Q1(a)

$$k = 0, x^0 = [1, 1, 1, 1]$$

$$k = 1, x^1 = [0.98, 0.98, 0.98, 0.98]$$

$$k = 2, x^2 = [0.9624, 0.9804, 0.9744, 0.9584]$$

$$k = 3, x^3 = [0.9427, 0.9824, 0.9668, 0.9433]$$

$$k = 4, x^4 = [0.9234, 0.9866, 0.9598, 0.9525]$$

$$k = 5, x^5 = [0.9044, 0.9916, 0.9526, 0.9169]$$

...

$$k = 158, x^{158} = [0.1006, 1.3217, 0.5127, 0.3499]$$

$$k = 159, x^{159} = [0.0999, 1.3223, 0.5118, 0.3488]$$

$$k = 160, x^{160} = [0.0991, 1.3227, 0.5114, 0.3483]$$

$$k = 161, x^{161} = [0.0984, 1.323, 0.511, 0.3487]$$

$$k = 162, x^{162} = [0.0984, 1.323, 0.511, 0.3487]$$

```
import numpy as np
import matplotlib.pyplot as plt
A = np.array([[1,2,1,-1],[-1,1,0,2],[0,-1,-2,1]])
x_t = np.array([[0.06292],[1.33817],[0.49067],[0.3224]])
b = np.array([[3],[2],[-2]])
step = 10000
lambdas = [0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.02,0.1,0.15]
def all_learning(lambdas):
    L = len(lambdas)
    for i in range(L):
        diff = []
        x = np.array([[1],[1],[1],[1]])
        plt.subplot(3, 3, i+1)
        gradient_decent(x,A,b,0.1,step,diff)
        #plt.plot(diff)
def cal_gradient(A, b, x):
    left = np.dot(np.dot(A.T, A), x)
    right = np.dot(A.T, b)
    gradient = left - right
    return gradient+0.2*x
def gradient_decent(x, A, b, learning_rate, step,diff):
    #print(learning_rate)
    diff = []
    for i in range(step):
        diff.append(np.linalg.norm(x-x_t))
        gradient = cal_gradient(A, b, x)
        if np.linalg.norm(gradient) < 0.001:
            break
        else:
            delta = learning_rate * gradient
            x = x - delta
        print('x{a} = {b}'.format(a=i, b=x.round(4)))
    plt.plot(diff)
    plt.axhline(y=0.001, color ='red')
    plt.xlabel(learning_rate)
```

```
    plt.plot(diff)
    plt.axhline(y=0.001, color ='red')
    plt.xlabel(learning_rate)
    print('finish! x = {a} diff = {b}'.format(a=x.round(4), b=diff[-1]))
all_learning(lambdas)
plt.subplots_adjust(wspace=0.2,hspace=0.6)
```

```
x0 = [[0.98]
 [0.98]
 [0.98]
 [0.98]]
x1 = [[0.9624]
 [0.9804]
 [0.9744]
 [0.9584]]
x2 = [[0.9427]
 [0.9824]
 [0.9668]
 [0.9433]]
x3 = [[0.9234]
 [0.9866]
 [0.9598]
 [0.9295]]
x4 = [[0.9044]
 [0.9916]
 [0.9526]
```

(b) If we make the termination condition smaller (0.0001), it would take 276 iterations(more time) to terminate the condition with $x^{276} = [0.0663, 1.3367, 0.4927, 0.3249]$. The output is much closer to the exact numerical value below, so that we can conclude that the gradient decent means in terms of convergence of the algorithm to the exact value.

