COMP9417 - Machine Learning Homework 1: Regularized Regression & Numerical Optimization

a)

The equation outlining the explicit gradient update:

$$x^{(k+1)} = x^k - \alpha(A^T(Ax^k - b) + \gamma x^k)$$

A print out of the first 5 (k = 5 inclusive) and last 5 rows of iterations:

```
(cv) ➜  comp9417 git:(master) python -u "/Users/yueyifei/Desktop/COMP9417/comp9417/hw1/code/q_a.py"
k = 0, x(k) = [1,1,1,1]
k = 1, x(k) = [0.98,0.98,0.98,0.98]
k = 2, x(k) = [0.9624,0.9804,0.9744,0.9584]
k = 3, x(k) = [0.9427,0.9824,0.9668,0.9433]
k = 4, x(k) = [0.9234,0.9866,0.9598,0.9295]
k = 5, x(k) = [0.9044,0.9916,0.9526,0.9169]
k = 272, x(k) = [0.0666,1.3366,0.4928,0.3251]
k = 273, x(k) = [0.0666,1.3366,0.4928,0.325]
k = 274, x(k) = [0.0665,1.3366,0.4927,0.325]
k = 275, x(k) = [0.0664,1.3367,0.4927,0.3249]
k = 276, x(k) = [0.0663,1.3367,0.4927,0.3249]
```

A screen shot of code for a):

```
comp9417 > hw1 > code > q_a.py > j
You, 1 hour ago | 1 author (You)
import numpy as np

alpha = 0.1
gamma = 0.2

A = np.array([[1,2,1,-1], [-1,1,0,2], [0,-1,-2,1]])      # A: 3 x 4
b = np.array([[3], [2], [-2]])                           # b: 3 x 1
A_t = A.T                                                # A_t: 4 x 3
x0 = np.array([1,1,1,1]).reshape(4, 1)                   # x_0: 1 x 4 -> 4 x 1

x_list = []
x_list.append(x0)

x = x0
k = 1

while True:
    grad = A_t @ (A @ x - b) + gamma * x
    if np.linalg.norm(grad, ord=2) < 0.001:
        break
    new_x = x - alpha * grad
    x = new_x
    x_list.append(new_x)
    k = k + 1

length = len(x_list)

for i in range(0, 6):
    x = x_list[i].reshape(1, 4)[0]
    print("k = {}, x(k) = [{},{},{},{}]".format(i,round(x[0],4) ,round(x[1],4), round(x[2],4), round(x[3],4)))

for j in range(length - 5, length):
    x = x_list[j].reshape(1, 4)[0]
    print("k = {}, x(k) = [{},{},{},{}]".format(j,round(x[0],4) ,round(x[1],4), round(x[2],4), round(x[3],4)))

```

b)

This termination condition means that when the gradient of the function is less than 0.001, we consider that the algorithm has converged to a minimizer of f.

If other conditions remain unchanged and the gradient required by the termination condition is limited to a smaller value, we will calculate a more accuate value of x that minimizes f. Therefore, we need to complete more iterations, and the time for the algorithm to converge to a minimizer of f also becomes longer.

c)

A print out of the first 5 (k = 5 inclusive) and last 5 rows of iterations:

```
(cv) → comp9417 git:(master) python -u "/Users/yueyifei/Desktop/COMP9417/comp9417/hw1/code/q_c.py"
k = 0, x(k) = [1.0,1.0,1.0,1.0]
k = 1, x(k) = [0.98,0.98,0.98,0.98]
k = 2, x(k) = [0.9624,0.9804,0.9744,0.9584]
k = 3, x(k) = [0.9427,0.9824,0.9668,0.9433]
k = 4, x(k) = [0.9234,0.9866,0.9598,0.9295]
k = 5, x(k) = [0.9044,0.9916,0.9526,0.9169]
k = 272, x(k) = [0.0666,1.3366,0.4928,0.3251]
k = 273, x(k) = [0.0666,1.3366,0.4928,0.325]
k = 274, x(k) = [0.0665,1.3366,0.4927,0.325]
k = 275, x(k) = [0.0664,1.3367,0.4927,0.3249]
k = 276, x(k) = [0.0663,1.3367,0.4927,0.3249]
```

A screen shot of code for c):

comp9417 > hw1 > code > q_c.py > i

You, 4 hours ago | 1 author (You)

```python
import torch
import torch.nn as nn
from torch import optim

gamma = 0.2
alpha = 0.1

A = torch.tensor([[1., 2., 1., -1.], [-1., 1., 0., 2.], [0., -1., -2., 1.]])
b = torch.tensor([[3.], [2.], [-2.]])
A_t = A.t()
x0 = torch.tensor([[1.], [1.], [1.], [1.]])

x_dict = {}
x_dict[0] = x0.tolist()
x = x0.clone()
```

