(a)

We have (derived from lecture slides)

$$\nabla f\left(x^{(k)}\right) = \frac{1}{2}\left(-2A^T\left(Ax^{(k)} - b\right) + 2\gamma x^{(k)}\right)$$
$$= A^T Ax^{(k)} - A^T b + \gamma x^{(k)},$$

thus, the batch gradient step equation is:

$$x^{(k+1)} = x^{(k)} - \nabla f\left(x^{(k)}\right),$$
$$= x^{(k)} - \alpha\left(A^T Ax^{(k)} - A^T b + \gamma x^{(k)}\right),$$

where α, γ, A, b are given in the question.

```
import numpy as np

answers = []

A = np.array([[1,2,1,-1],[-1,1,0,2],[0,-1,-2,1]])
A_transpose = A.transpose()
b = np.array([3,2,-2])
gamma = 0.2
alpha = 0.1

def gradient(x):
    return ((A_transpose @ A @ x) - (A_transpose @ b) + (gamma * x))

start = np.array([1,1,1,1])
x = start
answers.append(np.round_(x.copy(), decimals=4))
gradient_at_x = gradient(x)

while np.linalg.norm(gradient_at_x) >= 0.001:
    x = x - alpha * gradient_at_x
    answers.append(np.round_(x.copy(), decimals=4))
    gradient_at_x = gradient(x)

print(answers[:5])
print(answers[-5:])
```

With a start at $x = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$, we get the following:

$$k = 0, x = [1,1,1,1]$$
$$k = 1, x = [0.98,0.98,0.98,0.98]$$
$$k = 2, x = [0.9624,0.9804,0.9744,0.9584]$$
$$k = 3, x = [0.9427,0.9824,0.9668,0.9433]$$
$$k = 4, x = [0.9234,0.9866,0.9598,0.9295]$$

$$k = n - 4, x = [0.0666,1.3366,0.4928,0.3251]$$
$$k = n - 3, x = [0.0666,1.3366,0.4928,0.325]$$
$$k = n - 2, x = [0.0665,1.3366,0.4927,0.325]$$
$$k = n - 1, x = [0.0664,1.3367,0.4927,0.3249]$$
$$k = n, x = [0.0663,1.3367,0.4927,0.3249]$$

(b)
The termination condition is such that the algorithm stops when the derivative of the multivariate function is sufficiently close to 0, or in other words close to a critical point. We know the function f has a global minimum which the algorithm is approaching. If we make the RHS of the termination condition smaller, we make the termination stricter and so we terminate closer to the global minimum.

(c)

The modified loss equation was supplied to Pytorch and differentiated by their autodiff. Their inbuilt stochastic gradient descent was used (https://pytorch.org/docs/stable/generated/torch.optim.SGD.html) which has the gradient step equation

$$x^{(k+1)} = x^k - \alpha\nabla\left(a_{i_{(k)}}x^{(k)} - b_{i_{(k)}}\right),$$

where i is an index randomly chosen from $\{1, \dots, n\}$, n is the size of the training input matrix A, and $a_{i_{(k)}}$ and $b_{i_{(k)}}$ represent the input and output values at index i.

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


result = []


A = torch.tensor([[1,2,1,-1],[-1,1,0,2],[0,-1,-2,1]]).float()
b = torch.tensor([3,2,-2]).float()
gamma = 0.2
alpha = 0.1


class MyModel(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        self.x = nn.Parameter(torch.ones(A.shape[1], requires_grad=True))

    def forward(self, M):
        return M @ self.x

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

result.append(model.x.data.detach().clone())

terminationCond = False
while not terminationCond:
    bhat = model.forward(A)
    regularisation_term = (gamma / 2) * torch.linalg.norm(model.x, ord=2) ** 2
    loss = (1 / 2) * torch.linalg.norm(bhat - b) ** 2 + regularisation_term
    loss.backward()
    optimizer.step()

    result.append(model.x.data.detach().clone())

    if torch.linalg.norm(model.x.grad) < 0.001:
        terminationCond = True

    optimizer.zero_grad()

