Question1(a)

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

$$\nabla f(x) = \frac{\partial f(x)}{\partial x} = \frac{1}{2}\cdot 2\cdot A^T\cdot(Ax-b) + \frac{y}{2}\cdot 2x$$

$$= A^T(Ax-b) + y\cdot x$$

```python
import numpy as np

A = np.array([[1, 2, 1, -1],
              [-1, 1, 0, 2],
              [0, -1, -2, 1]])
x = np.array([1, 1, 1, 1])
b = np.array([3, 2, -2])

result = np.array([1, 1, 1, 1])
check = 1

while check == 1:
    grad = A.T@(A@x - b) + 0.2*x
    if np.linalg.norm(grad) < 0.001:
        check = 0
    else:
        x = x - 0.1*grad
        result = np.vstack([result, x])
```

```python
result = np.around(result, decimals=4)

for i in range(6):
    print(f'k = {i}, x(k) = {result[i]}')

for j in range(5):
    print(f'k = {len(result) - j - 1}, x(k) = {result[len(result) - j - 1]}')
```

```
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 = 276, x(k) = [0.0663 1.3367 0.4927 0.3249]
k = 275, x(k) = [0.0664 1.3367 0.4927 0.3249]
k = 274, x(k) = [0.0665 1.3366 0.4927 0.325 ]
k = 273, x(k) = [0.0666 1.3366 0.4928 0.325 ]
k = 272, x(k) = [0.0666 1.3366 0.4928 0.3251]
```

Question1(b)

This means that the minimum algorithm converges to three decimal places. If the righthand side smaller (say 0.0001), the output precision will be higher, the first five rows of data will not be affected, the value of K in the last five rows of data will increase, and x(K) will decrease accordingly.

Question1(c)

```
k = 0, x(k) =[0.98, 0.98, 0.98, 0.98]
k = 1, x(k) =[0.9624, 0.9804, 0.9744, 0.9584]
k = 2, x(k) =[0.9427, 0.9824, 0.9668, 0.9433]
k = 3, x(k) =[0.9234, 0.9866, 0.9598, 0.9295]
k = 4, x(k) =[0.9044, 0.9916, 0.9526, 0.9169]
k = 5, x(k) =[0.8861, 0.997, 0.9452, 0.9047]
k = 276, x(k) =[0.0663, 1.3367, 0.4926, 0.3248]
k = 275, x(k) =[0.0663, 1.3367, 0.4927, 0.3249]
k = 274, x(k) =[0.0664, 1.3367, 0.4927, 0.3249]
k = 273, x(k) =[0.0665, 1.3366, 0.4927, 0.325]
k = 272, x(k) =[0.0666, 1.3366, 0.4928, 0.325]
```

```python
import torch
from torch import nn, optim
A = torch.Tensor([[1, 2, 1, -1],
                  [-1, 1, 0, 2],
                  [0, -1, -2, 1]])
b = torch.Tensor([[3], [2], [-2]])
tol = 0.001
gamma = 0.2
alpha = 0.1
x = torch.Tensor([[1], [1], [1], [1]])

class MyModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.x_tensor = nn.Parameter(torch.ones(size=[4,1], requires_grad=True))

    def forward(self, A_tensor):
        return A_tensor@self.x_tensor
```

```python
model = MyModel()
optimizer = optim.SGD(model.parameters(), lr=alpha)
loss_func = nn.MSELoss(reduction='sum')
terminationCond = False
x_list = []
while not terminationCond:
    pred = model(A)
    loss = loss_func(pred, b)/2 + gamma/2*(model.x_tensor.norm(2)**2)
    loss.backward()
    optimizer.step()
    check = model.x_tensor.grad.norm(2)
    optimizer.zero_grad()
    x_list.append(model.x_tensor.data.tolist())
    if check < tol:
        terminationCond = True
```

```python
result = []
for i in range(len(x_list)):
    zs = []
    for j in x_list[i]:
        zs.append(round(j[0],4))
    result.append(zs)

for i in range(6):
    print(f'k = {i}, x(k) =', end='')
    print(result[i])

for j in range(5):
    print(f'k = {len(result) - j - 1}, x(k) =', end='')
    print(result[len(result) - j - 1])
```