(c)

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

$$= \frac{1}{2}(Ax-b)^T(Ax-b) + \gamma x$$

$$= \frac{1}{2}(x^TA^TAx - 2b^TAx + b^Tb) + \gamma x$$

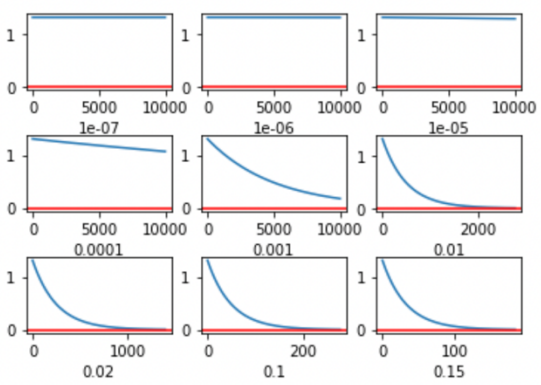
$$\frac{d}{dx} = \frac{1}{2}(-2A^Tb + 2A^TAx) + \gamma$$

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

$$\hat{x} = (A^TA + \gamma I)^{-1}A^Tb$$

The exact numerical value is $\hat{x} = [0.06292, 1.33817, 0.49067, 0.3224]$ which is close to the solution from (a) below

$$\begin{pmatrix} 0.30618 & -0.15579 & 0.27202 \\ 0.38929 & 0.20478 & 0.11963 \\ -0.19059 & 0.04731 & -0.48391 \\ -0.04258 & 0.30313 & 0.07806 \end{pmatrix}\begin{pmatrix} 3 \\ 2 \\ -2 \end{pmatrix} = \begin{pmatrix} 0.06292 \\ 1.33817 \\ 0.49067 \\ 0.3224 \end{pmatrix}$$



(d)when the stepsize is 0.0000001, the result after 10000 steps is $x = \begin{pmatrix} 0.9998 \\ 0.9998 \\ 0.9998 \\ 0.9998 \end{pmatrix}$.

Similarly for 0.000001 with result $x = \begin{pmatrix} 0.998 \\ 0.9981 \\ 0.9981 \\ 0.998 \end{pmatrix}$, 0.00001 $x = \begin{pmatrix} 0.9807 \\ 0.9977 \\ 0.9851 \\ 0.9801 \end{pmatrix}$and

```
0.0001
done! x = [[0.8205]
 [1.02  ]
 [0.9147]
 [0.8603]] diff = 1.0697078378991107
0.001
done! x = [[0.1872]
 [1.2849]
 [0.5617]
 [0.4112]] diff = 0.17676089668443723
0.01
done! x = [[0.0664]
 [1.3367]
 [0.4927]
 [0.3249]] diff = 0.004929551786228874
0.02
done! x = [[0.0664]
 [1.3367]
 [0.4927]
 [0.3249]] diff = 0.0049316867221867585
0.1
done! x = [[0.0663]
 [1.3367]
 [0.4927]
 [0.3249]] diff = 0.004887916821905518
0.15
done! x = [[0.0663]
 [1.3367]
 [0.4927]
 [0.3249]] diff = 0.004897389395280553
```

0.0001 $x = \begin{pmatrix} 0.8025 \\ 1.02 \\ 0.9147 \\ 0.8603 \end{pmatrix}$, these results are far away from the minimum after 10000 steps. The difference between the starting point and the result is small(less than 0.01) so that we should not use learning rate less than 0.001. the difference between $\hat{x}$ and x^k is the smallest at learning rate=0.1 If we use learning rate > 0.1 the MSE would be larger. for alpha=10, we cannot run gradient decent since we would encounter invalid value when subtracting.

(e)After standardizing of each feature, data from each feature follow normal distribution N(0,1) with mean 0 and var 1(shown in the following picture)

	CompPrice	Income	Advertising	Population	Price	Age	Education
0	0.850455	0.155361	0.657177	0.075819	0.177823	-0.699782	1.184449
1	-0.912484	-0.739060	1.409957	-0.032882	-1.386854	0.721723	-1.490113
2	-0.781896	-1.204159	0.506621	0.028262	-1.513719	0.350895	-0.725953
3	-0.520720	1.121336	-0.396715	1.366649	-0.794814	0.103677	0.038208
4	1.046337	-0.166631	-0.547271	0.510625	0.516132	-0.947000	-0.343872
...	...	...	...	...	...	...	...
397	2.417512	-1.526151	0.807733	0.700853	1.827078	-0.823391	1.566529
398	-1.630719	0.370022	0.054953	0.130170	-0.879391	-0.205346	-0.725953
399	0.589279	-1.132606	-0.998939	-1.615848	0.177823	-0.267150	0.802369
mean	0.000000	0.000000	-0.000000	0.000000	-0.000000	0.000000	-0.000000
var	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

After splitting training and testing set, first row of X_train, Y_train is row 0 and last row of X_train is row 199. First row of X_test,Y_test is row 200 and last row of X_test,Y_test is row 399.

