Computation in Data Science (Third Part) Homework 2

December 22, 2020

- Arthor: HO Ching Ru (R09946006, NTU Data Science Degree Program)
- Course: Computation in Data Science (MATH 5080), NTU 2020 Fall Semester
- Instructor: Tso-Jung Yen (Institute of Statistical Science, Academia Sinica)

1 Derivation of the Lipschitz constant for the logistic loss for regression estimation

Statement [**c**], $c = \frac{4}{\lambda_1}$ where λ_1 is the largest eigenvalue of the Gram matrix $n^{-1}\sum_{i=1}^{n}\mathbf{x}_i\mathbf{x}_i^T$, is true.

1.1 Reason

From the process of gradient descent:

$$\beta^{r+1} = \beta^r - c\nabla\mathcal{L}(\beta^r)$$

where c is the step size. Let $c = 1/L$ where L is the Lipschitz constant. If the step size $c = 1/L$ is small enough, the process of gradient descent will decrease $\mathcal{L}(\beta)$.

Assume $\mathcal{L}(\beta)$ is Lipschitz continuous, for any r and $r+1$, we have

$$||\nabla\mathcal{L}(\beta^r) - \nabla\mathcal{L}(\beta^{r+1})|| \leq L||\beta^r - \beta^{r+1}||$$

If $\beta^r = \beta^{r+1}$, then the Lipschitz continuity of the gradient is equivalent to

$$\nabla^2\mathcal{L}(\beta^r) = \mathbf{H}(\beta^r) \leq LI$$

where $\mathbf{H}(\cdot)$ denotes the Hessian matrix.

Therefore, the eigenvalues of the Hessian are bounded above by L. The minimum L is the maximum eigenvalue of the gram matrix.

---

2 Programming work

In this programming work we will build a gradient algorithm to find an estimate of regression coefficients in logistic regression model.

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

np.random.seed(5566)

##### PARAMETERS #####
n = 200
p = 9
max_iter = 10000

# generate data: true beta
beta_true = [-1, 1, -1, 1, -1, 1, -1, 1, -1, 1,]

# generate data: x, y
x = list()
y = list()
for i in range(n):
    tmpx = ([1] + list(np.random.normal(size = p)))
    tmpy = math.exp(np.dot(x[j], beta_true))/(1 + math.exp(np.dot(x[j],
 ↪beta_true)))
    x.append(tmpx)
    y.append(np.random.binomial(1, tmpy))
x = np.array(x)
y = np.array(y)

print("dimension of true beta:", np.shape(beta_true))
print("dimension of x:", np.shape(x))
print("dimension of y:", np.shape(y))
```

```
dimension of true beta: (10,)
dimension of x: (200, 10)
dimension of y: (200,)
```

From $f_i = \mathbf{x}_i^T \beta$,

$$\begin{aligned}\mathcal{L}(\beta) &= \frac{1}{n}\sum_{i=1}^{n}\left[-y_i f_i + \log(1+e^{f_i})\right] \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[-y_i \mathbf{x}_i^T\beta + \log(1+e^{\mathbf{x}_i^T\beta})\right]\end{aligned}$$

We can calculate the gradient:

$$\nabla\mathcal{L}(\beta) = \frac{1}{n}\sum_{i=1}^{N}\left[-y_i\mathbf{x}_i^T + \frac{\exp(\mathbf{x}_i^T\beta)}{1+\exp(\mathbf{x}_i^T\beta)}\cdot\mathbf{x}_i^T\right]$$

Therefore, the β updating function becomes:

$$\begin{aligned}\beta^{r+1} &= \beta^r - c\nabla\mathcal{L}(\beta^r) \\ &= \beta^r - c\cdot\left(\frac{1}{n}\sum_{i=1}^{N}\left[-y_i\mathbf{x}_i^T + \frac{\exp(\mathbf{x}_i^T\beta^r)}{1+\exp(\mathbf{x}_i^T\beta^r)}\cdot\mathbf{x}_i^T\right]\right)\end{aligned}$$

Finally, assume $\beta^0 = \mathbf{0}$.

[2]:

```
def gradient_descent(x, y, c, n, p, times):
    beta = np.zeros(shape = (10,))
    norm_ls = list()
    beta_ls = list()
    for j in range(times):
        grad = np.zeros(shape = (1+p,))
        for i in range(n):
            # grad_funct
            tmp = np.array(-y[i]*x[i] + x[i]*(math.exp(np.dot(x[i], beta))/(1 +␣
↪math.exp(np.dot(x[i], beta)))))
            grad = grad + tmp
        beta = beta - c * (grad/n)
        beta_ls.append(beta)
        norm_ls.append(np.linalg.norm(grad/n))
    return beta, beta_ls, norm_ls

