Exercises week 37

Implementing gradient descent for Ridge and ordinary Least Squares Regression

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

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

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.

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.

1a)

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}$andthedesignmatrix$\boldsymbol{X}$.

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

n_features = 100
x = np.linspace(-2, 2, n_features)
y = (2 - x + 5*x**2) + np.random.normal(0, 0.2, size=x.shape)

# Create design matrix using a loop for polynomial degree p
def polynomial_features(x, p):
    n = len(x)
    X = np.zeros((n, p + 1))
    for i in range(p + 1):
        X[:, i] = x**i
    return X

polynomials = 2
X = polynomial_features(x, polynomials)

# 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 featur
X_norm = (X - X_mean) / X_std


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

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

After this preprocessing, each column of $\boldsymbol{X}_{\mathrm{norm}}$ has mean zero and standard deviation 1 and $\boldsymbol{y}_{\mathrm{centered}}$ 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).

-----> Answer When X and y are centered, intercept are decoupled from slope estimation. Fitting can be done without intercept which can be beneficial since intercept estimation can impact other parameteres negatively if intercept values are very large compared to features. Intercept can be calculated with: intercept_ y_scaler - X_train_mean @ Theta

Exercise 2, calculate the gradients

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

In [2]:
```
"""
I'm not sure what you want me to do here...
"""
```

Out[2]:
```
"\nI'm not sure what you want me to do here...\n"
```

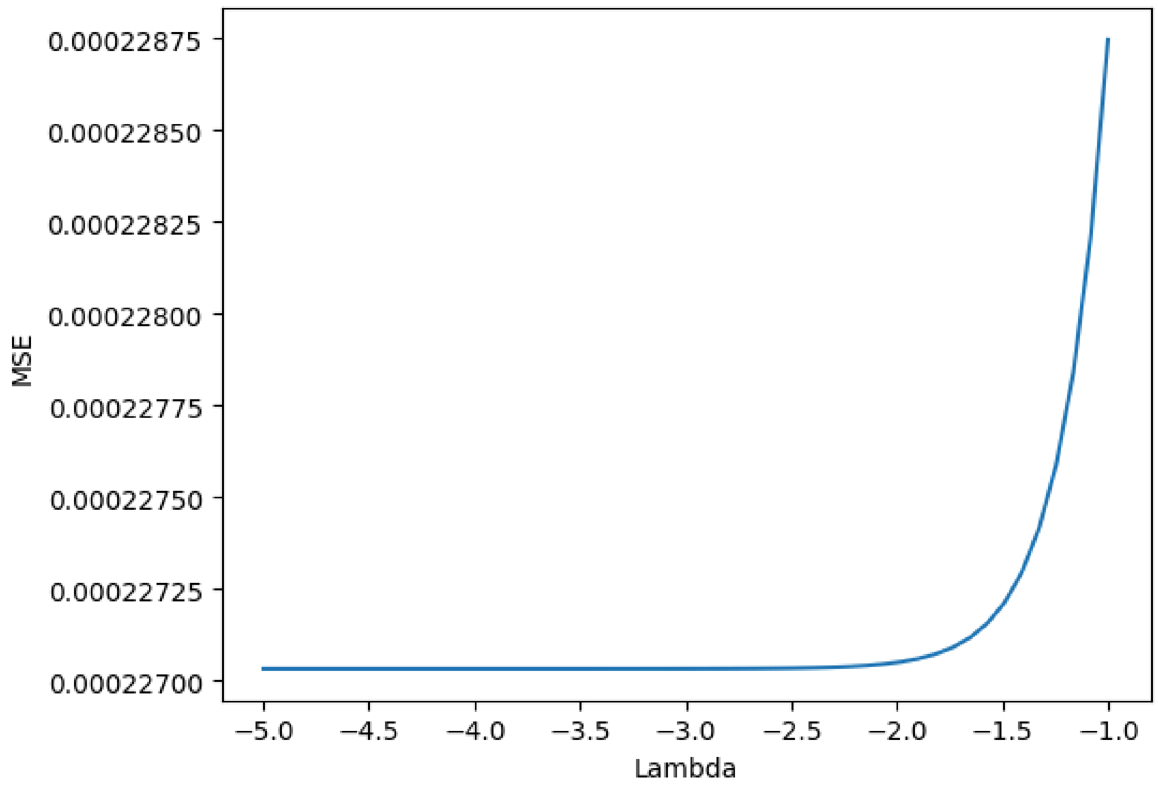
Exercise 3, using the analytical formulae for OLS and Ridge regression to find the optimal paramters θ