	CompPrice	Income	Advertising	Population	Price	Age	Education
0	0.850455	0.155361	0.657177	0.075819	0.177823	-0.699782	1.184449
1	-0.912484	-0.739060	1.409957	-0.032882	-1.386854	0.721723	-1.490113
2	-0.781896	-1.204159	0.506621	0.028262	-1.513719	0.350895	-0.725953
3	-0.520720	1.121336	-0.396715	1.366649	-0.794814	0.103677	0.038208
4	1.046337	-0.166631	-0.547271	0.510625	0.516132	-0.947000	-0.343872
..	...	...	...	...	...	...	...
195	-0.520720	0.870898	-0.396715	1.054133	-0.160485	0.783527	-1.108033
196	0.328103	-1.454597	-0.095603	0.986194	0.727575	1.154355	0.802369
197	-0.063662	-0.273961	-0.998939	0.463068	0.939018	1.401573	0.802369
198	-0.847190	0.405799	-0.246159	1.597640	0.516132	0.968941	-1.490113
199	-0.194250	0.692014	-0.246159	0.476656	0.431555	0.659918	0.038208

[200 rows x 7 columns]

	Sales
0	2.003675
1	3.723675
2	2.563675
3	-0.096325
4	-3.346325
..	...
195	-3.306325
196	-3.396325
197	-4.976325
198	-3.876325
199	-1.076325

[200 rows x 1 columns]

	CompPrice	Income	Advertising	Population	Price	Age	Education
200	1.242219	0.835121	-0.998939	0.571770	1.277326	0.536309	-0.725953
201	0.850455	0.513130	-0.998939	-0.854937	0.769863	0.041873	1.566529
202	-0.259544	0.334245	-0.396715	1.006576	0.600709	-0.452564	-1.490113
203	0.393397	0.477353	-0.998939	-0.902494	1.742501	-1.750460	0.038208
204	1.960454	0.405799	-0.998939	-0.189141	0.346978	-1.008805	0.038208
..	...	...	...	...	...	...	...
395	0.850455	1.407551	1.560513	-0.420131	0.516132	-1.256023	0.038208
396	0.915749	-1.633482	-0.547271	-1.547909	0.177823	0.103677	-1.108033
397	2.417512	-1.526151	0.807733	0.700853	1.827078	-0.823391	1.566529
398	-1.630719	0.370022	0.054953	0.130170	-0.879391	-0.205346	-0.725953
399	0.589279	-1.132606	-0.998939	-1.615848	0.177823	-0.267150	0.802369

[200 rows x 7 columns]

	Sales
200	-1.936325
201	-1.556325
202	-3.396325
203	-5.446325
204	1.243675
..	...
395	5.073675
396	-1.356325
397	-0.086325
398	-1.556325
399	2.213675

[200 rows x 1 columns]

