COMP9417 – Machine Learning

Homework 1: Regularized Regression & Numerical Optimization

Wanqing Yang - z5325987

Question 1.

(a) $f(x) = \frac{1}{2}\left|\left|Ax - b\right|\right|_2^2 + \frac{\gamma}{2}\left|\left|x\right|\right|_2^2 = \frac{1}{2}(Ax-b)^T(Ax-b) + \frac{\gamma}{2}x^Tx$

$$\frac{\partial f(x)}{\partial x} = \frac{\partial (Ax-b)^T(Ax-b) + \gamma x^T x}{\partial x}$$

$$= \frac{\partial (x^TA^TAx - x^TA^Tb - b^TAx - b^Tb) + \gamma x^T x}{\partial x}$$

$$= A^TAx - A^Tb + \gamma x$$

$$\left|\left|\nabla f(x)\right|\right|_2 = \left|\left|A^TAx - A^Tb + \gamma x\right|\right|_2$$

$$= \sqrt{(A^TAx - A^Tb + \gamma x)^T(A^TAx - A^Tb + \gamma x)}$$

After calculation, the first 5 and last 5 terms of $x^{(k)}$ are

$k = 0,$ $x^{(0)} = [1,1,1,1]$

$k = 1,$ $x^{(1)} = x^{(0)} - \alpha^0 \nabla f(x_0) = [0.98\ 0.98\ 0.98\ 0.98]$

$k = 2,$ $x^{(2)} = x^{(1)} - \alpha^1 \nabla f(x_1) = [0.9624\ 0.9804\ 0.9744\ 0.9584]$

$k = 3,$ $x^{(3)} = x^{(2)} - \alpha^2 \nabla f(x_2) = [0.9427\ 0.9824\ 0.9668\ 0.9433]$

$k = 4,$ $x^{(4)} = x^{(3)} - \alpha^3 \nabla f(x_3) = [0.9234\ 0.9866\ 0.9598\ 0.9295]$

……

$k = 272,$ $x^{(272)} = x^{(271)} - \alpha^{271} \nabla f(x_{271}) = [0.0666\ 1.3366\ 0.4928\ 0.3251]$

$k = 273,$ $x^{(273)} = x^{(272)} - \alpha^{272} \nabla f(x_{272}) = [0.0666\ 1.3366\ 0.4928\ 0.325]$

$k = 274,$ $x^{(274)} = x^{(273)} - \alpha^{273} \nabla f(x_{273}) = [0.0665\ 1.3366\ 0.4927\ 0.325]$

$k = 275,$ $x^{(275)} = x^{(274)} - \alpha^{274} \nabla f(x_{274}) = [0.0664\ 1.3367\ 0.4927\ 0.3249]$

$k = 276,$ $x^{(276)} = x^{(275)} - \alpha^{275} \nabla f(x_{275}) = [0.0663\ 1.3367\ 0.4927\ 0.3249]$

```python
import numpy as np
x = np.array([1, 1, 1, 1])
A = np.array([[1, 2, 1, -1], [-1, 1, 0, 2], [0, -1, -2, 1]])
b = np.array([3, 2, -2])
count = 0
while True:
    count += 1
    delta = (np.dot(np.dot(A.T, A), x)) - (np.dot(A.T, b)) + (0.2 * x)
    x = x - 0.1 * delta
    if np.dot(delta.T, delta) ** 0.5 < 0.001:
        break
    print(count)
    print(np.around(x, 4))
```

(b) (i) Because $\nabla f(x)^{(k)}$ is a slope, and as the gradient goes down to its lowest point the slope is going to approach 0, then $||\nabla f(x)||_2$ is going to approach 0.

(ii) Changing the value on the right increase the output of the algorithm. Because as the slope goes to zero, the slope goes down slower.