You, 4 hours ago | 1 author (You)

```python
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.x = nn.Parameter(x, requires_grad=True)
    def forward(self, x):
        z = torch.mul(torch.pow(torch.norm(torch.sub(torch.mm(A, x), b)), 2), 1/2)
        w = torch.mul(torch.pow(torch.norm(x), 2), gamma/2)
        return torch.add(z, w)

model = MyModel()
optimizer = optim.SGD(model.parameters(), lr=alpha)
terminationCond = False


k = 1

while not terminationCond:
    optimizer.zero_grad()
    loss = model(model.x)
    loss.backward()
    optimizer.step()

    if torch.norm(model.x.grad) < 0.001:
        terminationCond = True
    else:
        x_dict[k] = model.x.tolist()
        k = k + 1

length = len(x_dict)

for i in range(0, 6):
    x = x_dict[i]
    print("k = {}, x(k) = [{},{},{},{}]".format(i, round(x[0][0], 4) ,round(x[1][0], 4), round(x[2][0], 4), round(x[3][0], 4)))


for j in range(length - 5, length):
    x = x_dict[j]
    print("k = {}, x(k) = [{},{},{},{}]".format(j, round(x[0][0], 4) ,round(x[1][0], 4), round(x[2][0], 4), round(x[3][0], 4)))
```

d)

We apply fit_transform() on the training data to scale the training data (standardize the remaining features) and also learn the scaling parameters of that data which are the mean and variance of the features of the training set. After centering the target variable, we can create our training set and testing set and output the result.

A print out of the means and variances of features and the first and last rows of the 4 requested objects:

```
(base) ➜  hw1 git:(master) ✗ python -u "/Users/yueyifei/Desktop/COMP9417/comp9417/hw1/code/q_degh.py"
The means of features:
[ 3.81916720e-16  3.55271368e-17  2.66453526e-17  1.59872116e-16
 -6.21724894e-17  1.28785871e-16 -1.33226763e-16]

The variances of features:
[1. 1. 1. 1. 1. 1. 1.]

The first and last rows of X_train:
[ 0.85045499  0.15536099  0.65717702  0.07581929  0.17782345 -0.69978222
  1.18444912]
[-0.1942498   0.6920138  -0.24615909  0.47665602  0.43155489  0.65991828
  0.03820804]

The first and last rows of X_test:
[ 1.24221929  0.83512122 -0.99893918  0.57176982  1.27732635  0.53630914
 -0.72595268]
[ 0.58927879 -1.13260576 -0.99893918 -1.61584759  0.17782345 -0.26715025
  0.80236876]

The first and last rows of Y_train:
[2.003675]
[-1.076325]

The first and last rows of Y_test:
[-1.936325]
[2.213675]
```