Question1(d)

```
Feature Means: [ 3.81916720e-16  3.55271368e-17  2.66453526e-17  1.59872116e-16
 -6.21724894e-17  1.28785871e-16 -1.33226763e-16]
Feature Variances: [1. 1. 1. 1. 1. 1. 1.]
[[-1.76130698  1.76531943 -0.99893918 -1.67019833 -0.371928    0.65991828
   1.18444912]]
[[-1.30424863 -1.31149003 -0.99893918  0.6329144  -0.371928   -1.68865531
  -0.34387232]]
[[ 0.3281026   0.90667493  0.20550897 -0.86852487  0.51613203  0.65991828
  -0.72595268]]
[[ 0.58927879  1.37177403 -0.99893918 -1.09272169 -0.32963943  0.4127
  -0.72595268]]
27   -2.226325
Name: Sales, dtype: float64
61   -0.176325
Name: Sales, dtype: float64
119   -0.126325
Name: Sales, dtype: float64
107    1.053675
Name: Sales, dtype: float64
```

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

cs_df = pd.read_csv("D:/UNSW/2022-T2/9417/homework/CarSeats.csv")
cs_df = cs_df.drop(['ShelveLoc', 'Urban', 'US'], axis=1)
X = cs_df.iloc[:, 1:]
Y = cs_df.iloc[:, 0]
scaler = StandardScaler().fit(X)
scaled_X = scaler.transform(X)
print(f"Feature Means: {scaled_X.mean(axis=0)}")
print(f"Feature Variances: {scaled_X.var(axis=0)}")
Y = Y - Y.mean()
X_train, X_test, Y_train, Y_test = train_test_split(scaled_X, Y, test_size=0.5)
```

```python
print(X_train[0:1])
print(X_train[-2:-1])
print(X_test[0:1])
print(X_test[-2:-1])
print(Y_train[0:1])
print(Y_train[-2:-1])
print(Y_test[0:1])
print(Y_test[-2:-1])
```

Question1(e)

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

$$= (y - X\beta)^T(y - X\beta) + \phi\beta^T\beta$$

$$= y^Ty - y^TX\beta - \beta^TX^Ty + \beta^TX^TX\beta + \lambda\beta^T\beta$$

for $\frac{\partial \hat{\beta}_{Ridge}}{\partial \beta} = 0$

$$\Rightarrow 0 - X^Ty - X^Ty + 2X^TX\beta + 2\lambda\beta = 0$$

$$\Rightarrow \beta = (X^TX + \lambda I)^{-1}X^Ty$$

```
ridge = Ridge(alpha=0.5).fit(X_train, Y_train)
print(ridge.coef_)
```

```
[ 1.44158434  0.25982139  0.89153542 -0.00245964 -2.31787442 -0.70447382
 -0.09819542]
```

Question1(f)

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

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

$$= \frac{1}{n}(\beta^TX^TX\beta - 2\beta^TX^Ty + y^Ty) + \frac{1}{n}(n\phi\beta^T\beta)$$

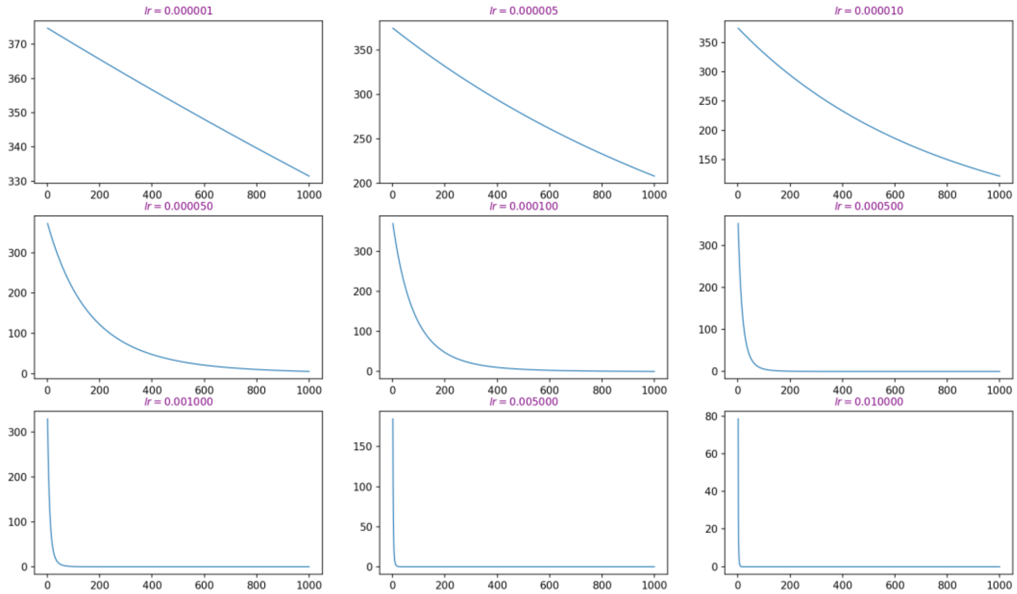
$$= \frac{1}{n}(\beta^TX^TX\beta - 2\beta^TX^Ty + y^Ty)$$

$$= \frac{1}{n}[\beta^T(X^TX + n\phi I)\beta - 2\beta^TX^Ty + y^Ty]$$

$L_1(\beta) = \frac{1}{n}\beta^T(X^TX + n\phi I)\beta$ $\quad L_2(\beta) = -\frac{2}{n}\beta^TX^Ty$ $\quad L_3(\beta) = \frac{1}{n}y^Ty$

$\nabla L_1(\beta) = \frac{2}{n}X^TX\beta + 2\phi I\beta$ $\quad \nabla L_2(\beta) = -\frac{2}{n}X^Ty$ $\quad \nabla L_3(\beta) = 0$

Question1(g)



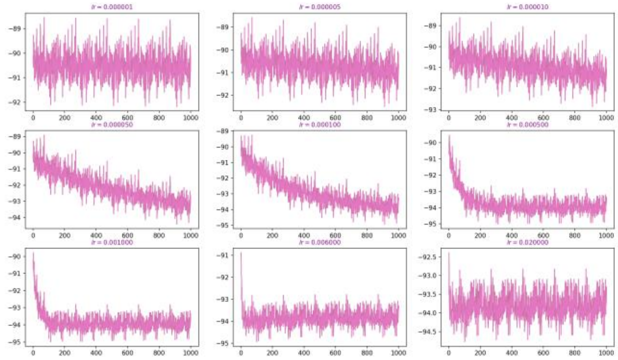
In my opinion, the progress is best when the step size is 0.005. As can be seen from the image, when the step size is 0.005, the value gradually converges and finally becomes stable, and the accuracy obtained will be higher than that of other step sizes.