(c) From (a) we know that $\frac{\partial f(x)}{\partial x} = A^T Ax - A^T b + \gamma x$

Let $\frac{\partial f(x)}{\partial x} = 0$

Then, $\hat{x} = (A^T Ax + \gamma I)^{-1} A^T b$

The result of $\hat{x}$ is $[0.06285483\ \ 1.33819951\ \ 0.49067315\ \ 0.32238443]$

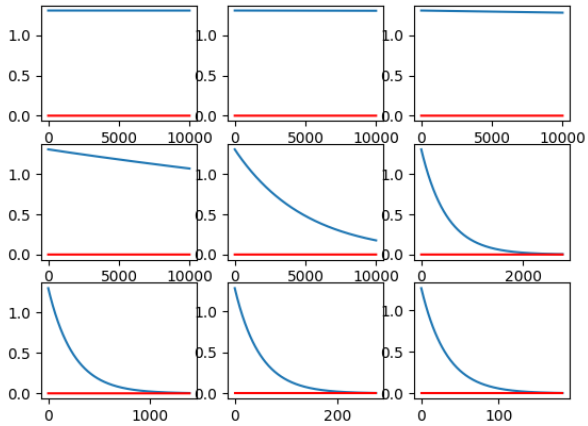
This result is very close to my result from gradient descent.

```
x_ = np.dot(np.linalg.pinv(np.dot(A.T, A) + 0.2*np.eye(4)), np.dot(A.T, b))
print(x_)
```

(d) The change of step-size results in the change of gradient descent velocity, and the larger step-size is, the faster descent speed is.

If step-size is too large, $||\mathrm{x}^{(\mathrm{k})} - \hat{x}||_2$ may not decrease on every iteration, and it may not converge.

```
import matplotlib.pyplot as plt
import numpy as np
A = np.array([[1, 2, 1, -1], [-1, 1, 0, 2], [0, -1, -2, 1]])
b = np.array([3, 2, -2])
alpha_set = np.array([0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.02, 0.1, 0.15])
x_ = np.dot(np.linalg.pinv(np.dot(A.T, A) + 0.2*np.eye(4)), np.dot(A.T, b))
num = 0
for alpha in alpha_set:
    num += 1
    plt.subplot(3, 3, num)
    count = 0
    loss = np.array([])
    line = np.array([])
    x = np.array([1, 1, 1, 1])
    while count <= 10000:
        count += 1
        delta = (np.dot(np.dot(A.T, A), x)) - (np.dot(A.T, b)) + (0.2 * x)
        x = x - alpha * delta
        loss = np.append(loss, np.linalg.norm(x - x_))
        line = np.append(line, 0.001)
        if np.dot(delta.T, delta) ** 0.5 < 0.001:
            break
    step = np.arange(0, count)
    plt.plot(step, loss)
    plt.plot(step, line, color='red')
plt.show()
```



(e) The output of this algorithm is that

mean: [124.975 68.6575 6.635 264.84 115.795 53.3225 13.9]

variance: [2.34559375e+02 7.81260194e+02 4.41167750e+01 2.16655144e+04 5.59182975e+02 2.61793494e+02 6.85000000e+00]

X_train[0]: [0.85045499 0.15536099 0.65717702 0.07581929 0.17782345 -0.69978222 1.18444912]

X_train[-1]:[-0.1942498 0.6920138 -0.24615909 0.47665602 0.43155489 0.65991828 0.03820804]

Y_train[0]: [9.5]

Y_train[-1]: [6.42]

X_test[0]: [1.24221929 0.83512122 -0.99893918 0.57176982 1.27732635 0.53630914 -0.72595268]

X_test[-1]: [0.58927879 -1.13260576 -0.99893918 -1.61584759 0.17782345 -0.26715025 0.80236876]

Y_test[0]: [5.56]

