Lab2 Optimization

April 28, 2024

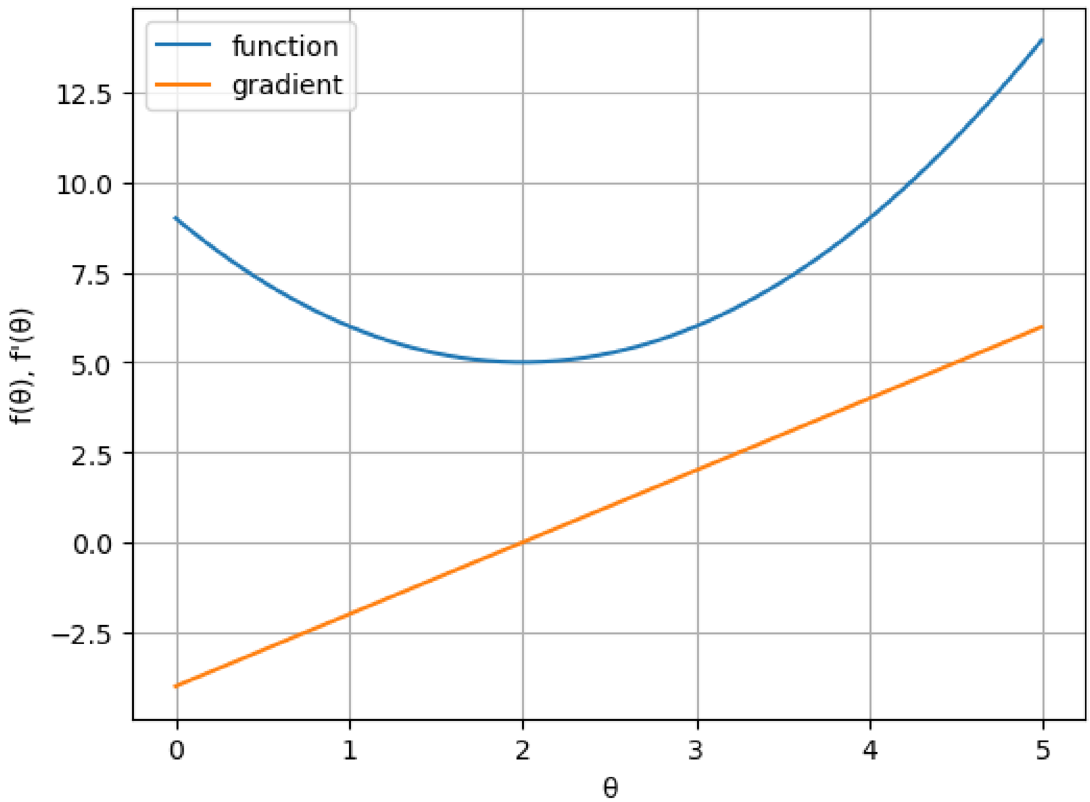
Simon Merten, Matriculation Number: 7375404
Leon Kvas, Matriculation Number: 7371206
Marius Meise, Matriculation Number: 7369546

1 Gradient Descent

In this exercise, we use gradient descent to find the minimum of: $f(\theta) = (\theta - 2)^2 + 5$

1.1 What's the gradient of our function f? Define a gradient function g and plot it.

[1]:
```
import warnings
import numpy as np
warnings.filterwarnings("ignore", category=RuntimeWarning)
import numpy as np
import matplotlib.pyplot as plt

def function(theta):
    f = (theta - 2)**2 + 5
    return f

def gradient(theta):
    f = 2 * theta - 4
    return f

xs = np.arange(0, 5, 0.01)
ys_function = function(xs)
ys_gradient = gradient(xs)

plt.plot(xs, ys_function, label='function')
plt.plot(xs, ys_gradient, label="gradient")

plt.xlabel(' ')
plt.ylabel('f( ), f\'( )')
plt.legend()
plt.grid(True)
plt.show()
```