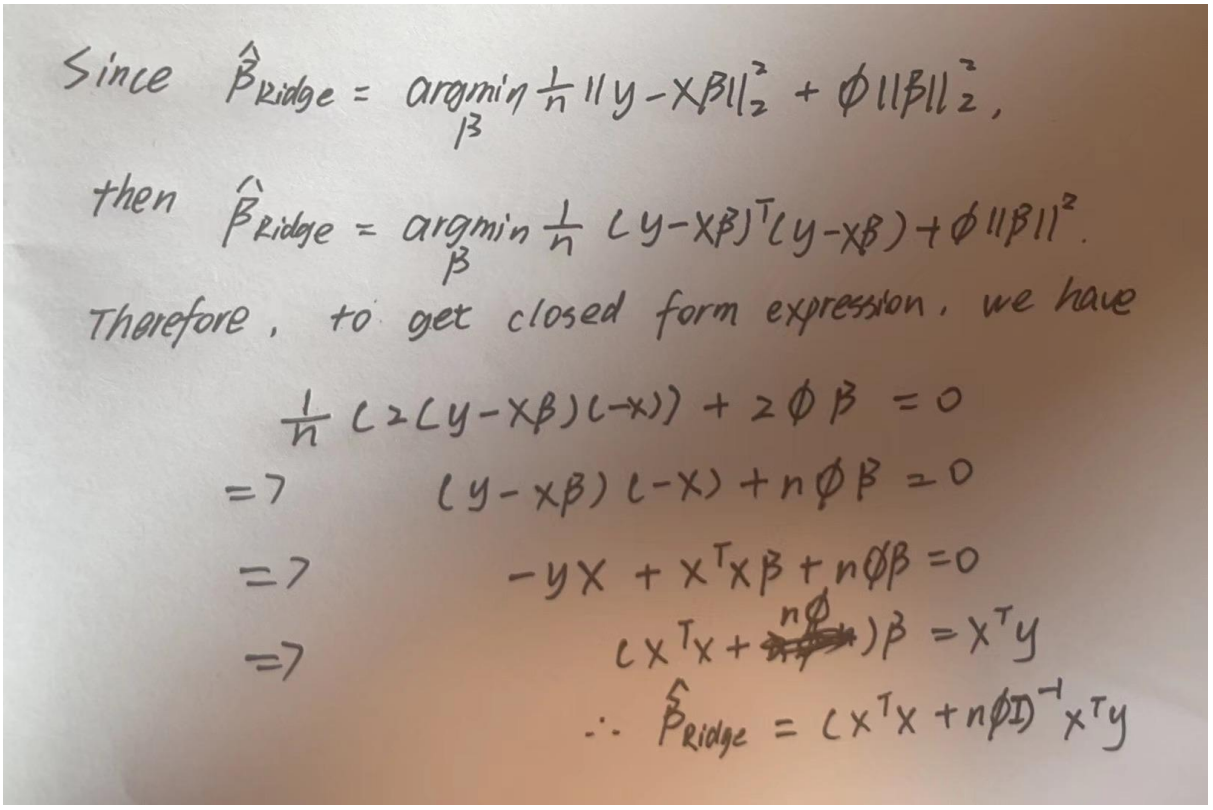
A screen shot of code for d):

comp9417 > hw1 > code > q_degh.py > ...

You, 56 seconds ago | 1 author (You)

```python
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import copy


### Code for Question d
data = pd.read_csv('CarSeats.csv')

label = ['Sales']
target = data[label]

categorical_features = ['ShelveLoc', 'Urban', 'US']

numerical_data = data.drop(label + categorical_features, axis=1)

scaler = StandardScaler()
numerical_data_scaled = scaler.fit_transform(numerical_data)

print('The means of features:')
print(numerical_data_scaled.mean(axis=0))
print()
print('The variances of features:')
print(numerical_data_scaled.var(axis=0))
print()

target_centered = (target - target.mean(axis=0)).to_numpy()
X_train, X_test, Y_train, Y_test = train_test_split(numerical_data_scaled, target_centered, test_size=0.5, shuffle=False)

print('The first and last rows of X_train:')
print(X_train[0])
print(X_train[-1])
print()
print('The first and last rows of X_test:')
print(X_test[0])
print(X_test[-1])
print()
print('The first and last rows of Y_train:')
print(Y_train[0])
print(Y_train[-1])
print()
print('The first and last rows of Y_test:')
print(Y_test[0])
print(Y_test[-1])
print()
```

e)



A print out of the value of the ridge solution:

```
The value of the ridge solution based on X_train and Y_train:
[[ 0.680673  ]
 [ 0.28229334]
 [ 0.65157017]
 [ 0.00834835]
 [-1.17129533]
 [-0.400892  ]
 [-0.10063355]]
```

A screen shot of code for e):

```
50  ### Code for Question e
51  phi = 0.5
52  size = len(data.columns) - len(label + categorical_features)
53  X_t = X_train.T
54
55  beta_hat = np.linalg.inv(X_t @ X_train + phi * len(X_train) * np.identity(size)) @ X_t @ Y_train
56
57  print('The value of the ridge solution based on X_train and Y_train:')
58  print(beta_hat)
```

f)

Since $L(\beta) = \frac{1}{n}\|y - X\beta\|_2^2 + \phi\|\beta\|_2^2$

$= \frac{1}{n}\sum_{i=1}^{n}(y_i - X_i\beta)^2 + \phi\sum_{j=1}^{n}\beta_j^2$,

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

Then we have, $nL(\beta) = \sum_{i=1}^{n}(y_i - X_i\beta)^2 + n\phi\sum_{j=1}^{n}\beta_j^2$