Y_test[-1]: [9.71]

```python
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random
X = pd.read_csv('CarSeats.csv', usecols=[1, 2, 3, 4, 5, 7, 8])
Y = pd.read_csv('CarSeats.csv', usecols=[0])
X = X.values
Y = Y.values
X_scaler = StandardScaler().fit(X)
print("mean: ", X_scaler.mean_)
print("variance: ", X_scaler.var_)
scaler_X = X_scaler.transform(X)
X_train = scaler_X[:200]
X_test = scaler_X[200:400]
Y_train = Y[:200]
Y_test = Y[200:400]
print("X_train[0]: ", X_train[0])
print("X_train[-1]: ", X_train[-1])
print("Y_train[0]: ", Y_train[0])
print("Y_train[-1]: ", Y_train[-1])
print("X_test[0]: ", X_test[0])
print("X_test[-1]: ", X_test[-1])
print("Y_test[0]: ", Y_test[0])
print("Y_test[-1]: ", Y_test[-1])
```

(f) When features are not standardized, some features have a large range (such as 0~3000) and others have a small range (such as 0~5). When gradient descent is carried out, the descent direction is always perpendicular to the contour line, so the descent speed will become slow.

(g) $\hat{\beta}_{Ridge} = arg\min_{\beta} \frac{1}{n}(y - X\beta)^T(y - X\beta) + \Phi\beta^T\beta$

Then $L(\beta) = \frac{1}{n}(y - X\beta)^T(y - X\beta) + \Phi\beta^T\beta$

Let $\frac{\partial L(\beta)}{\partial \beta}$ =0

Then $\hat{\beta}_{Ridge} = (X^TX + n\Phi I)^{-1}(X^Ty)$

The result of this algorithm is [0.41960543 0.83471693 0.00243251 -0.24508189 -1.31946063 -0.03459347 0.31302924]

```python
def Compute(x, y):
    m, n = x.shape
    I = np.eye(n)
    beta_R = np.dot(np.linalg.pinv(np.dot(x.T, x) + m*0.5*I), np.dot(x.T, y))
    return beta_R
```

(h) $L(\beta) = \frac{1}{n}||y - X\beta||_2^2 + \Phi||\beta||_2^2 = \frac{1}{n}\sum_{i=0}^{n}(y_i - x_i^T\beta)^2 + \Phi\beta^T\beta$

Then $\nabla L(\beta) = \frac{\partial L(\beta)}{\partial \beta} = \frac{1}{n}\sum_{i=0}^{n}[-2x_i(y_i - x_i^T\beta)] + 2\Phi\beta$

Because $L(\beta) = \frac{1}{n}\sum_{i=0}^{n} L_i(\beta)$

Then $\nabla L(\beta) = \frac{1}{n}\sum_{i=0}^{n} L_i(\beta)$

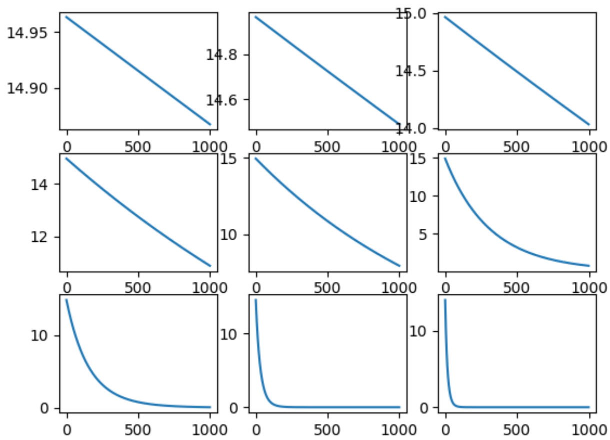
Then $\nabla L_i(\beta) = -2x_i\left(y_i - x_i^T\beta\right) + 2\Phi\beta$