In [3]:
```
# 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 sma
#lam = ?

# 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
import matplotlib.pyplot as plt

def MSE(y ,y_tilde):
    n = np.size(y_tilde)
    return np.sum((y - y_tilde)**2)/n

theta_ols = np.linalg.inv(X.T @ X) @ (X.T @ y)
lambda_ = 0.001
I = np.eye(X.shape[1])
theta_ridge = np.linalg.solve(X.T @ X + lambda_ * I, X.T @ y)

y_tilde_ols = X @ theta_ols
y_tilde_ridge = X @ theta_ridge
MSE_ols = MSE(y, y_tilde_ols)
MSE_ridge = MSE(y, y_tilde_ridge)

print(f"Optimal parameters for OLS: \n{theta_ols}")
print(f"MSE_ols: {MSE_ols}")
print(f"Optimal parameteres for Ridge: \n{theta_ridge}")
print(f"MSE ridge: {MSE_ridge}")

# For different values of hyperparameter lambda
lambdas = np.logspace(-1, -5, num=50)
x = np.linspace(0, 1, 100)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2) + np.random.normal(0, 0.1)

MSE_array = []

for lambda_ in lambdas:
    theta_ridge = np.linalg.solve(X.T @ X + lambda_ * I, X.T @ y)
    y_tilde_ridge = X @ theta_ridge
    MSE_ridge = MSE(y, y_tilde_ridge)
    MSE_array.append(MSE_ridge)

plt.plot(np.log10(lambdas), MSE_array)
```

```python
plt.xlabel('Lambda')
plt.ylabel('MSE')
```

```
Optimal parameters for OLS:
[ 1.96383276 -0.99669115  5.01756974]
MSE_ols: 0.0517552504374583
Optimal parameteres for Ridge:
[ 1.96383469 -0.99668383  5.01755389]
MSE ridge: 0.051755251268134905
```

```
Out[3]: Text(0, 0.5, 'MSE')
```



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 $\boldsymbol{\theta}$ is a NumPy array of shape (n_features,) containing the fitted parameters $\boldsymbol{\theta}$.

3a)

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

3b)

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

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 (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:

In [4]:
```
# Gradient descent parameters, learning rate eta first
eta = 0.1
# Then number of iterations
num_iters = 1000
# tolerance stopping criterion
tolerance = 0.001

# Initialize weights for gradient descent
#theta = np.zeros(n_features)

# Ridge regression setup
XT_X = X.T @ X
Id = n_features * lambda_ * np.eye(XT_X.shape[0])

theta_linreg = np.linalg.inv(XT_X + Id) @ X.T @ y
print(f"Theta analytical: \n{theta_linreg}\n")

theta_ridge = np.zeros(polynomials + 1)
theta_ridge = theta_ridge.reshape(-1, 1)
theta_ols = theta_ridge.copy()
Niterations = 100

# Reshape y for matrix operations
y = y.reshape(-1, 1)

# Store theta[0] over iterations for visualization
theta_list = []
# Gradient descent loop
for t in range(Niterations):
    gradients_ols = (2.0 / n_features) * X.T @ (X @ theta_ols - y)
    theta_ols -= eta * gradients_ols

    gradients_ridge = (2.0 / n_features) * X.T @ (X @ theta_ridge - y) + 2 * lam
    theta_ridge -= eta * gradients_ridge

    """
    grad_OLS = ?
    grad_Ridge = ?
    # Update parameters theta
    theta_gdOLS = ?
    theta_gdRidge = ?
    """

    #stopping criterion
    theta_list.append(theta_ridge.copy())
    if t > 0 and np.linalg.norm(theta_list[-1] - theta_list[-2]) < tolerance:
        print(f"Converged after {t} iterations.")
        break
```

```
print("OLS theta:\n", theta_ols)
print("Ridge theta:\n", theta_ridge)
```

```
Theta analytical:
[ 0.87886489 -0.04502659  0.00931954]

Converged after 53 iterations.
OLS theta:
 [[ 0.86805005]
 [-0.04502692]
 [ 0.01429712]]
Ridge theta:
 [[ 0.86803191]
 [-0.04502658]
 [ 0.01430442]]
```

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

-----> Answer: Too small eta results in slow converging. Too large eta results in not converging.

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?

