Homework 5: Gradient Calculations and Nonlinear Optimization

Introduction to Machine Learning

Shangwen Yan | N17091204 | sy2160

1.

(a)

$$A = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1d} \\ 1 & x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & \vdots & & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}, \quad \boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}, \quad \boldsymbol{z} = A\boldsymbol{w}$$

$$J(\boldsymbol{w}) = g(\boldsymbol{z}) = \sum_{i=1}^{n} g_i(z_i), \quad g_i(z_i) = (y_i - \frac{1}{z_i})^2, \quad z_i = w_0 + \sum_{j=1}^{d} x_{ij} w_j$$

(b)

$$\nabla J(\boldsymbol{w}) = A^T \nabla_z g(\boldsymbol{z}), \quad \nabla_z g(\boldsymbol{z}) = \begin{bmatrix} g_1'(z_1) \\ \vdots \\ g_n'(z_n) \end{bmatrix}, \quad g_i'(z_i) = \frac{2}{{z_i}^2}(y_i - \frac{1}{z_i})$$

(c) $\boldsymbol{w}^k$ is the estimate sequence of $\boldsymbol{w}$:

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^k - a_k \nabla J(\boldsymbol{w}^k)$$

(d)

```python
import numpy as np
def loss_function_grad(X,y,w):
    A = np.column_stack((np.ones(n,), X))
    z = A.dot(w)
    J = np.sum((y-1/z)**2)
    dJ_dz = (2/z**2)*(y-1/z)
    J_grad = A.T.dot(dJ_dz)
    return J,J_grad
```

3.

(a) $\nabla_{\boldsymbol{P}} z_i = \boldsymbol{x_i}\boldsymbol{x_i}^T$

(b) assume that $g_i(z_i) = [\frac{z_i}{y_i} - ln(z_i)]$, $\quad J(\boldsymbol{P}) = g(z) = \sum_{i=1}^{n} g_i(z_i)$, $\quad g_i'(z_i) = [\frac{1}{y_i} - \frac{1}{z_i}]$

$$X_{n\times m} = \begin{bmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_n} \end{bmatrix}, \quad \nabla_z g(z) = \begin{bmatrix} g_1'(z_1) \\ \vdots \\ g_n'(z_n) \end{bmatrix}, \quad H_{m\times n} = \begin{bmatrix} (\nabla_z g(z))^T \\ \vdots \\ (\nabla_z g(z))^T \end{bmatrix}$$

$$\nabla_{\boldsymbol{P}} J(\boldsymbol{P})_{m\times m} = (X^T . H)X$$

A.B produce matrix C, $C_{ij} = A_{ij} B_{ij}$

(c)

```python
def loss_function_grad(X,y,P):
    z = []
    for i in range(X.shape[0]):
        zi = X[i,:].dot(P).dot(X[i,:].T)
        z.append(zi)
    z = np.array(z)
    J = np.sum((z/y)-np.log(z))
    dJ_dz = (1/y)-(1/z)
    J_grad = (X.T*dJ_dz[None,:]).dot(X)  #broadcast
    return J,J_grad
```

(d)

```python
def loss_function_grad(X,y,P):
    z = X.dot(P).dot(X.T).dot(np.ones(X.shape[0]))
    z = np.squeeze(z)
    J = np.sum((z/y)-np.log(z))
    dJ_dz = (1/y)-(1/z)
    J_grad = (X.T*dJ_dz[None,:]).dot(X)
    return J,J_grad
```

4.

(a)

$$\nabla_{\boldsymbol{b}} J(\boldsymbol{b}) = 2\begin{bmatrix} \sum_{i=1}^{n}[(-e^{-a_1 x_i})(y_i - \sum_{j=1}^{d} b_i e^{-a_j x_i})] \\ \vdots \\ \sum_{i=1}^{n}[(-e^{-a_d x_i})(y_i - \sum_{j=1}^{d} b_i e^{-a_j x_i})] \end{bmatrix}_{|a=\hat{a}}$$

and we use gradient descent method to update $\boldsymbol{b}$, thus $\boldsymbol{b}^k$ is the estimate sequence of $\boldsymbol{b}$:

$$\boldsymbol{b}^{k+1} = \boldsymbol{b}^k - a_k \nabla J(\boldsymbol{b}^k)$$

after certain times of iteration or stop criterion, we can get $\hat{\boldsymbol{b}}$

(b)

$$J_1(\boldsymbol{a}) = min_{b=\hat{b}} J(\boldsymbol{a}, \boldsymbol{b})$$

here $\hat{\boldsymbol{b}}$ is constant, and we can use the same method as (a) to compute $\nabla_{\boldsymbol{a}} J(\boldsymbol{a}, \boldsymbol{b})$

$$\nabla J(\boldsymbol{a}, \boldsymbol{b}) = \nabla_{\boldsymbol{a}} J_1(\boldsymbol{a}, \boldsymbol{b})|_{b=\hat{b}} = 2\begin{bmatrix} \sum_{i=1}^{n} (b_1 x_i e^{-a_1 x_i})(y_i - \sum_{j=1}^{d} b_i e^{-a_j x_i}) \\ \vdots \\ \sum_{i=1}^{n} b_d e^{-a_d x_i}(y_i - \sum_{j=1}^{d} b_i e^{-a_j x_i}) \end{bmatrix}_{|b=\hat{b}}$$