print(result[:5])
print(result[-5:])
```

With a start at $x = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$, we get the following:

$$k = 0, x = [1,1,1,1]$$
$$k = 1, x = [0.98,0.98,0.98,0.98]$$
$$k = 2, x = [0.9624,0.9804,0.9744,0.9584]$$
$$k = 3, x = [0.9427,0.9824,0.9668,0.9433]$$
$$k = 4, x = [0.9234,0.9866,0.9598,0.9295]$$

$$k = n - 4, x = [0.0666,1.3366,0.4928,0.325]$$
$$k = n - 3, x = [0.0665,1.3366,0.4927,0.325]$$
$$k = n - 2, x = [0.0664,1.3367,0.4927,0.3249]$$
$$k = n - 1, x = [0.0663,1.3367,0.4927,0.3249]$$

$$k = n, x = [0.0663, 1.3367, 0.4926, 0.3248]$$

(d)

After removing categorical features, we have the remaining features.

	CompPrice	Income	Advertising	Population	Price	Age	Education
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

Listed in order of the columns above,

X_train first 3:

```
[[ 0.85045499  0.15536099  0.65717702  0.07581929  0.17782345 -0.69978222
   1.18444912]
[-0.91248434 -0.73906037  1.40995711 -0.0328822  -1.38685375  0.72172284
  -1.4901134 ]
[-0.78189624 -1.20415947  0.506621    0.02826239 -1.51371947  0.35089544
  -0.72595268]]
```

X_train last 3:

```
[[-0.0636617  -0.27396126 -0.99893918  0.46306833  0.93901776  1.40157309
   0.80236876]
 [-0.84719029  0.40579897 -0.24615909  1.59764009  0.51613203  0.96894112
  -1.4901134 ]
 [-0.1942498   0.6920138  -0.24615909  0.47665602  0.43155489  0.65991828
   0.03820804]]
```

X_test first 3:

```
[[ 1.24221929  0.83512122 -0.99893918  0.57176982  1.27732635  0.53630914
  -0.72595268]
 [ 0.85045499  0.51312953 -0.99893918 -0.85493719  0.76986347  0.04187259
   1.56652948]
 [-0.25954385  0.33424526 -0.39671511  1.00657576  0.60070918 -0.45256395
  -1.4901134 ]]
```

X_test last 3:

```
[[ 2.41751217 -1.52615116  0.80773304  0.70085283  1.82707779 -0.82339136
   1.56652948]
 [-1.63071888  0.37002211  0.05495295  0.13017003 -0.87939087 -0.20534568
  -0.72595268]
 [ 0.58927879 -1.13260576 -0.99893918 -1.61584759  0.17782345 -0.26715025
   0.80236876]]
```

Y_train first 3:

2.003675

3.723675
2.563675

Y_train last 3:
-4.976325
-3.876325
-1.076325

Y_test first 3:
-1.936325
-1.556325
-3.396325

Y_test last 3:
-0.086325
-1.556325
2.213675

```
        import pandas as pd
from sklearn.preprocessing import StandardScaler

features = pd.read_csv('features.csv')

scaler = StandardScaler()
features = scaler.fit_transform(features)

print(scaler.mean_)
print(scaler.var_)

target = pd.read_csv('target.csv')

target = target - target.mean()

X_train = features[:features.shape[0]//2]
X_test = features[features.shape[0]//2:]
Y_train = target[:target.shape[0]//2]
Y_test = target[target.shape[0]//2:]

print(X_train[0:3])
print(X_train[-3:])
print(X_test[0:3])
print(X_test[-3:])
```

```
print(Y_train[0:3])
print(Y_train[-3:])
print(Y_test[0:3])
print(Y_test[-3:])
```

(e)

The closed form solution for ridge regression is (derived from lecture slides)

$$\hat{\beta}_{(Ridge)} = (X^T X + \phi I)^{(-1)} X^T y,$$

with $\phi = 0.5$.

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler

features = pd.read_csv('features.csv')

scaler = StandardScaler()
features = scaler.fit_transform(features)

target = pd.read_csv('target.csv')

target = target - target.mean()

X_train = features[:features.shape[0]//2]
X_test = features[features.shape[0]//2:]
Y_train = target[:target.shape[0]//2]
Y_test = target[target.shape[0]//2:]

phi = 0.5

ridge_regression_weights = np.linalg.inv(X_train.transpose() @ X_train + phi * np.eye(X_train.shape[1])) @
X_train.transpose() @ Y_train
print(ridge_regression_weights)
```

The optimal weights are evaluated to be

$$\hat{\beta}_{(Ridge)} = \begin{pmatrix} 1.674911 \\ 0.368707 \\ 1.109761 \\ 0.020805 \\ -2.321392 \\ -0.519396 \\ -0.149282 \end{pmatrix}$$

Where $\hat{\beta}^{(i)}{}_{(Ridge)}$ corresponds with the i^{th} column feature in the table above.

