Notebook_W37

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.

1.3 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.

[84]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.metrics import mean_squared_error
```

[85]:
```
def polynomial_features(x, p, intercept=False):
    n = len(x)

    # Only include the intercept column if the itnercept argumenSt is True
    if intercept:
        X = np.zeros((n, p + 1))

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

    else:
        X = np.zeros((n, p))

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

    return X
```

[86]:
```
# Setting seed to get the same data every time
np.random.seed(4155)

# Generate the data
n = 100
x = np.linspace(-2, 2, n)
y = 2 - x + 5*x**2 + np.random.randn(n)
```

[87]:
```
X = polynomial_features(x, 2)
```

[88]:
```
# Standardize features (zero mean, unit variance for each feature)
X_mean = X.mean(axis=0)
X_std = X.std(axis=0)
X_std[X_std == 0] = 1  # safeguard to avoid division by zero for constant␣
 ↪features
```

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

# Center the target to zero mean (optional, to simplify intercept handling)
y_mean = y.mean(axis=0)
y_centered = (y - y_mean).reshape(-1, 1)

print(y_centered.shape)
```

```
(100, 1)
```

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).

1.4 Exercise 2, calculate the gradients

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

```
[89]: def gradient_decent_OLS(X, y):

          # Extract number of datapoints and number of parameters from the design␣
      ↪matrix
          n, p = X.shape

          # Hessian matrix
          H = (2.0/n)* X.T @ X

          # Find Eigenvalues and vectors
          EigVal, EigVec = np.linalg.eig(H)

          # Finding learningrate based on eigenvalues
          learning_rate = 1 / max(EigVal)

          # Selecting a random theta to begin the gradient decent
          theta = np.random.randn(p,1)

          # initializing a "previous cost" variable to infinity
          prev_cost = float("inf")

          for i in range(100):

              # Gradient
              gradient = ((2/n) * X.T) @ (X @ theta - y)

              # Updating theta
              theta -= learning_rate * gradient
```

```
        # Predicting y
        y_pred = X @ theta

        # Calculating cost
        cost = mean_squared_error(y, y_pred)

        print(prev_cost - cost)

        if (prev_cost - cost) < 0.001:
            print(f"Cost reduced below 0.001 in {i} iterations")
            break

        # Update previous cost
        prev_cost = cost

    return gradient


gradient_decent_OLS(X, y_centered)
```

```
inf
0.03329407729891187
0.011649843172744312
0.004076366037441659
0.0014263505375105012
0.0004990905716368843
Cost reduced below 0.001 in 5 iterations
```

```
[89]: array([[6.46315419e-02],
             [3.33066907e-16]])
```

```
[90]: def gradient_decent_ridge(X, y, Lambda):

          # Extract number of datapoints and number of parameters from the design␣
       ↪matrix
          n, p = X.shape

          # Hessian matrix
          H = (2.0/n)* X.T @ X + 2*Lambda * np.eye(p)

          # Find Eigenvalues and vectors
          EigVal, EigVec = np.linalg.eig(H)

          # Finding learningrate based on eigenvalues
          learning_rate = 1 / max(EigVal)

          # Selecting a random theta to begin the gradient decent
          theta = np.random.randn(p,1)
```

```
    # initializing a "previous cost" variable to infinity
    prev_cost = float("inf")

    for i in range(100):

        # Gradient
        gradient = 2 * ((1/n) * X.T @ (X @ theta - y) + Lambda * theta)

        # Updating theta
        theta -= learning_rate * gradient

        # Predicting y
        y_pred = X @ theta

        # Calculating cost
        cost = mean_squared_error(y, y_pred)

        print(prev_cost - cost)

        if (prev_cost - cost) < 0.001:
            print(f"Cost reduced below 0.001 in {i} iterations")
            break

        # Update previous cost
        prev_cost = cost


    return gradient


gradient_decent_ridge(X, y_centered, 0.01)
```

```
inf
0.016015642701432853
0.005270283369345208
0.0016560380974119937
0.0004715802407062597
Cost reduced below 0.001 in 4 iterations
```

```
[90]: array([[-7.84037702e-02],
             [ 6.73072709e-16]])
```

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

[91]:
```python
# 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

n_features = 2


# 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(n_features)
theta_closed_formRidge = np.linalg.inv(X.T @ X) @ X.T @ y
theta_closed_formOLS = np.linalg.inv(X.T @ X + n * lam * I) @ X.T @ y

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

```
Closed-form Ridge coefficients: [-1.09228307  5.80432686]
Closed-form OLS coefficients: [-1.08431177  5.78694944]
```

This computes the Ridge and OLS regression coefficients directly. The identity matrix I has the same size as X^TX. It adds λ to the diagonal of X^TX for Ridge regression. We then invert this matrix and multiply by X^Ty. 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.

1.5.2 3b)

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

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:

