_centering():
    print(f"y mean: {y.mean()}")
    print(f"y_centered mean: {y_centered.mean()}")


    from matplotlib import pyplot as plt
    import matplotlib.pyplot as plt

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))
    ax1.hist(y, bins=20, alpha=0.7)
    ax1.axvline(y.mean(), color='red', linestyle='--')
    ax1.set_xlim(-5, 25)

    ax2.hist(y_centered, bins=20, alpha=0.7)
    ax2.axvline(y_centered.mean(), color='red', linestyle='--')
    ax2.set_xlim(-5, 25)

    plt.show()

# check_centering()
```

Both the plot and the values confirm that we have centered

Exercise 2, calculate the gradients

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

In [136…

```python
def gradient_ols(X, y, theta):
    n = len(y)
    return (1/n) * X.T @ (X @ theta - y)

def gradient_ridge(X, y, theta, lam):
    n = len(y)
    return (1/n) * X.T @ (X @ theta - y) + 2 * lam * theta
```

These gradients are more throughly explored in the previous exercise weeks.

Exercise 3, using the analytical formulae for OLS and Ridge regression to find the optimal parameters $\boldsymbol{\theta}$

3a)

Finalize, in the above code, the OLS and Ridge regression determination of the optimal parameters $\boldsymbol{\theta}$.

In [137…

```
lam = 0.01

n_features = X_norm.shape[1]
I = np.eye(n_features)
theta_closed_formRidge = np.linalg.pinv(X_norm.T @ X_norm + lam * I) @ X_norm.T @ y
theta_closed_formOLS = np.linalg.pinv(X_norm.T @ X_norm) @ X_norm.T @ y_centered

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

```
closed-form Ridge coefficients: [ 0.         -1.16815283  6.07965364]
closed-form OLS coefficients: [ 0.         -1.16826964  6.0802616 ]
```

We use np.linalg psuedo-inv.

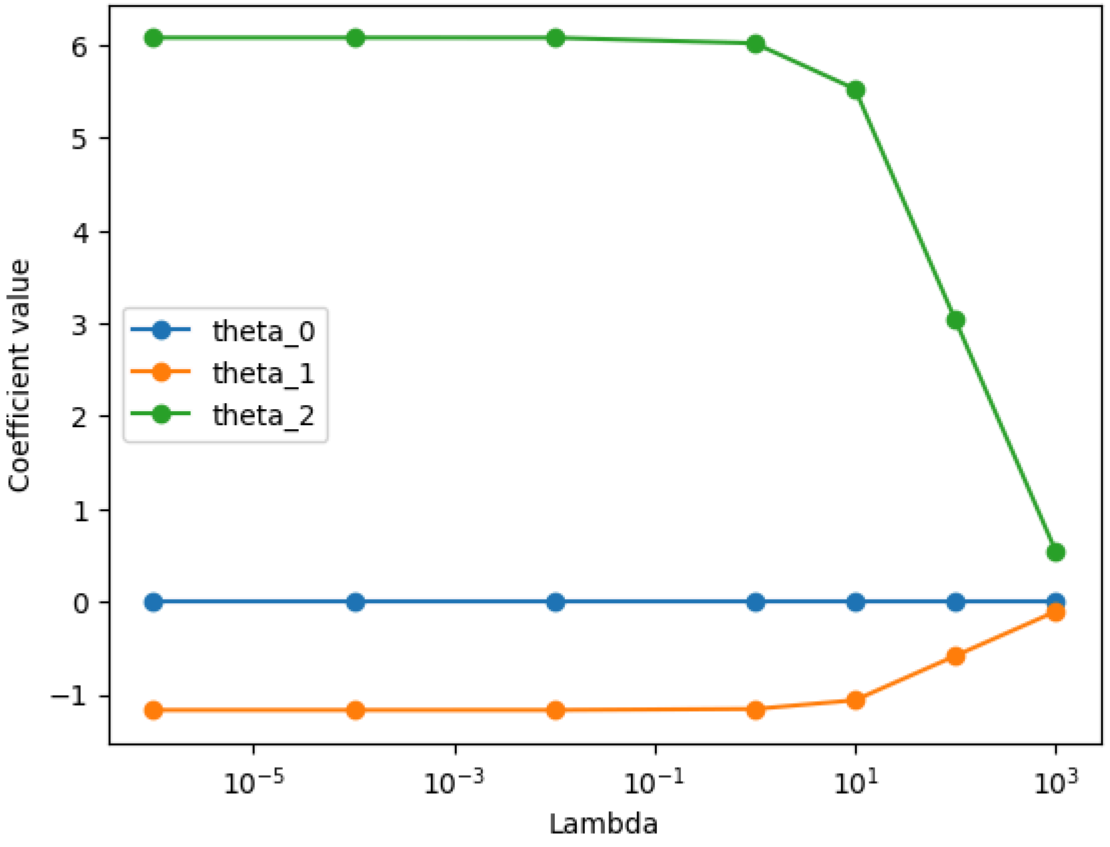
3b)