(f)

We have

$$\begin{aligned} L(\beta) &= \frac{1}{n}||y - X\beta||_2^2 + \phi||\beta||_2^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}((y_i - x_i^T\beta)^2) + \phi(\beta^T\beta), \end{aligned}$$

where x_i is the vector corresponding to the i^{th} row in X.

Therefore,

$$\begin{aligned} L_i(\beta) &= (y_i - x_i^T\beta)^2 \\ &= y_i^2 - 2y_i(x_i^T\beta) + (x_i^T\beta)^2, \end{aligned}$$

Thus, we can deduce that

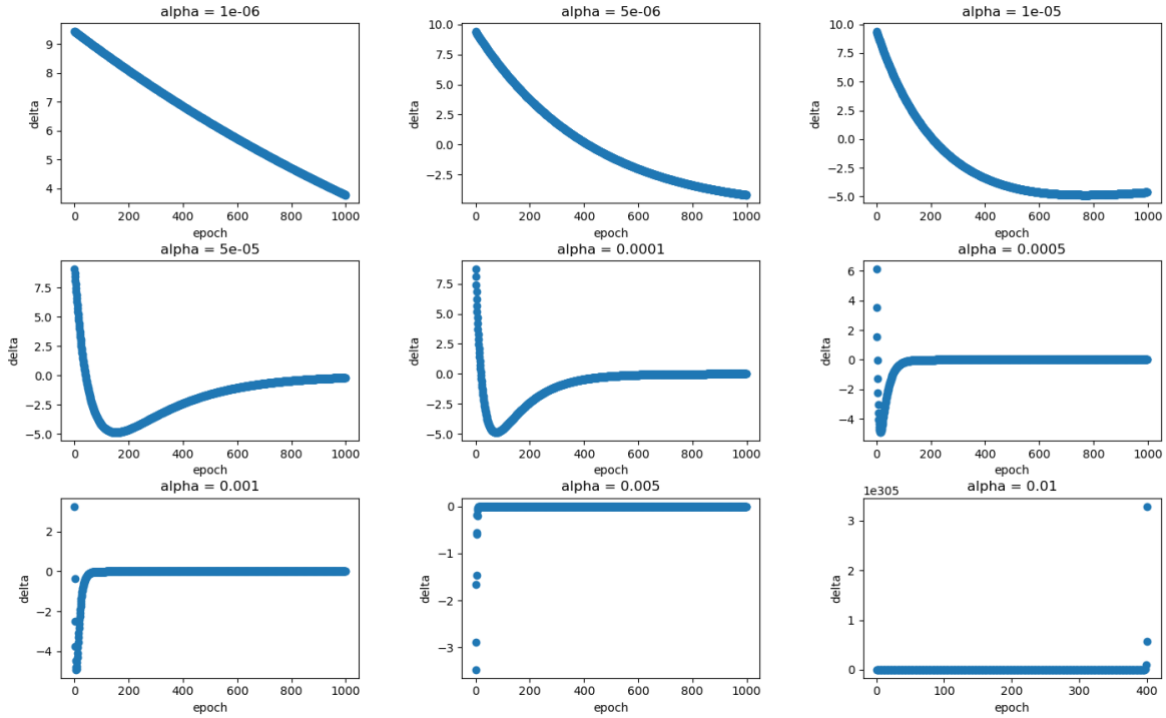
$$\nabla L_i(\beta) = 2\left((x_i^T\beta)x_i - y_i x_i\right).$$

(g)

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

features = pd.read_csv('features.csv')

scaler = StandardScaler()
features = scaler.fit_transform(features)
```

```
target = pd.read_csv('target.csv')

target = target - target.mean()

X_train = pd.DataFrame(features[:features.shape[0]//2]).to_numpy()
X_test = pd.DataFrame(features[features.shape[0]//2:]).to_numpy()
Y_train = pd.DataFrame(target[:target.shape[0]//2]).to_numpy()
Y_test = pd.DataFrame(target[target.shape[0]//2:]).to_numpy()
Y_train = np.reshape(Y_train, -1)
Y_test = np.reshape(Y_test, -1)