1.2 Assume a constant learning rate of $\quad = .8$. Write down the general update step for gradient descent.

General: $\theta_{t+1} = \theta_t - \lambda f'(\theta_t)$

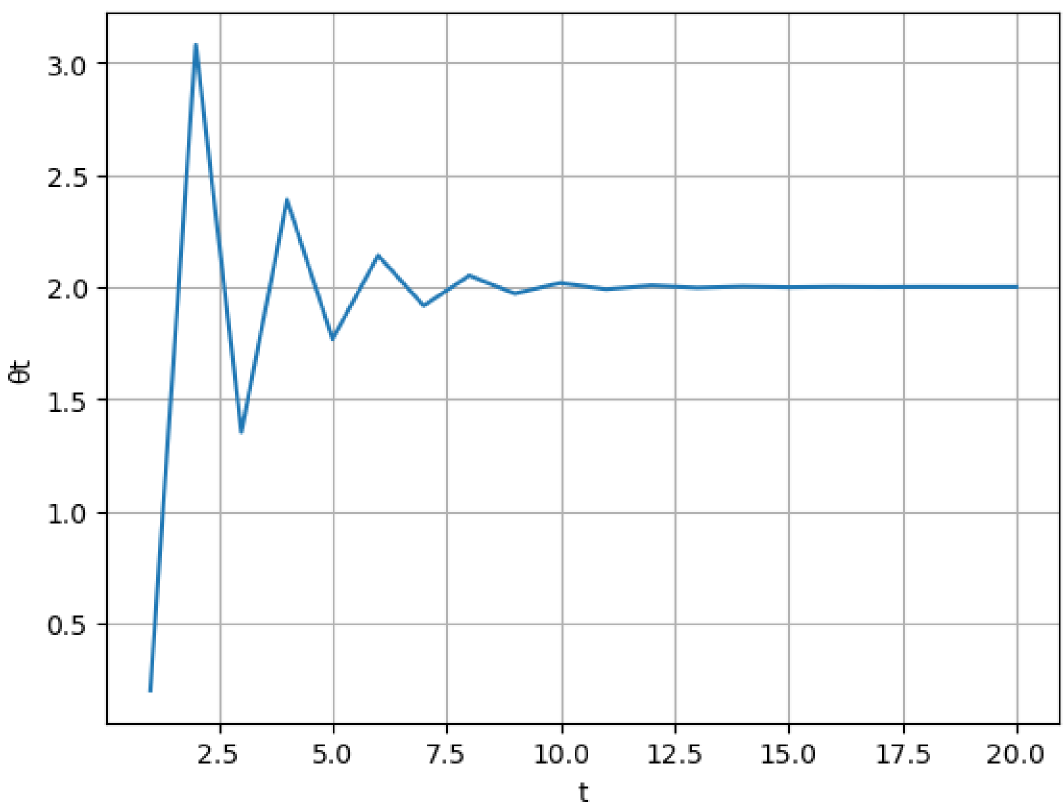
Example: $\theta_{t+1} = \theta_t - 0.8 \cdot (2\theta - 4)$

In every iteration, the coefficient theta is updated by subtracting the specified learning rate of 0.8 times the gradient at the point of theta. The learning rate determines the size of the steps taken toward the local/global minimum of the function. The gradient indicates the direction in which the step is taken. In the above example, if the gradient is negative, it suggests that the minimum lies towards the right, thus shifting the coefficient towards the right side along the x-axis, and vice versa.

1.3 Implement gradient descent for minimizing f making use of your defined gradient function g. Compute 20 iterations to find the θ that minimizes $f(\theta)$. Plot the sequence of $\theta_t s$ against the iteration t. Start with $\theta_0 = 5$.