(i) (i) The train MSE is 58.5945375855534

(ii) The test MSE is 13880.305071604535

```python
def BGD(x, y, beta_R, x_t, y_t, max_count = 1000):
    m, n = x.shape
    alpha_set = np.array([0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01])
    num = 0
    for alpha in alpha_set:
        num += 1
        plt.subplot(3, 3, num)
        beta = np.ones((n,), dtype=np.float64)
        count = 0
        loss = np.array([])
        while count < max_count:
            count += 1
            delta = np.zeros((n,), dtype=np.float64)
            for i in range(m):
                delta += -2*x[i] * (y[i]-np.dot(x[i].T, beta)) + beta
            beta = beta - (alpha/m) * delta
            loss = np.append(loss, L(x, y, beta) - L(x, y, beta_R))
        if alpha == 0.01:
            MSE1 = y - np.dot(x, beta)
            Train_MSE = (1/m) * (np.linalg.norm(MSE1) ** 2)
            MSE2 = y_t - np.dot(x_t, beta)
            Test_MSE = (1/m) * (np.linalg.norm(MSE2) ** 2)
            print("Train_MSE: ", Train_MSE)
            print("Test_MSE: ", Test_MSE)
        step = np.arange(0, 1000)
        plt.plot(step, loss)
    plt.show()
```

```python
def L(x, y, beta):
    m, n = x.shape
    ans = (1/m) * np.dot((y-np.dot(x, beta)).T, (y-np.dot(x, beta))) + 0.5*np.dot(beta.T, beta)
    return ans
```

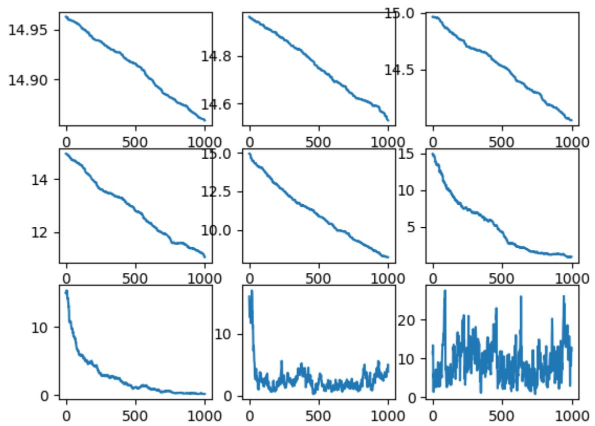


(j) (i) The train MSE is 58.98010069682144

(ii) The test MSE is 13977.751807191089

```python
def SGD(x, y, beta_R, x_t, y_t, max_count = 1000):
    m, n = x.shape
    alpha_set = np.array([0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02])
    num = 0
    for alpha in alpha_set:
        num += 1
        plt.subplot(3, 3, num)
        beta = np.ones((n,), dtype=np.float64)
        count = 0
        loss = np.array([])
        while count < max_count:
            count += 1
            delta = np.zeros((n,), dtype=np.float64)
            i = random.randint(0, m - 1)  # 随机选择一个样本
            delta += -2*x[i] * (y[i]-np.dot(x[i].T, beta)) + beta
            beta = beta - alpha * delta
            loss = np.append(loss, L(x, y, beta) - L(x, y, beta_R))
        if alpha == 0.001:
            MSE1 = y - np.dot(x, beta)
            Train_MSE = (1/m) * (np.linalg.norm(MSE1) ** 2)
            MSE2 = y_t - np.dot(x_t, beta)
            Test_MSE = (1/m) * (np.linalg.norm(MSE2) ** 2)
            print("Train_MSE: ", Train_MSE)
            print("Test_MSE: ", Test_MSE)
        step = np.arange(0, 1000)
        plt.plot(step, loss)
    plt.show()
```

```python
def L(x, y, beta):
    m, n = x.shape
    ans = (1/m) * np.dot((y-np.dot(x, beta)).T, (y-np.dot(x, beta))) + 0.5*np.dot(beta.T, beta)
    return ans
```



(k) I prefer SGD. Because the real problem training data is now very large, the execution efficiency of SGD is higher than BDG.

BGD is used when the data size is small or the error required is small.

SGD is used when the data is large.