```
[92]: # Setting seed to get the same data every time
np.random.seed(4155)

# Generate the data
n = 100
x = np.linspace(-2, 2, n)
y = 2 - x + 5*x**2 + x**3 + np.random.randn(n)

# Standardize features (zero mean, unit variance for each feature)
X_mean = X.mean(axis=0)
X_std = X.std(axis=0)
X_std[X_std == 0] = 1  # safeguard to avoid division by zero for constant␣
 ↪features
X_norm = (X - X_mean) / X_std

# Center the target to zero mean (optional, to simplify intercept handling)
y_mean = y.mean(axis=0)
y_centered = (y - y_mean).reshape(-1, 1)
```

```
[93]: def gradient_decent_OLS(X, y):

    y = y.reshape(-1, 1)

    # Extract number of datapoints and number of parameters from the design␣
 ↪matrix
    n, p = X.shape

    # Hessian matrix
    H = (2.0/n)* X.T @ X

    # Find Eigenvalues and vectors
    EigVal, EigVec = np.linalg.eig(H)

    # Finding learningrate based on eigenvalues
    learning_rate = 1 / max(EigVal)

    # Selecting a random theta to begin the gradient decent
    theta = np.random.randn(p,1)

    # Copy theta
    prev_theta = theta.copy()

    for i in range(100):

        # Gradient
        gradient = ((2/n) * X.T) @ (X @ theta - y)
        # Updating theta
        theta -= learning_rate * gradient

        # Predicting y
        y_pred = X @ theta

        # Calculating cost
        cost = mean_squared_error(y, y_pred)

        #print(prev_theta - theta)

        if (np.linalg.norm(prev_theta - theta)) < 1e-6:
            print(f"Theta reduced below 1e-6 in {i} iterations")
            break

        # Update previous cost
        prev_theta = theta.copy()

    return theta


gradient_decent_OLS(X, y_centered)
```

```
Theta reduced below 1e-6 in 27 iterations
```

[93]:
```
array([[1.35587441],
       [2.24340581]])
```

[94]:
```
def gradient_decent_ridge(X, y, Lambda=0.01):

    y = y.reshape(-1, 1)

    # Extract number of datapoints and number of parameters from the design␣
 ↪matrix
    n, p = X.shape

    # Hessian matrix
    H = (2.0/n)* X.T @ X + 2*Lambda * np.eye(p)

    # Find Eigenvalues and vectors
    EigVal, EigVec = np.linalg.eig(H)

    # Finding learningrate based on eigenvalues
    learning_rate = 1 / max(EigVal)

    # Selecting a random theta to begin the gradient decent
    theta = np.random.randn(p,1)
    # Copy theta
    prev_theta = theta.copy()

    for i in range(100):

        # Gradient
        # Gradient
        gradient = 2 * ((1/n) * X.T @ (X @ theta - y) + Lambda * theta)

        # Updating theta
        theta -= learning_rate * gradient

        # Predicting y
        y_pred = X @ theta

        # Calculating cost
        cost = mean_squared_error(y, y_pred)

        #print(prev_theta - theta)

        if (np.linalg.norm(prev_theta - theta)) < 1e-6:
            print(f"Theta reduced below 1e-6 in {i} iterations")
            break

        # Update previous cost
        prev_theta = theta.copy()

    return theta


gradient_decent_ridge(X, y_centered)
```

```
Theta reduced below 1e-6 in 27 iterations
```

```
[94]: array([[1.34597932],
             [2.23668934]])
```

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

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?

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:

[95]:
```
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
```

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?

[96]:
```python
theta_OLS = gradient_decent_OLS(X, y)

print(theta_OLS)

theta_ridge = gradient_decent_ridge(X, y)

print(theta_ridge)
```

```
Theta reduced below 1e-6 in 27 iterations
[[ 5.00905312e+00]
 [-3.00383291e+00]
 [-1.62717147e-02]
 [ 1.44820054e-01]
 [-7.16009828e-02]
 [-4.29659936e-02]
 [ 2.05558065e+00]
 [ 1.97577329e-03]
 [ 4.11921650e-02]
 [-5.10225476e-02]]
Theta reduced below 1e-6 in 25 iterations
[[ 4.96007550e+00]
 [-2.96287954e+00]
 [-1.66160582e-02]
 [ 1.51234729e-01]
 [-8.08032029e-02]
 [-5.09072683e-02]
 [ 2.02863280e+00]
 [ 4.83606626e-03]
 [ 3.16609656e-02]
 [-5.16769074e-02]]
```

Theta from OLS is slightly closer to the true coefficients than ridge, however, both are very close