$= \sum_{i=1}^{n} L_i(\beta)$,

then we have

$L_i(\beta) = (y_i - X_i\beta)^2 + \phi\sum_{j=1}^{n}\beta_j^2$

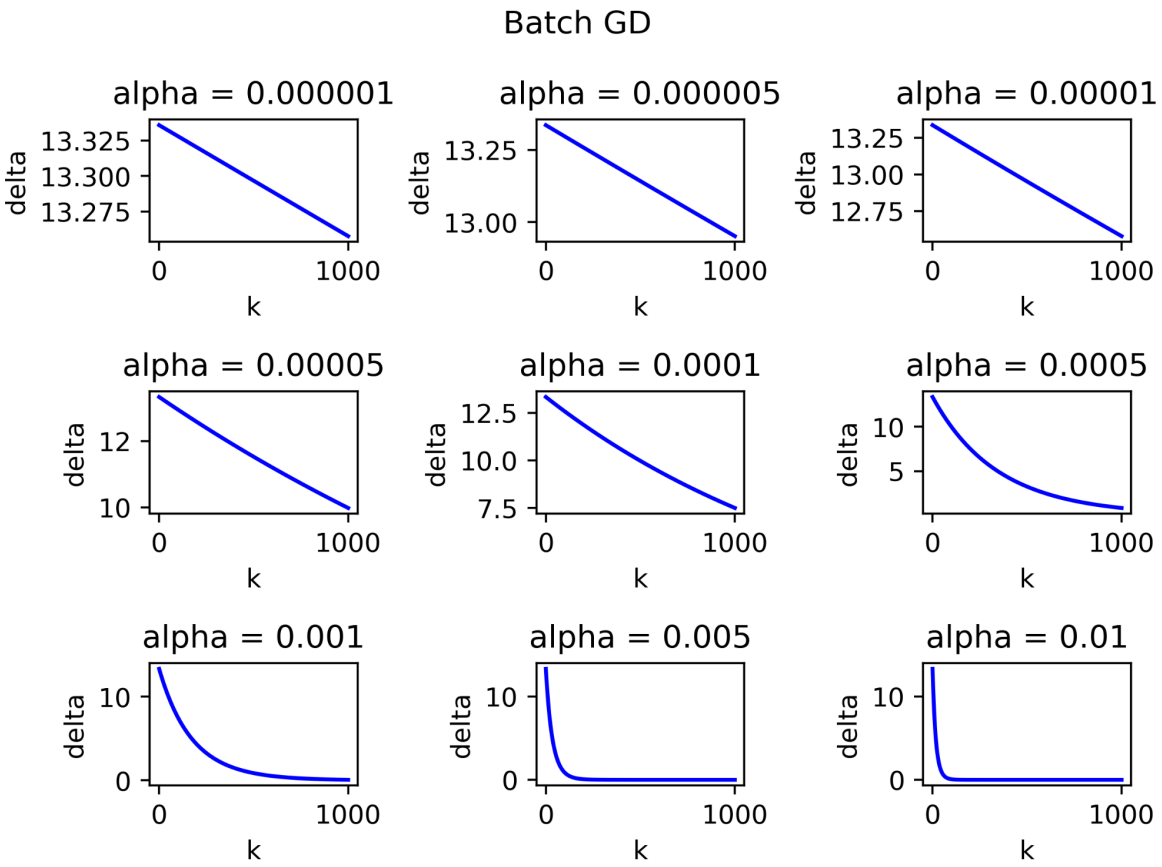
Therefore, $L_n(\beta) = (y_n - X_n\beta)^2 + \phi\sum_{j}\beta_j^2$

$\nabla L_n(\beta) = \frac{\partial L_n(\beta)}{\partial \beta} = 2(y_n - X_n\beta)(-X_n) + 2\phi\beta$

$= -2(y_n - X_n\beta)X_n + 2\phi\beta$

Therefore, $\nabla L_n(\beta) = -2(y_n - X_n\beta)X_n + 2\phi\beta$

g)



We choose the step-size alpha = 0.005, in fact, both 0.005 and 0.01 have good effects, which can make the algorithm converge faster. But we choose 0.005 to further avoid the potential error caused by large step size.

A print out of the train and test MSE requested:

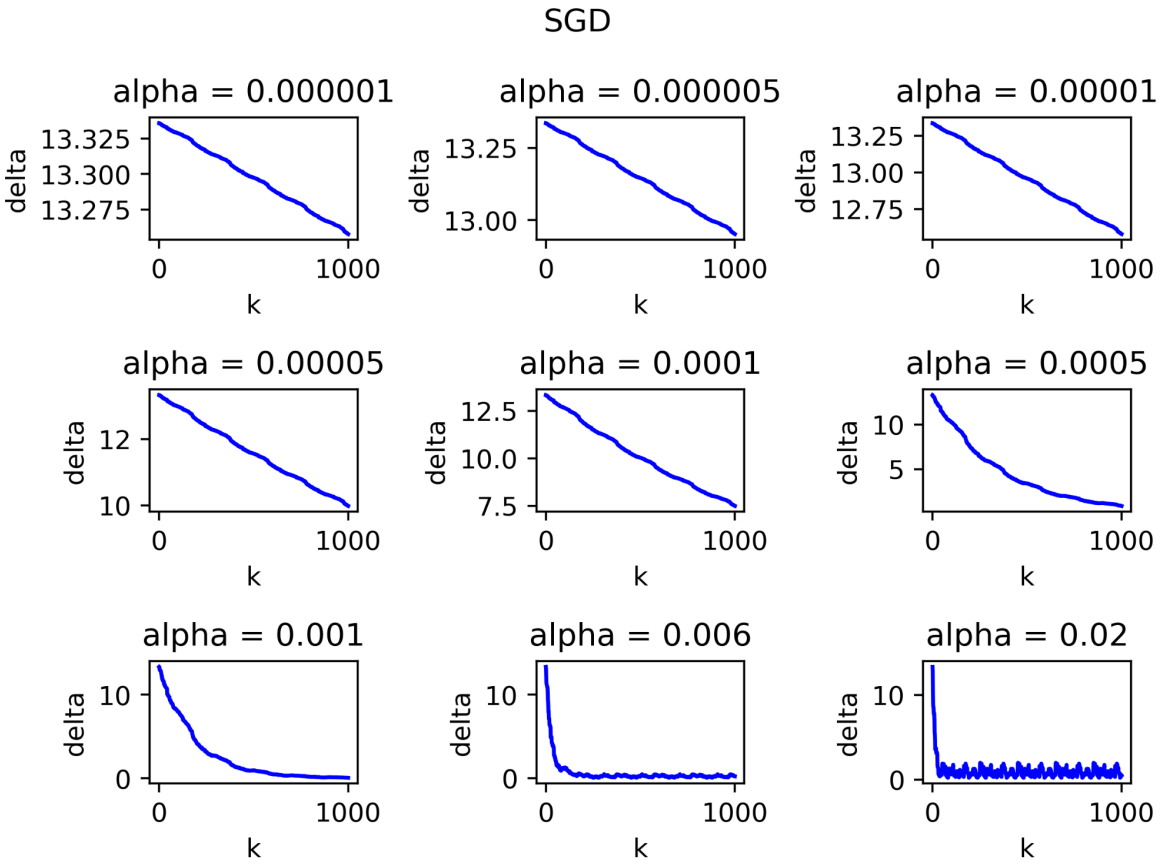
```
The train MSE is : 4.559027077966185
The test MSE is : 4.380486336397683
```

A screen shot of code for g):

```python
### Code for Question g
beta_0 = np.ones(X_train.shape[1]).reshape(7, 1)

phi = 0.5
epochs = 1000
alphas = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01]

def loss(X_train, Y_train, beta, phi):
    loss_beta = 1/len(X_train) * (np.linalg.norm(Y_train - X_train @ beta, ord=2) ** 2) + phi * (np.linalg.norm(beta, ord=2) ** 2)
    return loss_beta

def grad(X_train, Y_train, beta, phi):
    grad_beta = -2 * (1/len(X_train)) * X_train.T @ (Y_train - X_train @ beta) + 2 * phi * beta
    return grad_beta

def get_gd_betas(beta_0, X_train, Y_train, alpha, phi, epochs):
    beta = beta_0
    betas = []
    betas.append(beta)
    for k in range(epochs):
        b_updated = beta - alpha * grad(X_train, Y_train, beta, phi)
        beta = b_updated
        betas.append(b_updated)
    return betas

def get_gd_deltas(beta_0, X_train, Y_train, alpha, phi, epochs):
    deltas = []
    betas = get_gd_betas(beta_0, X_train, Y_train, alpha, phi, epochs)
    loss_hat = loss(X_train, Y_train, beta_hat, phi)
    for beta in betas:
        delta = loss(X_train, Y_train, beta, phi) - loss_hat
        deltas.append(delta)
    return deltas

i = 1
for alpha in alphas:
    deltas = get_gd_deltas(beta_0, X_train, Y_train, alpha, phi, epochs)
    axes = plt.subplot(3, 3, i)
    plt.plot(range(epochs + 1), deltas, color='blue')
    plt.xlabel('k')
    plt.ylabel('delta')
    plt.title('alpha = ' + str(np.format_float_positional(alpha)))
    i = i + 1
plt.suptitle('Batch GD')
plt.tight_layout()
plt.savefig('GD_plot.png', dpi=400)
plt.show()
```

```python

  beta_best = get_gd_betas(beta_0, X_train, Y_train, 0.005, phi, epochs)[-1]

  train_MSE = 1/len(X_train) * (np.linalg.norm(Y_train - X_train @ beta_best) ** 2)
  print('The train MSE is :', train_MSE)

  test_MSE = 1/len(X_train) * (np.linalg.norm(Y_test - X_test @ beta_best) ** 2)
  print('The test MSE is :', test_MSE)

```

h)



We choose the step-size alpha = 0.006, because when the step size used by the algorithm is smaller than this value, although the descent process is stable, but the convergence speed is slow, and when the step size is larger than this, the descending process is very unstable. Therefore, we choose alpha = 0.006.