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

In [138…

```
lambdas = [1e-6, 1e-4, 1e-2, 1, 10, 100, 1000]
ridge_coeffs = []

for lam in lambdas:
    theta_ridge = np.linalg.inv(X_norm.T @ X_norm + lam * I) @ X_norm.T @ y_centere
    ridge_coeffs.append(theta_ridge)
    print(f"Lambda={lam}: {theta_ridge}")
import matplotlib.pyplot as plt
ridge_coeffs = np.array(ridge_coeffs)
for i in range(n_features):
    plt.plot(lambdas, ridge_coeffs[:, i], marker='o', label=f'theta_{i}')
plt.xscale('log')
plt.xlabel('Lambda')
plt.ylabel('Coefficient value')
plt.legend()
plt.show()
```

```
Lambda=1e-06: [ 0.         -1.16826963  6.08026154]
Lambda=0.0001: [ 0.         -1.16826848  6.08025552]
Lambda=0.01: [ 0.         -1.16815283  6.07965364]
Lambda=1: [ 0.         -1.15670262  6.02006099]
Lambda=10: [ 0.         -1.06206331  5.52751055]
Lambda=100: [ 0.         -0.58413482  3.0401308 ]
Lambda=1000: [ 0.         -0.10620633  0.55275105]
```



The plot shows the evolution of Ridge regression coefficients as a function of the regularization parameter lambda.
Starting from values close to the ordinary least squares (OLS) solution at small lambda, the coefficients shrink progressively as lambda increases reflecting the penalty set in the new (i.e Ridge) loss function.

Smaller coefficient remain stable for moderate level of lambda, but larger ones are pushed down.

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 $\boldsymbol{\theta}$ and set up your own gradient descent code for OLS and Ridge regression.

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

In [139…

```python
etas = [0.001, 0.1, 1.0]
iters = [100, 1000, 5000]

for eta in etas:
    for num_iters in iters:
        theta_gd = np.zeros(n_features)
        for t in range(num_iters):
            grad_OLS = gradient_ols(X_norm, y_centered, theta_gd)
            theta_gd = theta_gd - eta * grad_OLS

        diff = np.linalg.norm(theta_gd - theta_closed_formOLS)
        print(f"eta={eta:<6}, iters={num_iters:<5} -> GD coeffs={theta_gd}, "
              f"||diff||={diff:.4e}")

print("\nClosed-form OLS coefficients:", theta_closed_formOLS)
```