[2]:
```
def gradient_descent(n_iter, init_theta, lr):

    t_list = []
    theta_t_list = []
```

```
    theta = init_theta

    for t in range(1,n_iter+1):
        theta = theta - lr * (2*theta - 4)

        t_list.append(t)
        theta_t_list.append(theta)

    return t_list, theta_t_list

xs, ys = gradient_descent(n_iter=20, init_theta=5, lr=0.8)

plt.plot(xs, ys)

plt.xlabel('t')
plt.ylabel(' t')
plt.grid(True)
plt.show()
```



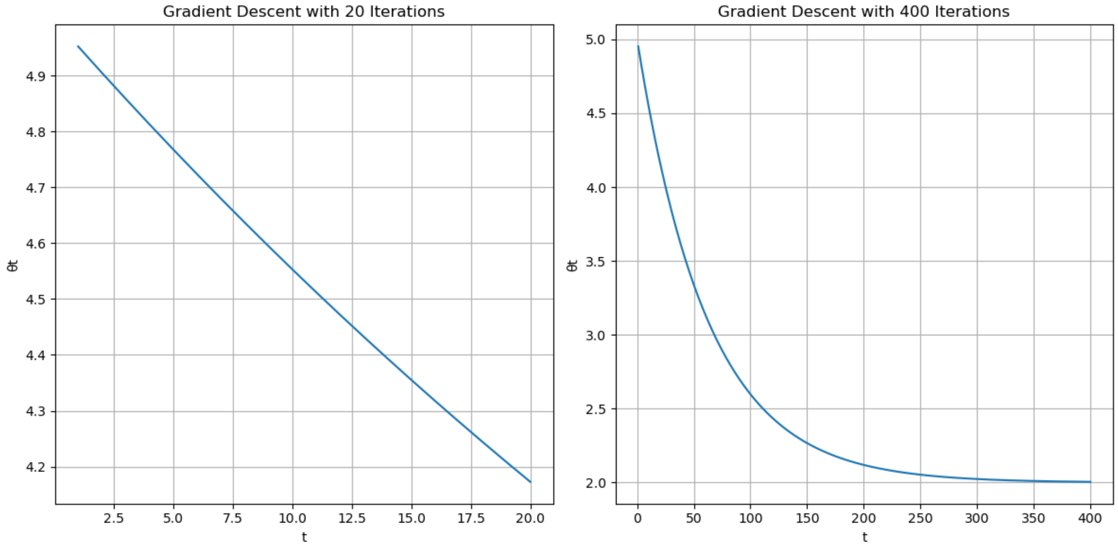
Interpretation: After approximately 10 iterations the algorithm finds the coefficient value that minimizes the given function.

1.4 Replace the analytical gradient by a two-sided numerical approximation. Repeat part 3 using the numerical gradient. Use a two-sided approximation such that:

$$\hat{g}(\theta) = \frac{f(\theta + h) - f(\theta - h)}{2h}$$

[3]:
```python
def function(theta):
    f = (theta - 2)**2 + 5
    return f

def gradient_descent(n_iter, theta, lr):

    t_list = []
    theta_t_list = []

    h = 0.1

    for t in range(1,n_iter+1):
        grad = ((function(theta+h)) - (function(theta-h))) / 2*h
        theta = theta - lr * grad
        t_list.append(t)
        theta_t_list.append(theta)

    return t_list, theta_t_list

x, y = gradient_descent(n_iter=20, theta=5, lr=0.8)
j, k = gradient_descent(n_iter=400, theta=5, lr=0.8)

fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Plotting x and y
axs[0].plot(x, y)
axs[0].set_xlabel('t')
axs[0].set_ylabel(' t')
axs[0].set_title('Gradient Descent with 20 Iterations')
axs[0].grid(True)

# Plotting k and j
axs[1].plot(j, k)
axs[1].set_xlabel('t')
axs[1].set_ylabel(' t')
axs[1].set_title('Gradient Descent with 400 Iterations')
axs[1].grid(True)

plt.tight_layout()
```

```
plt.show()
```



Interpretation: The two-sided numerical approximation requires significantly more iterations to converge to the coefficient at which the loss function reaches its minimum, in comparison to the analytical gradient in task 3. After 20 iterations, the estimated coefficient remains significantly distant from the optimal value of 2. Only after approximately 300 to 400 iterations does it finally reach the minimum and attain the optimal coefficient value.