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from numpy.linalg import inv
df = pd.read_csv('CarSeats.csv')
df = df.drop(['ShelveLoc','Urban','US'],axis = 1)
sales = df['Sales']
df.drop('Sales',axis = 1)
features = ['CompPrice','Income','Advertising','Population','Price','Age','Education']
scaler = StandardScaler().fit_transform(df[features])
scaled_features_df = pd.DataFrame(scaler, index=df.index, columns = features)
sales = sales - sales.mean()
sales = pd.DataFrame(sales, index=sales.index)
#scaled_features_df.loc['mean'] = scaled_features_df.mean().round(5)
#scaled_features_df.loc['var'] = scaled_features_df.std().round(5)
pd.set_option('display.width',500)
X_train = scaled_features_df[:200]
Y_train = sales[:200]
X_test = scaled_features_df[200:400]
Y_test =sales[200:400]
print(X_test)
print(Y_test)
```

(f)

Ridge regression place a penalty on the magnitude of the coefficients associated to each variable. And the scale of variables will affect how much penalty will be applied on their coefficients, because coefficients of variables with large variance are small and thus led penalized. Therefore, standardization is required before Ridge regression.

Moreover, without standardization, the feature with a broad range will dominate over the other feature and will have more contribution.

(g)

$$\hat{\beta}^{Ridge} = \arg\min_{\beta} \frac{1}{n} \|y - X\beta\|_2^2 + \phi \|\beta\|_2^2$$

$$\frac{1}{n} \|y - X\beta\|_2^2 + \phi \|\beta\|_2^2$$

$$= \frac{1}{n} (Y - X\beta)^T (Y - X\beta) + \phi \beta^T \beta$$

$$= \frac{1}{n} \left(Y^T Y - 2\beta^T X^T Y + \beta^T X^T X \beta \right) + \phi \beta^T \beta$$

$$\frac{d}{d\beta} = -\frac{2}{n} X^T Y + \frac{2}{n} X^T X \beta + 2\phi \beta \overset{set}{=} 0$$

$$\frac{2}{n} X^T Y = \frac{2}{n} X^T X \beta + 2\phi \beta$$

$$X^T Y = X^T X \beta + n\phi \beta$$

$$\beta = (X^T X + n\phi I)^{-1} X^T Y$$

```
xtx = np.dot(X_train.transpose(),X_train)
iden = 100*np.identity(7)
mul = np.add(xtx, iden)
mul_inv = inv(mul)
last = np.dot(mul_inv,X_train.transpose())
ans = np.dot(last, Y_train)
print('true beta is {a}'.format(a=ans))
```