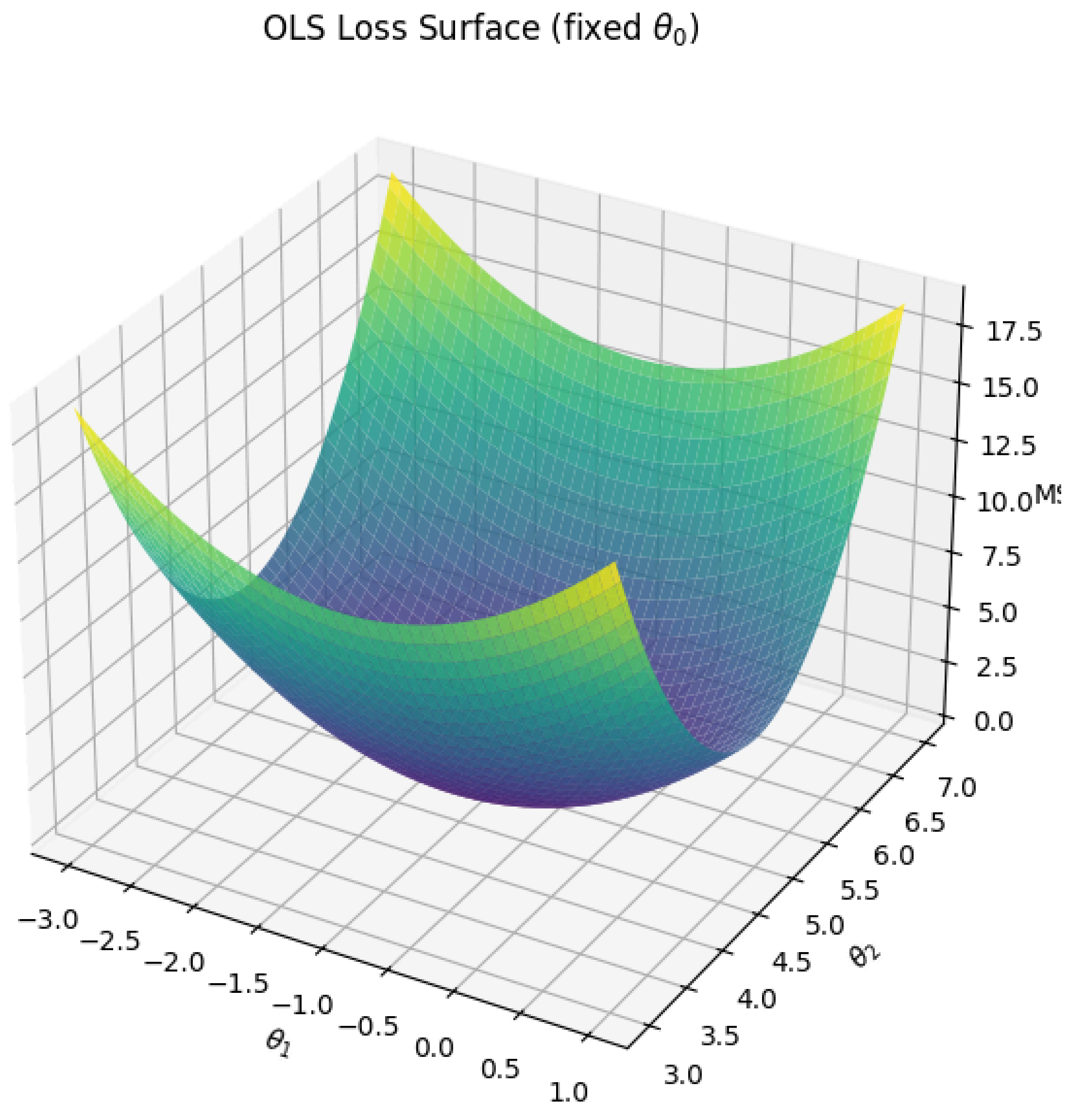
```
eta=0.001 , iters=100   -> GD coeffs=[ 0.         -0.11122844  0.57888865], ||diff||
=5.6020e+00
eta=0.001 , iters=1000  -> GD coeffs=[ 0.         -0.73870224  3.84457723], ||diff||
=2.2766e+00
eta=0.001 , iters=5000  -> GD coeffs=[ 0.         -1.16041757  6.03939548], ||diff||
=4.1614e-02
eta=0.1   , iters=100   -> GD coeffs=[ 0.         -1.16823861  6.0801001 ], ||diff||
=1.6445e-04
eta=0.1   , iters=1000  -> GD coeffs=[ 0.         -1.16826964  6.0802616 ], ||diff||
=6.3156e-15
eta=0.1   , iters=5000  -> GD coeffs=[ 0.         -1.16826964  6.0802616 ], ||diff||
=6.3156e-15
eta=1.0   , iters=100   -> GD coeffs=[ 0.         -1.16826964  6.0802616 ], ||diff||
=1.7902e-15
eta=1.0   , iters=1000  -> GD coeffs=[ 0.         -1.16826964  6.0802616 ], ||diff||
=1.7902e-15
eta=1.0   , iters=5000  -> GD coeffs=[ 0.         -1.16826964  6.0802616 ], ||diff||
=1.7902e-15

Closed-form OLS coefficients: [ 0.         -1.16826964  6.0802616 ]
```

In [ ]:

As we can see from the prints, gradient descent results depend on learning rates and number of iterations. Naturally, with small learning rate, $\eta = 0.001$ and 100 iterations, the algortihm converges slowly. Only after 5000 iterations does it come decently close. For learning rate $\eta = 1$ the algorithm converges very quickly. Looking at the plot below, the loss surface for OLS, we can see that such a large learning rate is not an issue here because the surface is a well-conditioned convex paraboloid. But, in problems where the loss surface is less uniform, descents with high learning rates might overshoot or diverge at local minimas. Regularization, like LASSO or Ridge, counteract this by "shrinking" the effective solution space the gradient descent traverses.