```
The train MSE: 3.175015
The test MSE: 4.440754
```

```python
lr = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01]
result = []
Y_train_a = Y_train.values
beta = np.ones((X_train.shape[1], 1))
for i in range(len(lr)):
    process = []
    beta = np.ones((X_train.shape[1], 1))
    for j in range(1000):
        grad1 = (X_train.T @ X_train + X_train.shape[1] * 0.5 * np.eye(X_train.shape[1])) @ beta
        grad2 = X_train.T @ Y_train_a
        grad = (grad1.T - grad2)*2/X_train.shape[1]
        beta = beta - grad.T*lr[i]
        l = (beta.T@X_train.T@ X_train@ beta - 2*beta.T@X_train.T@ Y_train + Y_train.T@Y_train +
             X_train.shape[1]* 0.5*beta.T@beta)/X_train.shape[1]
        s = (l - ridge.coef_).tolist()[0]
        process.append(s)
```

```python
        if i == 5 and j == 999:
            beta_GD = beta
    process = np.array(process)
    result.append(process.T)
result = np.array(result)
dot = np.linspace(1, 1000, num=1000)
for i in range(9):
    plt.subplot(3, 3, i + 1)
    for j in range(7):
        plt.plot(dot, result[i, j], linewidth=1)
        plt.title(f"$lr={lr[i]:f}$", fontsize=10, loc='center', color='purple')
plt.show()
print('The train MSE: %f' % mean_squared_error(Y_train, X_train @ beta_GD))
print('The test MSE: %f' % mean_squared_error(Y_test, X_test @ beta_GD))
```

Question1(h)



the best step-size choice:0.02