# Calculating with different c.
c_1_beta, c_1_beta_ls, c_1_norm_ls = gradient_descent(x, y, 1, n, p, max_iter)
c_01_beta, c_01_beta_ls, c_01_norm_ls = gradient_descent(x, y, 0.1, n, p,␣
↪max_iter)
c_001_beta, c_001_beta_ls, c_001_norm_ls = gradient_descent(x, y, 0.01, n, p,␣
↪max_iter)
c_0001_beta, c_0001_beta_ls, c_0001_norm_ls = gradient_descent(x, y, 0.001, n,␣
↪p, max_iter)
c_00001_beta, c_00001_beta_ls, c_00001_norm_ls = gradient_descent(x, y, 0.0001,␣
↪n, p, max_iter)
```

List the estimated β, i.e., $\hat{\beta}$ with different c.

[3]:

```
print("when c=1, \hat{beta}:", c_1_beta)
print("when c=0.1, \hat{beta}:", c_01_beta)
print("when c=0.01, \hat{beta}:", c_001_beta)
print("when c=0.001, \hat{beta}:", c_0001_beta)
print("when c=0.0001, \hat{beta}:", c_0001_beta)
```

```
when c=1, \hat{beta}: [-0.72640148  0.78970027 -0.82271918  0.87046678
-0.61724436  0.84589305
 -1.01837049  0.69528805 -0.72401332  1.01070486]
```

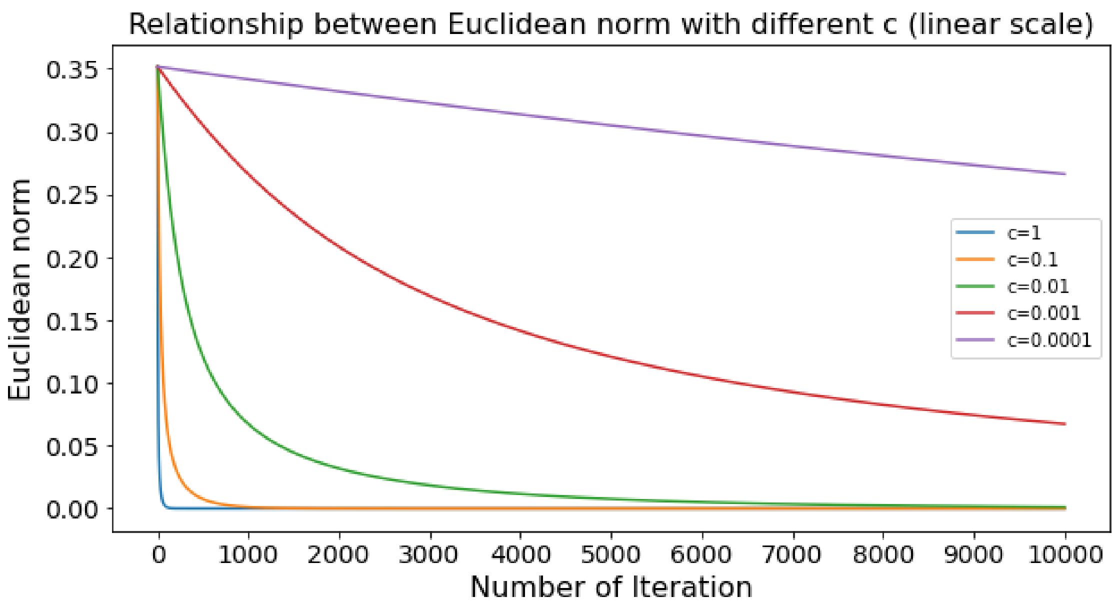
```
when c=0.1, \hat{beta}: [-0.72640148  0.78970027 -0.82271918  0.87046678
-0.61724436  0.84589305
 -1.01837049  0.69528805 -0.72401332  1.01070486]
when c=0.01, \hat{beta}: [-0.71562986  0.78002612 -0.81227073  0.85986662
-0.608811    0.83641202
 -1.00716505  0.68510478 -0.7159228   0.99492316]
when c=0.001, \hat{beta}: [-0.38297618  0.45260963 -0.46642624  0.5133647
-0.33603609  0.51373995
 -0.6200703   0.34056102 -0.40727183  0.50268503]
when c=0.0001, \hat{beta}: [-0.38297618  0.45260963 -0.46642624  0.5133647
-0.33603609  0.51373995
 -0.6200703   0.34056102 -0.40727183  0.50268503]
```

2.1 Plot 1

The x-axis is the number of iterations r and the y-axis is $||\nabla l(\beta^r)||_2$, the Euclidean norm of the gradient of the loss function $l(\beta^r)$ use in iterative scheme. In here, I plot c=1, c=0.1, c=0.01, c=0.001 and c=0.0001.

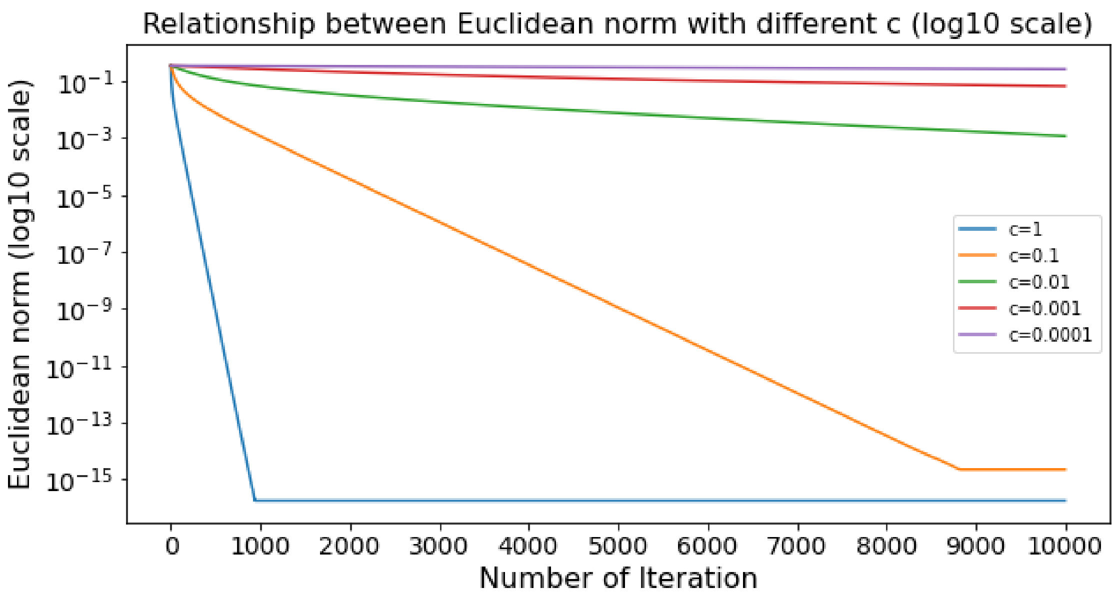
[5]:
```
import matplotlib.pyplot as plt