In [140…

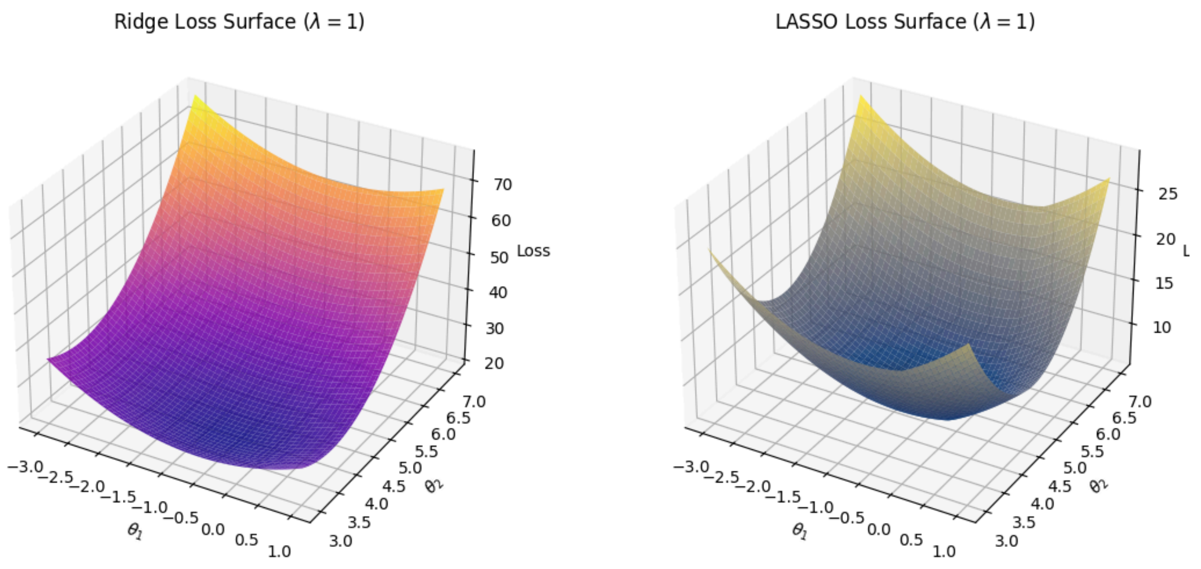
```python
import numpy as np
import matplotlib.pyplot as plt

theta_closed = np.linalg.inv(X.T @ X) @ X.T @ y

def mse_loss(theta0, theta1, theta2, X, y):
    theta = np.array([theta0, theta1, theta2])
    y_pred = X @ theta
    return np.mean((y - y_pred)**2)

t0 = theta_closed[0]
theta1_vals = np.linspace(theta_closed[1] - 2, theta_closed[1] + 2, 100)
theta2_vals = np.linspace(theta_closed[2] - 2, theta_closed[2] + 2, 100)

Theta1, Theta2 = np.meshgrid(theta1_vals, theta2_vals)
Loss = np.zeros_like(Theta1)

for i in range(Theta1.shape[0]):
    for j in range(Theta1.shape[1]):
        Loss[i, j] = mse_loss(t0, Theta1[i, j], Theta2[i, j], X, y)

# Plot loss surface
fig = plt.figure(figsize=(8,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(Theta1, Theta2, Loss, cmap='viridis', alpha=0.85, edgecolor='none')

ax.set_xlabel(r'$\theta_1$')
ax.set_ylabel(r'$\theta_2$')
ax.set_zlabel('MSE')
ax.set_title('OLS Loss Surface (fixed $\\theta_0$)')

plt.tight_layout()
plt.show()
```



In [141…

```
import numpy as np
import matplotlib.pyplot as plt

def ridge_loss(theta0, theta1, theta2, X, y, lam=1.0):
    return mse_loss(theta0, theta1, theta2, X, y) + lam * (theta1**2 + theta2**2)

def lasso_loss(theta0, theta1, theta2, X, y, lam=1.0):
    return mse_loss(theta0, theta1, theta2, X, y) + lam * (np.abs(theta1) + np.abs(

Loss_ridge = np.zeros_like(Theta1)
Loss_lasso = np.zeros_like(Theta1)
lam = 1.0

for i in range(Theta1.shape[0]):
    for j in range(Theta1.shape[1]):
        Loss_ridge[i, j] = ridge_loss(t0, Theta1[i, j], Theta2[i, j], X, y, lam=lam
        Loss_lasso[i, j] = lasso_loss(t0, Theta1[i, j], Theta2[i, j], X, y, lam=lam

fig = plt.figure(figsize=(12,5))
```

```
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(Theta1, Theta2, Loss_ridge, cmap='plasma', alpha=0.85, edgecolor='
ax1.set_title('Ridge Loss Surface ($\\lambda=1$)')
ax1.set_xlabel(r'$\theta_1$')
ax1.set_ylabel(r'$\theta_2$')
ax1.set_zlabel('Loss')

ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(Theta1, Theta2, Loss_lasso, cmap='cividis', alpha=0.85, edgecolor=
ax2.set_title('LASSO Loss Surface ($\\lambda=1$)')
ax2.set_xlabel(r'$\theta_1$')
ax2.set_ylabel(r'$\theta_2$')
ax2.set_zlabel('Loss')

plt.tight_layout()
plt.show()
```



As we can see, ridge steepens the loss surface, pulling coefficients towards the smaller values. LASSO, on the other hand, creates sharp "edges" around the the axes.