2 Ordinary Least Squares

[4]:
```
import pandas as pd
data = pd.read_csv("Lab2_Optimization.csv", sep=";")
data.head()
```

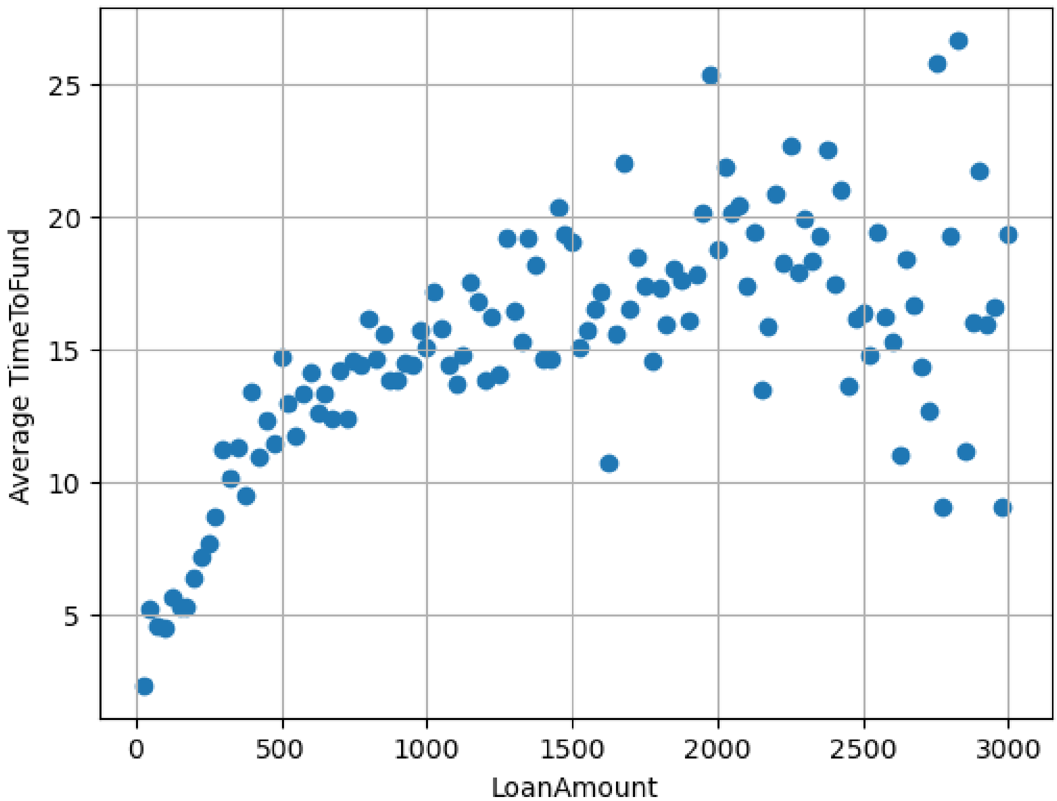
[4]:

	id	LoanAmount	TimeToFund
0	109570	575	0
1	111913	900	1
2	1457371	200	1
3	1470250	700	21
4	228017	450	3

2.1 Plot *LoanAmount* against *TimeToFund* to get a sense of the relationship between these two variables.

[5]:
```
# Data grouped by TimeToFund
grouped_data = data.groupby(data["LoanAmount"])["TimeToFund"].mean().
↪reset_index()
```

```python
plt.scatter(grouped_data["LoanAmount"], grouped_data["TimeToFund"])
plt.xlabel('LoanAmount')
plt.ylabel('Average TimeToFund')
plt.grid(True)
plt.show()
```