X_train_transpose = X_train.transpose()

fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(10,10))
fig.tight_layout()

phi = 0.5
n = X_train.shape[0]
p = X_train.shape[1]

betahat = np.linalg.inv(X_train_transpose @ X_train + phi * np.eye(p)) @ X_train_transpose @ Y_train

def loss_func(beta):
    return (1/n) * (np.linalg.norm(Y_train - X_train @ beta, ord=2) ** 2) + (np.linalg.norm(beta, ord=2)**2)

def gradient(beta):
    return ((X_train_transpose @ X_train @ beta) - (X_train_transpose @ Y_train) + (phi * beta))

def MSE_train(beta):
    return (1/n) * (np.linalg.norm(Y_train - X_train @ beta, ord=2) ** 2)

def MSE_test(beta):
    return (1/n) * (np.linalg.norm(Y_test - X_test @ beta, ord=2) ** 2)

r_plot = 0
c_plot = 0

for alpha in [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005, 0.01]:
    indexes = np.arange(1000)
    deltas = np.full(1000, -1.0)
```

```
    start = np.ones(p)
    beta_k = start
    gradient_beta = gradient(beta_k)

    for i in range(1000):
        beta_k = beta_k - alpha * gradient_beta

        delta_k = loss_func(beta_k) - loss_func(betahat)

        deltas[i] = delta_k

        gradient_beta = gradient(beta_k)

    if alpha == 0.0005:
        print(MSE_train(beta_k))
        print(MSE_test(beta_k))

    axes[r_plot, c_plot].scatter(indexes, deltas)
    axes[r_plot, c_plot].set_title(f'alpha = {alpha}')
    axes[r_plot, c_plot].set_xlabel('epoch')
    axes[r_plot, c_plot].set_ylabel('delta')

    if c_plot >= 2:
        r_plot += 1
        c_plot = 0
    else:
        c_plot += 1