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?

In [142…

```
eta = 0.1
num_iters = 1000
tol = 1e-6
lam = 0.01

theta_ridge = np.zeros(n_features)

for t in range(num_iters):
    theta_old = theta_ridge.copy()
    grad_Ridge = gradient_ridge(X_norm, y_centered, theta_ridge, lam)
    theta_ridge = theta_ridge - eta * grad_Ridge

    if np.linalg.norm(theta_ridge - theta_old) < tol:
        print(f"Success after {t+1} iterations")
        break

print("Gradient Descent Ridge coefficients:", theta_ridge)
print("Closed-form Ridge coefficients:", theta_closed_formRidge)
```

```
Success after 125 iterations
Gradient Descent Ridge coefficients: [ 0.         -1.14536074  5.96103218]
Closed-form Ridge coefficients: [ 0.         -1.16815283  6.07965364]
```

I define the stopping criterion based on the change in the parameters between iterations. If the L2 difference falls under a certain threshold, $1e-16$, I stop the algorthim and print As we can see from the prints, the closed forms and gradient descent are trivially close.

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.

In [143…

```
np.random.seed(0)

n_samples = 100
n_features = 10

theta_true = np.array([5.0, -3.0, 0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 0.0])

X_synth = np.random.randn(n_samples, n_features)
noise = 0.5 * np.random.randn(n_samples)
y_synth = X_synth @ theta_true + noise

X_synth_norm = (X_synth - X_synth.mean(axis=0)) / X_synth.std(axis=0)
y_synth_centered = y_synth - y_synth.mean()
```

In [144…

```
lam = 0.1
I_synth = np.eye(n_features)

theta_ols_analytical = np.linalg.pinv(X_synth_norm) @ y_synth_centered
theta_ridge_analytical = np.linalg.inv(X_synth_norm.T @ X_synth_norm + lam * I_synt

eta = 0.01
num_iters = 1000

theta_ols_gd = np.zeros(n_features)
for t in range(num_iters):
    grad = gradient_ols(X_synth_norm, y_synth_centered, theta_ols_gd)
    theta_ols_gd = theta_ols_gd - eta * grad
```

```
theta_ridge_gd = np.zeros(n_features)
for t in range(num_iters):
    grad = gradient_ridge(X_synth_norm, y_synth_centered, theta_ridge_gd, lam)
    theta_ridge_gd = theta_ridge_gd - eta * grad

print("True coefficients:", theta_true)
print("OLS Analytical:", theta_ols_analytical)
print("OLS Gradient Descent:", theta_ols_gd)
print("Ridge Analytical:", theta_ridge_analytical)
print("Ridge Gradient Descent:", theta_ridge_gd)
```

```
True coefficients: [ 5. -3.  0.  0.  0.  0.  2.  0.  0.  0.]
OLS Analytical: [ 5.03241281e+00 -2.89258175e+00 -1.55189951e-02  1.51795012e-01
 -6.83299260e-02 -4.40147965e-02  1.76999871e+00  4.37643569e-03
  4.52550260e-02 -4.97610000e-02]
OLS Gradient Descent: [ 5.03052627e+00 -2.88962545e+00 -1.54631907e-02  1.53537540e-
01
 -7.12034281e-02 -4.55513413e-02  1.76973981e+00  4.18611669e-03
  4.31429304e-02 -4.94513872e-02]
Ridge Analytical: [ 5.02716096e+00 -2.88896347e+00 -1.55169486e-02  1.52287662e-01
 -6.93302234e-02 -4.47229113e-02  1.76836298e+00  4.62139092e-03
  4.41380481e-02 -4.96664608e-02]
Ridge Gradient Descent: [ 4.17145961 -2.3219415  -0.01540697  0.20874048 -0.1946188
-0.13721978
  1.49447228  0.04064028 -0.10520121 -0.03682043]
```

Analysis:

Ridge regression performs better than OLS for this sparse dataset. Ridge shrinks the coefficients toward zero, which helps when many features are irrelevant (have true coefficient = 0). The regularization penalty helps Ridge identify the important features (0, 1, and 6) while suppressing noise in the irrelevant features. OLS tends to overfit and assigns non-zero weights to irrelevant features due to noise.