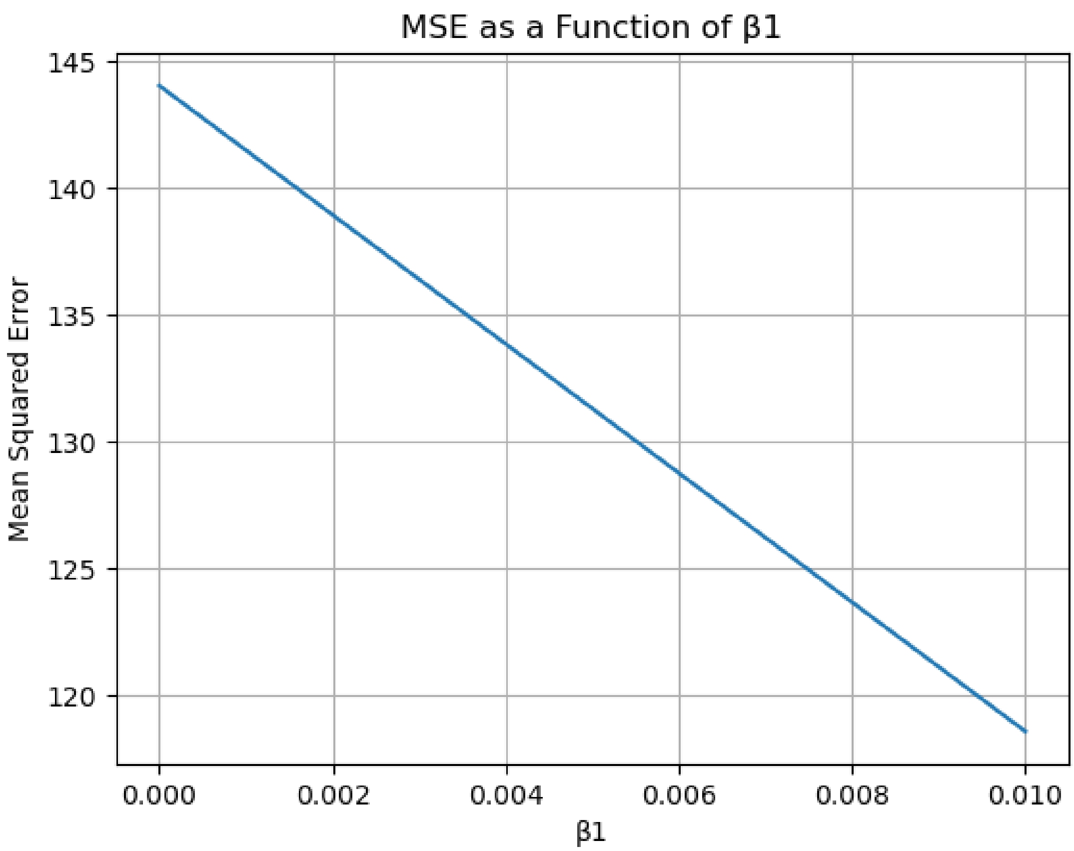


Interpretation: The plot indicates a positive relationship between LoanAmount and the average TimeToFund. Moreover, it reveals heteroskedasticity in their relationship.

2.2 Plot the objective function (average loss) as a function of $\beta 1 \in [0, .01]$, keeping $\beta 0$ fixed at 7.

[6]:
```python
# Define the objective function
def MSE(beta0, beta1, y, X):
    return np.sum((y - beta0 - beta1*X)**2) / len(y)

beta0 = 7
beta1 = [0, 0.01]
losses = []
X = data["LoanAmount"]
y = data["TimeToFund"]
```

```
for i in beta1:
    loss = MSE(beta0, i, y, X)
    losses.append(loss)

plt.plot(beta1, losses)
plt.xlabel(' 1')
plt.ylabel('Mean Squared Error')
plt.title('MSE as a Function of  1')
plt.grid(True)
plt.show()
```



Interpretation: The plot suggests that increasing the coefficient of , while keeping constant, results in a decrease in the mean squared error.