A print out of the train and test MSE requested:

```
The train MSE is : 4.661749176041872
The test MSE is : 4.447228989660326
```

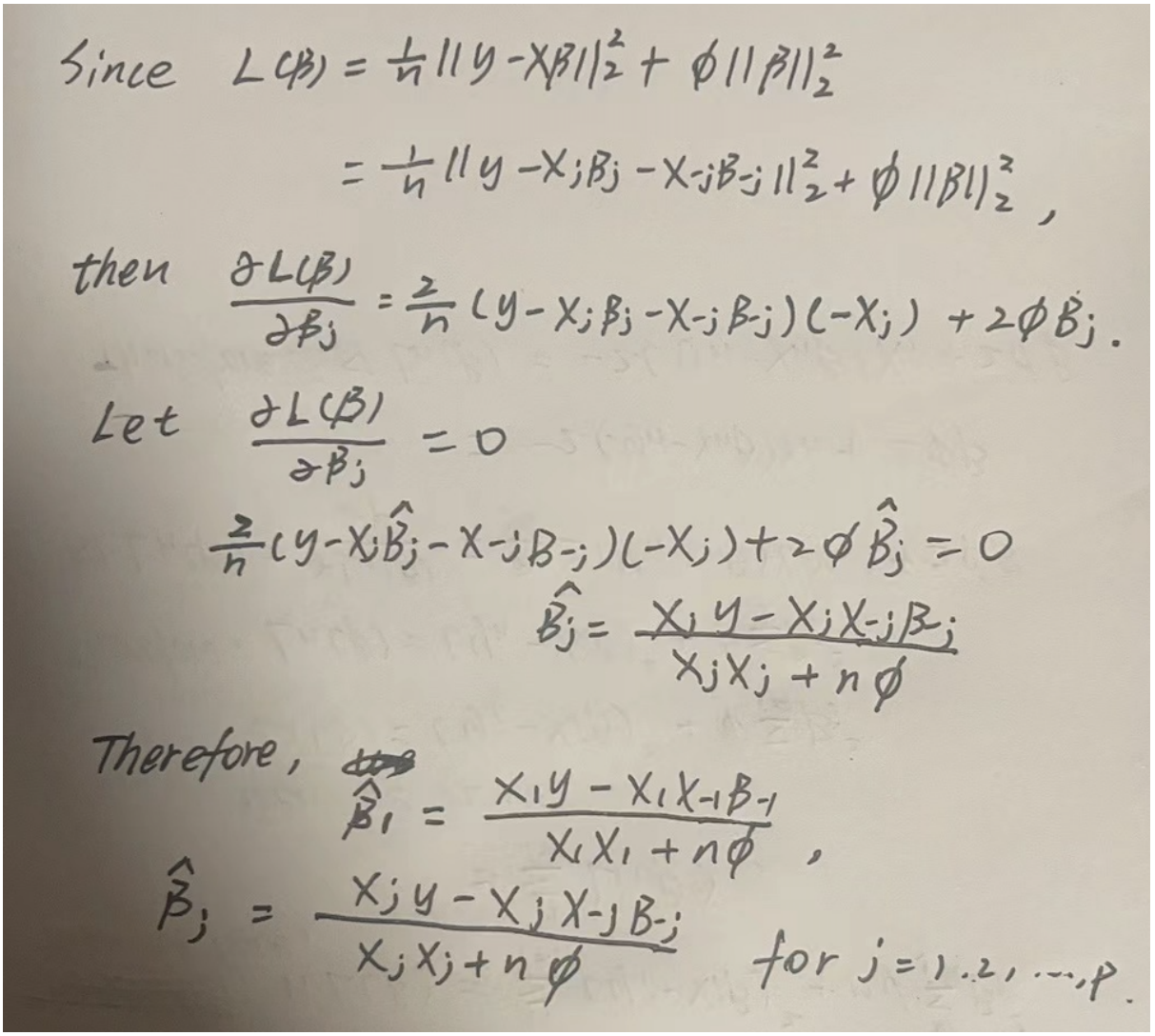
A screen shot of code for h):

```python
### Code for Question h
beta_0 = np.ones(X_train.shape[1]).reshape(7, 1)
phi = 0.5
epochs = 5
sgd_alphas = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02]

def get_sgd_betas(beta_0, X_train, Y_train, alpha, phi, epochs):
    beta = beta_0
    betas = []
    betas.append(beta)
    for i in range(epochs):
        for j in range(len(X_train)):
            x = X_train[j]
            y = Y_train[j]
            x = x.reshape(7,1)
            x_t = x.T
            b_updated = beta - alpha * (-2 * x @ ( y - x_t @ beta) + 2 * phi * beta)
            beta = b_updated
            betas.append(b_updated)
    return betas

def get_sgd_deltas(beta_0, X_train, Y_train, alpha, phi, epochs):
    deltas = []
    betas = get_sgd_betas(beta_0, X_train,Y_train, alpha, phi, epochs)
    loss_hat = loss(X_train, Y_train, beta_hat, phi)
    for beta in betas:
        delta = loss(X_train, Y_train, beta, phi) - loss_hat
        deltas.append(delta)
    return deltas

i = 1
for alpha in sgd_alphas:
    deltas = get_sgd_deltas(beta_0, X_train, Y_train, alpha, phi, epochs)
    axes = plt.subplot(3, 3, i)
    plt.plot(range(epochs * len(X_train) + 1), deltas, color='blue')
    plt.xlabel('k')
    plt.ylabel('delta')
    plt.title('alpha = ' + str(np.format_float_positional(alpha)))
    i = i + 1
plt.suptitle('SGD')
plt.tight_layout()
plt.savefig('SGD_plot.png', dpi=400)
plt.show()

beta_best = get_sgd_betas(beta_0, X_train, Y_train, 0.006, phi, epochs)[-1]

train_MSE = 1/len(X_train) * (np.linalg.norm(Y_train - X_train @ beta_best) ** 2)
print('The train MSE is :', train_MSE)

test_MSE = 1/len(X_train) * (np.linalg.norm(Y_test - X_test @ beta_best) ** 2)
print('The test MSE is :', test_MSE)
```