```
true beta is [[ 1.41276471]
 [ 0.48674513]
 [ 0.8100989 ]
 [ 0.03929668]
 [-2.15319101]
 [-0.63633658]
 [-0.27205856]]
```

(h)

The ridge regression loss is

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

$$= \frac{1}{n}(Y - X\beta)^T(Y - X\beta) + \phi\beta^T\beta$$

$$= \frac{1}{n}\left(Y^TY - 2\beta^TX^TY + \beta^TX^TX\beta\right) + \phi\beta^T\beta$$

$$\frac{d}{d\beta} = -\frac{2}{n}X^TY + \frac{2}{n}X^TX\beta + 2\phi\beta \overset{\text{set}}{=} 0$$

$$= -\frac{2}{n}X^T(Y - X\beta) + 2\phi\beta, \text{ when } n=1 \quad \cdots \textcircled{1}$$

for each $x_i \in X$, since size of β is $n\times 1$, X is $m\times n$.

x_i should be $n\times 1$ vector, so that x_i^T is $1\times n$ to multiply β.

$$\mathcal{L}_i(\beta) = \|y_i - x_i^T\beta\|_2^2 + \phi\|\beta\|_2^2$$

$$\nabla\mathcal{L}_i(\beta) = -2x_i(y_i - x_i^T\beta) + 2\phi\beta \quad \text{(from } \textcircled{1}\text{)}$$

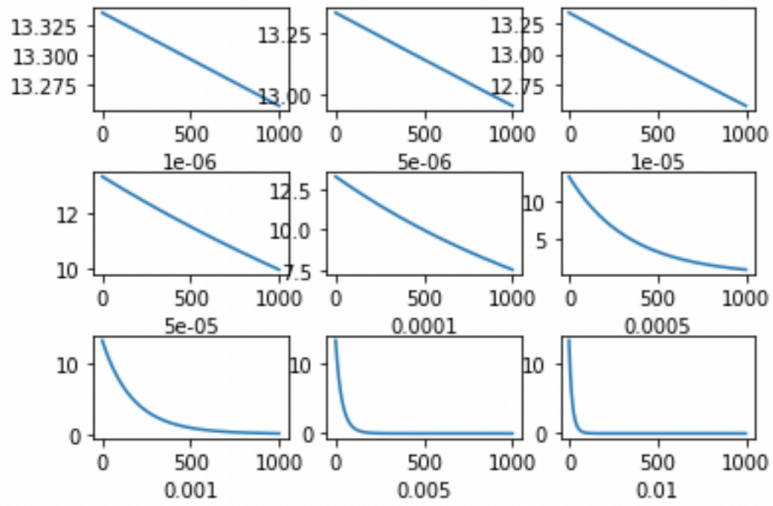
Sum of squared error for m samples:

$$\mathcal{L}(\beta) = \sum_{i=1}^{m}\left(y_i - \sum_{j=1}^{n}x_{ji}\beta_i\right)^2 + \|\beta\|_2^2$$

$$= \sum_{i=1}^{m}(y_i - x_i^T\beta)^2 = (Y - X\beta)^T(Y - X\beta) + \|\beta\|_2^2$$

$$= \sum_{i=1}^{m}\mathcal{L}_i(\beta), \text{ so that } \mathcal{L}(\beta) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_i(\beta)$$

(i)



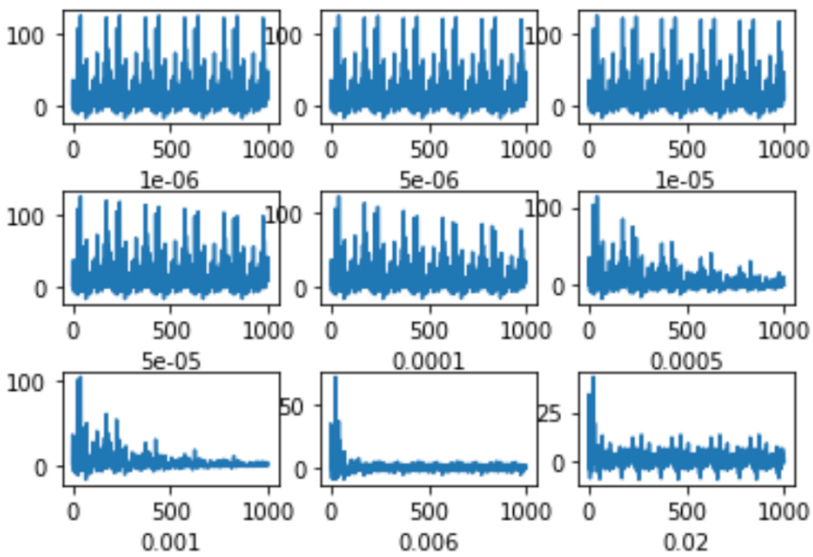
Step-size 0.01 is the best since the β^k achieved from 0.01 is same as $\widehat{\beta}$(closed form solution derived ealier).

```
The train MSE is 4.558906713880709
The test MSE is 4.3804291756539575
```

```
import matplotlib.pyplot as plt
beta = np.array([[1],[1],[1],[1],[1],[1],[1]])
true_beta = np.array ([[0.680673],[0.28229334],[0.65157017],[0.00834835],[-1.17129533],[-0.400892],[-0.10063355]])
lambdas = [0.000001,0.000005,0.00001,0.00005,0.0001,0.0005,0.001,0.005,0.01]
def cal_gradient(X, Y, beta):
    ans1 = -2*np.dot(X.T,Y)
    ans2 = 2*(np.dot(np.dot(X.T,X),beta))
    left = np.add(ans1,ans2)
    res = np.add((1/200)*left, beta)
    return res
def cal_loss(Y, beta, X):
    left = Y-np.dot(X,beta)
    leftnorm = np.linalg.norm(left, ord = 2)
    left2 = leftnorm**2
    right = np.linalg.norm(beta, ord = 2)
    right2 = right**2
    ans = (1/200)*left2 + 0.5*right2
    return ans
def batchgd(X, Y, beta, learning_rate,step):
    loss = []
    for i in range(step):
        lo = cal_loss(Y,beta,X)
        true_loss = cal_loss(Y,true_beta,X)
        res = lo-true_loss
        loss.append(res)
        gradient = cal_gradient(X, Y, beta)
        delta = learning_rate * gradient
        beta = beta - delta
    plt.plot(loss)
    print(loss[-1])
    plt.xlabel(learning_rate)
    return beta
```