plt.figure(figsize = (10, 5))
plt.plot(list(range(max_iter)), c_1_norm_ls, label = "c=1")
plt.plot(list(range(max_iter)), c_01_norm_ls, label = "c=0.1")
plt.plot(list(range(max_iter)), c_001_norm_ls, label = "c=0.01")
plt.plot(list(range(max_iter)), c_0001_norm_ls, label = "c=0.001")
plt.plot(list(range(max_iter)), c_00001_norm_ls, label = "c=0.0001")
plt.legend()
plt.title("Relationship between Euclidean norm with different c (linear␣
 ↪scale)", fontsize = 16)
plt.xlabel("Number of Iteration",fontsize = 16)
plt.ylabel("Euclidean norm",fontsize = 16)
plt.xticks(np.arange(0, max_iter+1, max_iter/10), fontsize = 14)
plt.yticks(fontsize = 14)
plt.yscale("linear")
plt.show()
```



[6]:

```
plt.figure(figsize = (10, 5))
plt.plot(list(range(max_iter)), c_1_norm_ls, label = "c=1")
plt.plot(list(range(max_iter)), c_01_norm_ls, label = "c=0.1")
plt.plot(list(range(max_iter)), c_001_norm_ls, label = "c=0.01")
plt.plot(list(range(max_iter)), c_0001_norm_ls, label = "c=0.001")
plt.plot(list(range(max_iter)), c_00001_norm_ls, label = "c=0.0001")
plt.legend()
plt.title("Relationship between Euclidean norm with different c (log10 scale)",␣
 ↪fontsize = 16)
plt.xlabel("Number of Iteration", fontsize = 16)
plt.ylabel("Euclidean norm (log10 scale)", fontsize = 16)
plt.xticks(np.arange(0, max_iter+1, max_iter/10), fontsize = 14)
plt.yticks(fontsize = 14)
plt.yscale("log")
plt.show()
```



During 1000 iterations, only `c=1`, `c=0.1`, `c=0.01` will converge.

2.2 Plot 2

The x-axis is the number of iterations r and the y-axis is $l(\beta^r) - l(\beta^*)$, the difference between the loss functions evaluated at the current update β^r and the optimizer β^*. The optimizer β^* can be obtained from functions or software for carrying out logistic regression estimation available in your programming environment.

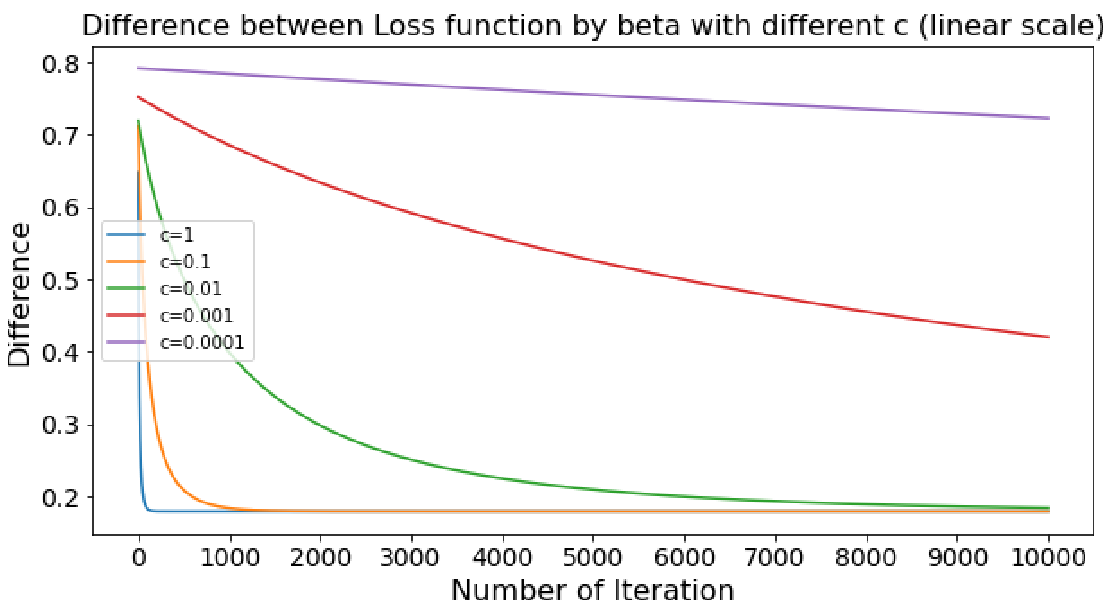
[7]:
```
def loss_funct(x, y, n, beta):
    loss = 0
    for i in range(n):
        tmp = np.array(-y[i]*np.dot(x[i], beta) + np.log(1 + np.exp(np.