```
The train MSE: 5.612344
The test MSE: 7.010576
```

```python
# question(h)
lr = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02]
beta = np.ones((X_train.shape[1], 1))
result = []
for i in range(len(lr)):
    process = []
    beta = np.ones((X_train.shape[1], 1))
    for k in range(5):
        for j in range(len(X_train)):
            batch = X_train[j]
            batch_y = Y_train_a[j]
            grad1 = (batch.T @ batch + X_train.shape[1] * 0.5 * np.eye(X_train.shape[1])) @ beta
            grad2 = batch.T * batch_y
            grad = (grad1.T - grad2) * 2 / X_train.shape[1]
            beta = beta - grad.T * lr[i]
            los = (batch_y-batch@ beta)/ X_train.shape[1] + 0.5 * beta.T @ beta
            s = (los - ridge.coef_).tolist()[0]
            process.append(s)
            if i == 8 and k == 4 and j == 199:
                beta_SGD = beta
    process = np.array(process)
    result.append(process.T)
result = np.array(result)
for i in range(9):
    plt.subplot(3, 3, i + 1)
    for j in range(7):
        plt.plot(dot, result[i, j], linewidth=1)
        plt.title(f"$lr={lr[i]:f}$", fontsize=10, loc='center', color='purple')
plt.show()
print('The train MSE: %f' % mean_squared_error(Y_train, X_train @ beta_SGD))
print('The test MSE: %f' % mean_squared_error(Y_test, X_test @ beta_SGD))
```

Stochastic gradient descent may not go in the right direction each time it is updated, thus causing optimization fluctuations.

Question1(i)

I prefer GD because the MSE for GD is smaller and more accurate in this data set.
An advantage of the fluctuation caused by stochastic gradient descent is that for non-convex functions, the optimization direction may jump from the current local minimum point to another better local minimum point, and eventually converge to a better local extremum point, or even the global extremum point. That is, use GD for convex functions and SGD for non-convex functions.

Question1(j)

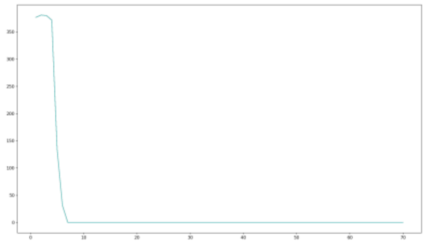
$$\hat{\beta}_1 = \operatorname{argmin} L(\beta_1, \beta_2, \cdots, \beta_p)$$

$$\hat{\beta}_2 = \operatorname{argmin} L(\hat{\beta}_1, \beta_2, \cdots, \beta_p)$$

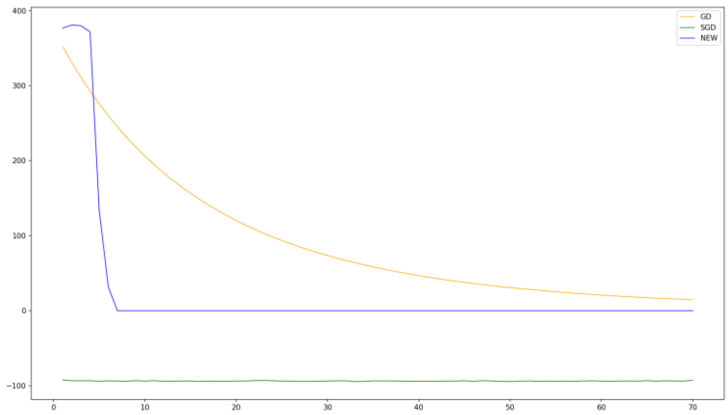
$$\hat{\beta}_3 = \operatorname{argmin} L(\hat{\beta}_1, \hat{\beta}_2, \beta_3, \cdots, \beta_p)$$

$$\hat{\beta}_j = \operatorname{argmin} L(\hat{\beta}_1, \cdots \hat{\beta}_{j-1}, \beta_j, \cdots \beta_p)$$

Question1(k)



```
The train MSE: 3.175013
The test MSE: 4.440733
```




```python
# question(k)
beta = np.ones((X_train.shape[1], 1))
result3 = []
process = []
for i in range(10):
    for j in range(X_train.shape[1]):
        beta_new = np.linalg.inv(X_train.T@X_train+ X_train.shape[1]*0.5*np.eye(X_train.shape[1]))
        beta[j] = beta_new[j]
        los = (beta.T @ X_train.T @ X_train @ beta - 2 * beta.T @ X_train.T @ Y_train
               + Y_train.T @ Y_train +X_train.shape[1] * 0.5 * beta.T @ beta) / X_train.shape[1]
        s = (los - los0).tolist()[0]
        result3.append(s)
result3 = np.array(result3)
dot2 = np.linspace(1, 70, num=70)
for j in range(70):
    plt.plot(dot2, result3, linewidth=1)
plt.show()
print('The train MSE: %f' % mean_squared_error(Y_train, X_train @ beta))
print('The test MSE: %f' % mean_squared_error(Y_test, X_test @ beta))
result_GD = result1[5, 0:70]
result_SGD = result2[8, 0:70]
result_NEW = result3
plt.plot(dot2, result_GD, linewidth=1, color='orange', label='GD')
plt.plot(dot2, result_SGD, linewidth=1, color='green', label='SGD')
plt.plot(dot2, result_NEW, linewidth=1, color='blue', label='NEW')
plt.legend(loc=0)
plt.show()
```

Question1(l)

My results in parts (e)-(k) are more reliable, because the features in this dataset basically follow gaussian normal distribution.