-----> Answer: With eta = 0.1 convergence criteria is met after 53 iterations. By using np.linalg.norm the distance between the previous θ and new θ are checked

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 $\hat{\boldsymbol{\theta}} = [5.0, -3.0, 0.0, 0.0, 0.0, 0.0, 2.0, 0.0, 0.0, 0.0]$, meaning only features 0, 1, and 6 have a real effect on y.

Then we sample feature values for $\boldsymbol{X}$ 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 $\boldsymbol{X}\hat{\boldsymbol{\theta}}$ and add some noise (to simulate measurement error or unexplained variance).

Below is the code to generate the dataset:

In [5]:
```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 n
noise = 0.5 * np.random.randn(n_samples)    # Gaussian noise
y = X @ theta_true + noise
y = y.reshape(-1,1)
```

This code produces a dataset where only features 0, 1, and 6 significantly influence $\boldsymbol{y}$. 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?

------> Answer: Ridge regression should be better since the datasett is sparse and should generalize better

In [6]:
```python
# Gradient descent parameters, learning rate eta first
eta = 0.01
# Then number of iterations
num_iters = 1000
# tolerance stopping criterion
tolerance = 0.001

# Initialize weights for gradient descent
#theta = np.zeros(n_features)
```

```python
# Ridge regression setup
XT_X = X.T @ X
Id = n_features * lambda_ * np.eye(XT_X.shape[0])

theta_linreg = np.linalg.inv(XT_X + Id) @ X.T @ y
print(f"Theta analytical: \n{theta_linreg.T}\n")

theta_ridge = np.zeros(len(theta_true))
theta_ridge = theta_ridge.reshape(-1, 1)
theta_ols = theta_ridge.copy()
Niterations = 100

# Reshape y for matrix operations
y = y.reshape(-1, 1)

# Store theta[0] over iterations for visualization
theta_list = []
# Gradient descent loop
for t in range(Niterations):
    gradients_ols = (2.0 / n_features) * X.T @ (X @ theta_ols - y)
    theta_ols -= eta * gradients_ols

    gradients_ridge = (2.0 / n_features) * X.T @ (X @ theta_ridge - y) + 2 * lam
    theta_ridge -= eta * gradients_ridge

    #stopping criterion
    theta_list.append(theta_ridge.copy())
    if t > 0 and np.linalg.norm(theta_list[-1] - theta_list[-2]) < tolerance:
        print(f"Converged after {t} iterations.")
        break

print("OLS theta:\n", theta_ols.T)
print("Ridge theta:\n", theta_ridge.T)

difference_ols = np.linalg.norm(theta_linreg - theta_ols)
print(f"Difference ols analytical vs gradient OLS: {difference_ols}")
difference_ridge = np.linalg.norm(theta_linreg - theta_ridge)
print(f"Difference ols analytical vs gradient ridge: {difference_ridge}")
```

```
Theta analytical:
[[ 5.00904823e+00 -3.00382922e+00 -1.62718642e-02  1.44820478e-01
  -7.16015964e-02 -4.29664522e-02  2.05557843e+00  1.97612738e-03
   4.11912509e-02 -5.10225846e-02]]

Converged after 46 iterations.
OLS theta:
 [[ 5.00788101e+00 -2.99964463e+00 -1.59558467e-02  1.46999432e-01
  -7.47942756e-02 -4.54716560e-02  2.05245102e+00  1.61639608e-03
   4.00001058e-02 -5.09587962e-02]]
Ridge theta:
 [[ 5.00787608e+00 -2.99964052e+00 -1.59558854e-02  1.47000065e-01
  -7.47951817e-02 -4.54724391e-02  2.05244832e+00  1.61669072e-03
   3.99991477e-02 -5.09588639e-02]]
Difference ols analytical vs gradient OLS: 0.007177964289374318
Difference ols analytical vs gradient ridge: 0.0071833454844783175
Converged after 46 iterations.
OLS theta:
 [[ 5.00788101e+00 -2.99964463e+00 -1.59558467e-02  1.46999432e-01
  -7.47942756e-02 -4.54716560e-02  2.05245102e+00  1.61639608e-03
   4.00001058e-02 -5.09587962e-02]]
Ridge theta:
 [[ 5.00787608e+00 -2.99964052e+00 -1.59558854e-02  1.47000065e-01
  -7.47951817e-02 -4.54724391e-02  2.05244832e+00  1.61669072e-03
   3.99991477e-02 -5.09588639e-02]]
Difference ols analytical vs gradient OLS: 0.007177964289374318
Difference ols analytical vs gradient ridge: 0.0071833454844783175
```