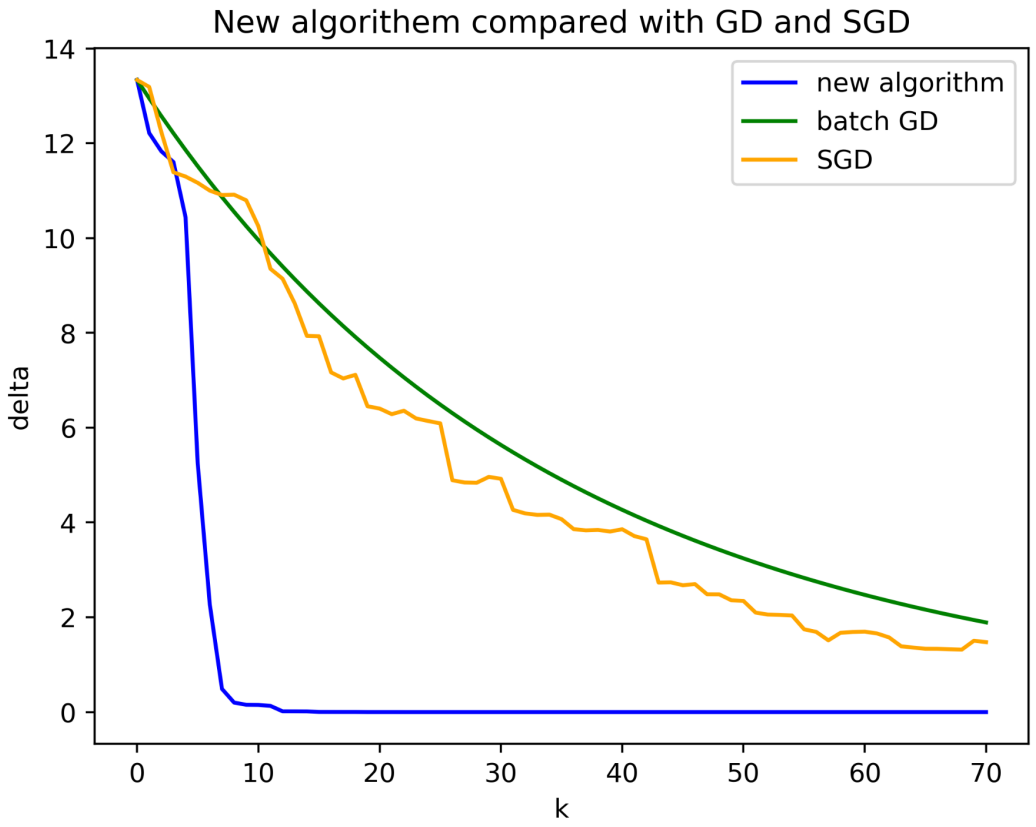
i)