↪dot(x[i], beta))))
    loss = loss + tmp
    return loss
```

[8]:
```
def diff_L(optimal_beta, beta_ls, n, times):
    diff_ls = list()
    optimal_loss = loss_funct(x, y, n, optimal_beta)
    for i in range(times):
        diff_ls.append(loss_funct(x, y, n, beta_ls[i] - optimal_loss))
    return diff_ls

c_1_diff_L = diff_L(c_1_beta, c_1_beta_ls, n, max_iter)
c_01_diff_L = diff_L(c_01_beta, c_01_beta_ls, n, max_iter)
c_001_diff_L = diff_L(c_001_beta, c_001_beta_ls, n, max_iter)
c_0001_diff_L = diff_L(c_0001_beta, c_0001_beta_ls, n, max_iter)
```

```
c_00001_diff_L = diff_L(c_00001_beta, c_00001_beta_ls, n, max_iter)
```

[9]:
```
plt.figure(figsize = (10, 5))
plt.plot(list(range(max_iter)), c_1_diff_L, label = "c=1")
plt.plot(list(range(max_iter)), c_01_diff_L, label = "c=0.1")
plt.plot(list(range(max_iter)), c_001_diff_L, label = "c=0.01")
plt.plot(list(range(max_iter)), c_0001_diff_L, label = "c=0.001")
plt.plot(list(range(max_iter)), c_00001_diff_L, label = "c=0.0001")
plt.legend()
plt.title("Difference between Loss function by beta with different c (linear␣
 ↪scale)", fontsize = 16)
plt.xlabel("Number of Iteration", fontsize = 16)
plt.ylabel("Difference", fontsize = 16)
plt.xticks(np.arange(0, max_iter+1, max_iter/10),fontsize = 14)
plt.yticks(fontsize = 14)
plt.yscale("linear")
plt.show()
```



[10]:
```
plt.figure(figsize = (10, 5))
plt.plot(list(range(max_iter)), c_1_diff_L, label = "c=1")
plt.plot(list(range(max_iter)), c_01_diff_L, label = "c=0.1")
plt.plot(list(range(max_iter)), c_001_diff_L, label = "c=0.01")
plt.plot(list(range(max_iter)), c_0001_diff_L, label = "c=0.001")
plt.plot(list(range(max_iter)), c_00001_diff_L, label = "c=0.0001")
plt.legend()
plt.title("Difference between Loss function by beta with different c (log10␣
 ↪scale)", fontsize = 16)
```

```
plt.xlabel("Number of Iteration",fontsize = 16)
plt.ylabel("Difference",fontsize = 16)
plt.xticks(np.arange(0, max_iter+1, max_iter/10), fontsize = 14)
plt.yticks(fontsize = 14)
plt.yscale("log")
plt.show()
```