plt.show()
```



Note: encountered overflow errors for alpha = 0.01 at around 400epoch.

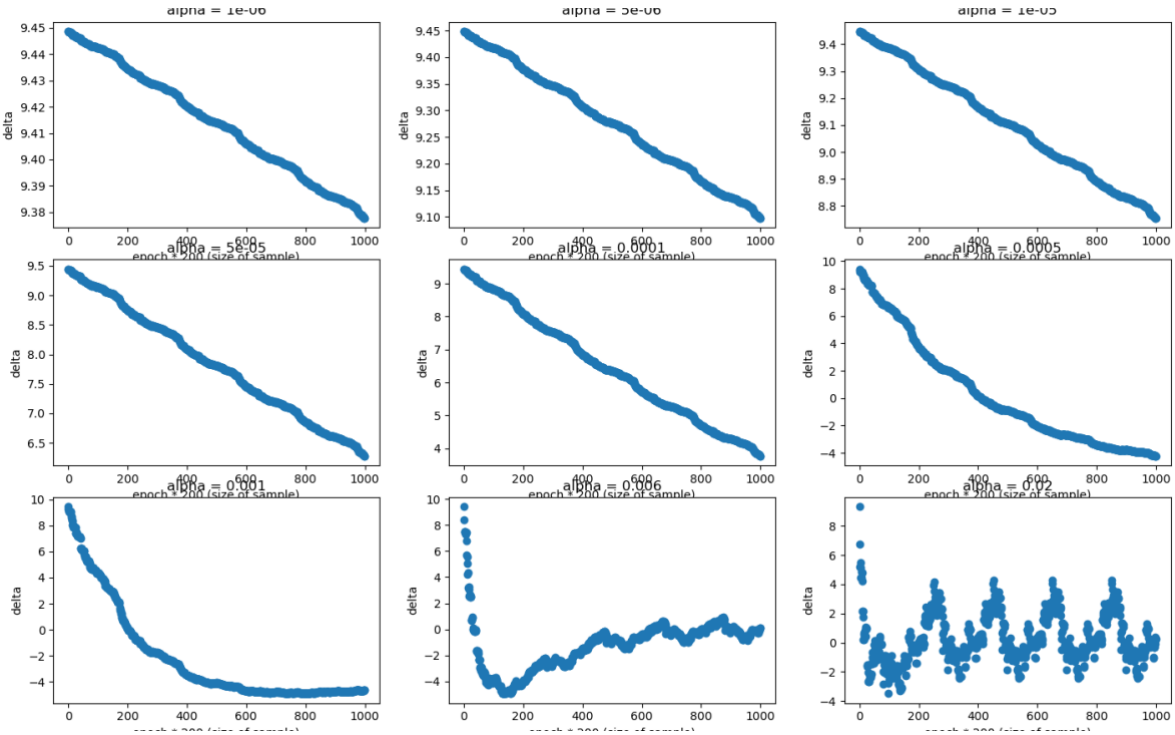
Alpha = 0.0001, 0.0005, 0.001, 0.005 all converges to 0 within 1000epochs. Among those we will pick Alpha = 0.0005 as our best step size. For such alpha, we use the corresponding β to evaluate MSE_train = 3.3468964325141797 and MSE_test = 4.2154164959610565

(h)

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

features = pd.read_csv('features.csv')

scaler = StandardScaler()
features = scaler.fit_transform(features)

target = pd.read_csv('target.csv')

target = target - target.mean()

X_train = pd.DataFrame(features[:features.shape[0]//2]).to_numpy()
X_test = pd.DataFrame(features[features.shape[0]//2:]).to_numpy()
Y_train = pd.DataFrame(target[:target.shape[0]//2]).to_numpy()
Y_test = pd.DataFrame(target[target.shape[0]//2:]).to_numpy()
Y_train = np.reshape(Y_train, -1)
Y_test = np.reshape(Y_test, -1)

X_train_transpose = X_train.transpose()

fig, axes = plt.subplots(nrows=3, ncols=3, figsize=(10,10))
fig.tight_layout()

phi = 0.5
n = X_train.shape[0]
p = X_train.shape[1]

betahat = np.linalg.inv(X_train_transpose @ X_train + phi * np.eye(p)) @ X_train_transpose @ Y_train

def loss_func(beta):
    return (1/n) * (np.linalg.norm(Y_train - X_train @ beta, ord=2) ** 2) + (np.linalg.norm(beta, ord=2) ** 2)

def gradient_i(beta, i):
    return 2 * ((np.dot(X_train[i], beta) * X_train[i]) - (Y_train[i] * X_train[i]) + (1/n) * phi * beta)

def MSE_train(beta):
```

```
    return (1/n) * (np.linalg.norm(Y_train - X_train @ beta, ord=2) ** 2)

def MSE_test(beta):
    return (1/n) * (np.linalg.norm(Y_test - X_test @ beta, ord=2) ** 2)

r_plot = 0
c_plot = 0

epochs = 5*n

results = []

for alpha in [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006, 0.02]:
    indexes = np.arange(epochs)
    deltas = np.full(epochs, -1.0)

    start = np.ones(p)
    beta_k = start

    for i in range(0, epochs):
        gradient_beta_i = gradient_i(beta_k, i % n)
        beta_k = beta_k - alpha * gradient_beta_i

        delta_k = loss_func(beta_k) - loss_func(betahat)
        deltas[i] = delta_k

    if alpha == 0.006:
        print(MSE_train(beta_k))
        print(MSE_test(beta_k))

    axes[r_plot, c_plot].scatter(indexes, deltas)
    axes[r_plot, c_plot].set_title(f'alpha = {alpha}')
    axes[r_plot, c_plot].set_xlabel('epoch')
    axes[r_plot, c_plot].set_ylabel('delta')

    if c_plot >= 2:
        r_plot += 1
        c_plot = 0
    else:
        c_plot += 1
```

```
plt.show()
```



Only alpha = 0.006, 0.02 can be seen visibly approaching 0 at 5 epochs. Among those two we will pick Alpha = 0.006 as our best step size. For this alpha, we use the corresponding β to evaluate MSE_train = 3.408591829591856 and MSE_test = 4.335519230967411.

For alpha = 0.006 and 0.02 we can see most dramatically an oscillating pattern which is not present when analysing batch gradient descent. This is most likely due to the fact that there is a lot more randomness involved with stochastic gradient descent which only takes in one input pair at each step, compared to the whole dataset for batch gradient descent.

(i)

For this example, it is better to use GD over SGD due to the fact it is more 'stable' and able to properly converge to minimum in under the set maximum steps. However, SGD is much more appropriate when we have very large datasets as it is able to 'converge' faster (it more or less converged in 5 epochs compared to 1000 in this example), whilst GD will take a much longer time.

(l)

Standardizing the dataset helps improve reliability when we are applying ridge regression. We want to scale every feature to the same size so that larger feature value ranges will not dominate the weights. Splitting the dataset will reduce reliability (due to the nature of a smaller sample size), however it is a necessary step if we wish to test our model.