2.3 Use this gradient function to optimize the MSE via gradient descent, starting at $\beta 0 = 5$ and $\beta 1 = .005$. Use a learning rate of $\lambda = .0001$ and 1000 iterations. Why does the algorithm yield NaNs for $\beta 0$ and $\beta 1$?

[7]:
```
def gradient_OLS(beta0, beta1, y, X):
    error = beta0 + beta1 * X - y
    g0 = 2 * np.sum(error) / len(y)
    g1 = 2 * np.sum(error * X) / len(y)
    return np.array([g0, g1])
```

[8]:
```
def gradient_descent(beta0, beta1, y, X, lr, n_iter):
    losses = []
    for _ in range(n_iter):
        grad_beta0, grad_beta1 = gradient_OLS(beta0, beta1, y, X)
        beta0 -= lr * grad_beta0
        beta1 -= lr * grad_beta1
        loss = MSE(beta0, beta1, y, X)
        losses.append(loss)
    return beta0, beta1, losses
```

[9]:
```
beta0, beta1, losses = gradient_descent(5, 0.005, y, X, lr=0.0001, n_iter=1000)
print(beta0, beta1)
```

```
inf nan
```

Interpretation: The gradient descent algorithm yields NaN and INF values for the coefficients because the gradients grow exponentially larger. At a certain iteration, the gradient values become too high to calculate the coefficient values. This issue is typically caused by using a learning rate that is too high. With each iteration, the calculated coefficient value ‘jumps’ from one side of the loss function to the other. As iterations progress, these jumps become larger, resulting in infinite values for the gradients.

2.4 Does it help to change the learning rate?

[10]:
```
beta0, beta1, losses = gradient_descent(5, 1, y, X, lr=0.000001, n_iter=1000)
print(beta0, beta1)
```

```
5.001089747679826 0.009389088703329686
```

Interpretation: The issue encountered in task 3 can be addressed by decreasing the learning rate. This modification leads to smaller steps taken towards the minimum at each iteration, preventing the coefficient values from ‘jumping’ drastically from one side to the other on the loss function. Consequently, this prevents the gradients from increasing exponentially, as observed in task 3.

2.5 What happens when we express $LoanAmount$ in $1000USD$ terms rather than in raw dollar terms? Try a learning rate of $\lambda \in .1, .01$.

[11]:
```
data["LoanAmount"] = data["LoanAmount"] / 1000
X = data["LoanAmount"]
y = data["TimeToFund"]
```

[12]:
```
data.head()
```

[12]:

	id	LoanAmount	TimeToFund
0	109570	0.575	0
1	111913	0.900	1
2	1457371	0.200	1
3	1470250	0.700	21
4	228017	0.450	3

[13]:
```
# learning rate 0.1
beta0, beta1, losses = gradient_descent(5, 0.005, y, X, lr=0.1, n_iter=1000)
print(beta0, beta1)
```

```
7.707706374341195 6.5565406628600575
```

[14]:
```
# learning rate 0.01
beta0, beta1, losses = gradient_descent(5, 0.005, y, X, lr=0.01, n_iter=1000)
print(beta0, beta1)
```

```
7.804579019509307 6.4155973932512484
```

Interpretation: When expressing the loan amount in $1000 USD terms rather than raw dollar terms, it essentially means scaling the loan amount by a factor of 1000. This scaling can affect the convergence of the gradient descent algorithm. As a consequence, the learning rate can be set to a larger value as compared to task 3, because the the relatively small dollar values (in thousands), allow for larger adjustments without overshooting the optimal solution.