```
def all_learning(lambdas):
    L = len(lambdas)
    for i in range(L):
        loss = []
        beta = np.array([[1],[1],[1],[1],[1],[1],[1]])
        plt.subplot(3, 3, i+1)
        batchgd(X_train, Y_train, beta, lambdas[i],1000)
all_learning(lambdas)
plt.subplots_adjust(wspace=0.2,hspace=0.6)
mse = Y_train-np.dot(X_train, true_beta)
mse = np.linalg.norm(mse)
mse = (1/200)*mse**2
mse1 = Y_test-np.dot(X_test, true_beta)
mse1 = np.linalg.norm(mse1)
mse1 = (1/200)*mse1**2
print('The train MSE is {a} '.format(a=mse))
print('The test MSE is {a} '.format(a=mse1))
```

(j)



The best step-size choice is 0.001 with result beta

```
final beta is [[ 0.57555462]
 [ 0.36246031]
 [ 0.64832453]
 [ 0.04874021]
 [-1.03566752]
 [-0.28154239]
 [-0.02666543]]
The train MSE is 4.862309306194937
The test MSE is 4.643536468317737
```

```
import matplotlib.pyplot as plt
beta = np.array([[1],[1],[1],[1],[1],[1],[1]])
true_beta = np.array ([[0.680673],[0.28229334],[0.65157017],[0.00834835],[-1.17129533],[-0.400892],[-0.10063355]])
lambdas = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02]
def cal_gradient(xi, yi, beta):
    ans1 = -2*np.dot(xi,yi)
    ans2 = 2*(np.dot(np.dot(xi,xi.T),beta))
    left = np.add(ans1,ans2)
    res = np.add(left, beta)
    return res
def cal_loss(yi, beta, xi):
    left = yi-np.dot(xi.T,beta)
    leftnorm = np.linalg.norm(left)
    left2 = leftnorm**2
    right = np.linalg.norm(beta)
    right2 = right**2
    ans = left2 + 0.5*right2
    return ans
def sgd(X, Y, beta, learning_rate,step):
    loss = []
    for i in range(step):
        k = i % 200
        xi = X.loc[k]
        xiv = np.mat(xi.values)
        xit = xiv.transpose()
        yi = Y.loc[k]
        yiv = np.mat(yi.values)
        gradient = cal_gradient(xit, yiv, beta)
        lo = cal_loss(yiv,beta,xit)
        true_loss = cal_loss(yiv,true_beta,xit)
        res = lo-true_loss
        loss.append(res)
        delta = learning_rate * gradient
        beta = beta - delta
    print('final beta is {a} '.format(a=beta))
```

```
    print('final beta is {a} '.format(a=beta))
    plt.plot(loss)
    plt.xlabel(learning_rate)
    return beta
def all_learning(lambdas):
    L = len(lambdas)
    for i in range(L):
        loss = []
        beta = np.array([[1],[1],[1],[1],[1],[1],[1]])
        plt.subplot(3, 3, i+1)
        sgd(X_train, Y_train, beta, lambdas[i],1000)
all_learning(lambdas)
#beta = sgd(X_train, Y_train, beta, 0.001, 1000)
#print(beta)
plt.subplots_adjust(wspace=0.2,hspace=0.6)
mse = Y_train-np.dot(X_train, beta)
mse = np.linalg.norm(mse)
mse = (1/200)*mse**2
mse1 = Y_test-np.dot(X_test, beta)
mse1 = np.linalg.norm(mse1)
mse1 = (1/200)*mse1**2
print('The train MSE is {a} '.format(a=mse))
print('The test MSE is {a} '.format(a=mse1))
```

When step-size is small, the value of $\Delta^{(k)}$ jumps up and down since we only consider one xi from X_train instead of the whole X_train dataset. Although the gradient we calculated in each iteration is not directed towards the global optimal

solution, but the general direction is towards the global optimal solution, and the final result is often close to the global optimal solution. compared to batch gradient, such method is faster

(k)I prefer GD for this question. However, when the dataset is relevant small GD is a better idea. If the dataset is large we should use SGD since time complexity for sgd is smaller.