According to our results, the descending process of SGD has some fluctuation and is not very stable, while the descending process of GD is more stable. There is no big difference between the two in terms of the convergence speed, so we choose to use GD.

j)


Since $L(\beta) = \frac{1}{n}\|y - X\beta\|_2^2 + \phi\|\beta\|_2^2$
$= \frac{1}{n}\|y - X_j\beta_j - X_{-j}\beta_{-j}\|_2^2 + \phi\|\beta\|_2^2$,
then $\frac{\partial L(\beta)}{\partial \beta_j} = \frac{2}{n}(y - X_j\beta_j - X_{-j}\beta_{-j})(-X_j) + 2\phi\beta_j$.
Let $\frac{\partial L(\beta)}{\partial \beta_j} = 0$
$\frac{2}{n}(y - X_j\hat{\beta}_j - X_{-j}\beta_{-j})(-X_j) + 2\phi\hat{\beta}_j = 0$
$\hat{\beta}_j = \frac{X_j y - X_j X_{-j}\beta_{-j}}{X_j X_j + n\phi}$
Therefore,
$\hat{\beta}_1 = \frac{X_1 y - X_1 X_{-1}\beta_{-1}}{X_1 X_1 + n\phi}$,
$\hat{\beta}_j = \frac{X_j y - X_j X_{-j}\beta_{-j}}{X_j X_j + n\phi}$ for $j = 1, 2, \ldots, p$.

k)



A print out of the train and test MSE requested:

```
The train MSE is : 4.55890673054406 65
The test MSE is : 4.3804291836600 91
```

A screen shot of code for k):

```python
### Code for Question k
cycles = 10
p = len(X_train[0])
beta_0 = np.ones(X_train.shape[1]).reshape(7, 1)
phi = 0.5


def get_new_betas(beta_0, X_train, Y_train, phi):
    beta = beta_0
    betas = []
    betas.append(beta)
    y = Y_train
    for i in range(cycles):
        for j in range(p):
            x_j = X_train[:, j].reshape(1, len(X_train))
            x_nj = np.delete(X_train, j, 1)
            beta_nj = np.delete(beta, j, 0)
            bj_updated = (x_j @ y - x_j @ x_nj @ beta_nj) / (x_j @ x_j.T + len(X_train) * phi)
            beta = copy.deepcopy(beta)
            beta[j] = bj_updated
            betas.append(beta)
    return betas


def get_new_deltas(beta_0, X_train, Y_train, phi):
    deltas = []
    betas = get_new_betas(beta_0, X_train,Y_train, phi)
    loss_hat = loss(X_train, Y_train, beta_hat, phi)
    for beta in betas:
        delta = loss(X_train, Y_train, beta, phi) - loss_hat
        deltas.append(delta)
    return deltas


deltas = get_new_deltas(beta_0, X_train, Y_train, phi)

plt.plot(range(cycles * p + 1), deltas, color='blue', label='new algorithm')
plt.plot(range(cycles * p + 1), get_gd_deltas(beta_0, X_train, Y_train, 0.005, phi, 1000)[0:cycles * p + 1], color='green', label='batch GD')
plt.plot(range(cycles * p + 1), get_sgd_deltas(beta_0, X_train, Y_train, 0.006, phi, 5)[0:cycles * p + 1], color='orange', label='SGD')
plt.xlabel('k')
plt.legend(loc='upper right')                          # creates legend in top right corner of plot
plt.ylabel('delta')
plt.title('New algorithem compared with GD and SGD')
plt.savefig('comparison.png', dpi=400)
plt.show()

beta_best = get_new_betas(beta_0, X_train, Y_train, phi)[-1]

train_MSE = 1/len(X_train) * (np.linalg.norm(Y_train - X_train @ beta_best) ** 2)
print('The train MSE is :', train_MSE)

test_MSE = 1/len(X_train) * (np.linalg.norm(Y_test - X_test @ beta_best) ** 2)
print('The test MSE is :', test_MSE)
```

l)

Less reliable. If we also standardize the target value, the proportion of the target value predicted by the model is different. The model needs to transform the predicted target value to the original value before the standardization, and there may be errors in the process.