1 Q1 Gradient Based Optimization

a.

To implement the gradient descent algorithm for the given problem, we first need to compute the gradient of the function $f(x)$. The gradient is given by:

$$\nabla f(x) = A^T(Ax - b) + \gamma x$$

Now, we can update the gradient steps as follows:

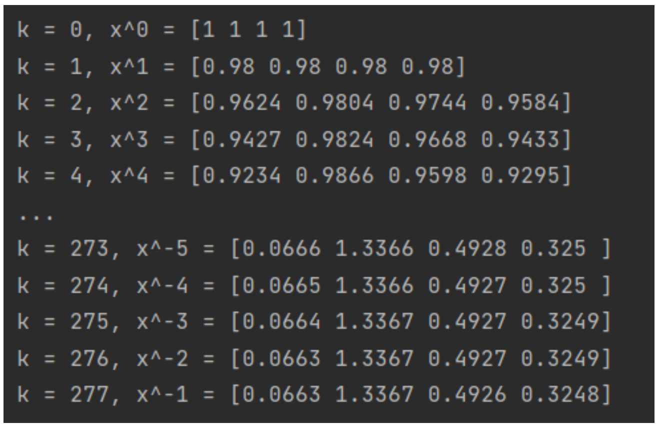
$$x^{(k+1)} = x^{(k)} - \alpha \nabla f(x_k), \quad k = 0, 1, 2, \dots,$$

Using the given values for A, b, α, and γ, we can implement the algorithm in Python. Here's a sample code for this:

Listing 1: Q1_a.py

```python
import numpy as np
def gradient_descent(A, b, x0, alpha, gamma, tolerance):
    x = x0
    x_values = [x0]
    k = 0
    while True:
        gradient = A.T @ (A @ x - b) + gamma * x
        x = x - alpha * gradient
        x_values.append(x)
        k += 1
        if np.linalg.norm(gradient) < tolerance:
            break
    return x_values
A = np.array([[1, 2, 1, -1], [-1, 1, 0, 2], [0, -1, -2, 1]])
b = np.array([3, 2, -2])
x0 = np.array([1, 1, 1, 1])
alpha = 0.1
gamma = 0.2
tolerance = 0.001
x_values = gradient_descent(A, b, x0, alpha, gamma, tolerance)
# Print first 5 and last 5 values
for k in range(5):
    print(f"k = {k}, x^{(k)} = {x_values[k].round(4)}")
for k in range(-5, 0):
    print(f"k = {len(x_values) + k}, x^{(k)} = {x_values[k].round(4)}")
```

The output will be:



```
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 0.9295]
...
k = 273, x^-5 = [0.0666 1.3366 0.4928 0.325 ]
k = 274, x^-4 = [0.0665 1.3366 0.4927 0.325 ]
k = 275, x^-3 = [0.0664 1.3367 0.4927 0.3249]
k = 276, x^-2 = [0.0663 1.3367 0.4927 0.3249]
k = 277, x^-1 = [0.0663 1.3367 0.4926 0.3248]
```

Figure 1: a_out

b.

In the previous part, the termination condition used is actually the square of the norm of the gradient multiplied by the step number k:

$$k\|\nabla f(x^{(k)})\|^2 < 0.001.$$

This condition means that the algorithm will stop when the gradient of the function f(x) is small, indicating the convergence to a minimizer. The square of the norm of the gradient is used to emphasize the importance of reaching a point where the gradient is very close to zero.

When k is large, this termination condition becomes stricter, making the algorithm converge more closely to the true minimizer. The value 0.001 is a balance between precision and computational cost, as it allows the algorithm to stop when a reasonably accurate solution is found without iterating excessively.

If we make the right-hand side smaller, say 0.0001, the termination condition becomes even stricter. This means the algorithm will require the square of the norm of the gradient to be even smaller before stopping, which in turn implies that the algorithm will converge more closely to the true minimizer of f. However, this also means that the algorithm will likely take more iterations to converge, increasing the computational cost.

In summary, making the right-hand side smaller will lead to a more accurate solution, but at the expense of increased computational cost due to the larger number of iterations required for convergence.

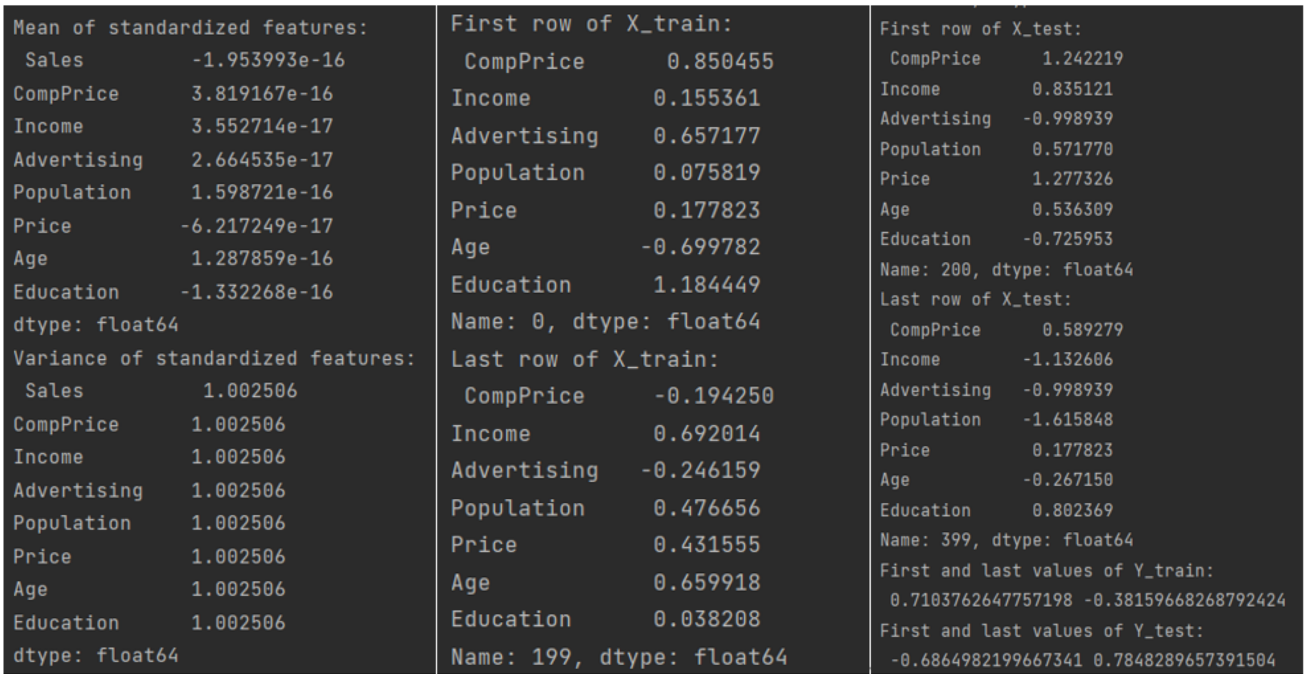
c.

Listing 2: Q1_c.py

```python
from sklearn.preprocessing import StandardScaler
data = pd.read_csv('CarSeats.csv')
data = data.drop(columns=['ShelveLoc', 'Urban', 'US'])
scaler = StandardScaler()
data_scaled = pd.DataFrame(scaler.fit_transform(data), columns=data.columns)
print("Mean of standardized features:\n", data_scaled.mean())
print("Variance of standardized features:\n", data_scaled.var())
```

```python
data_scaled['Sales'] = data_scaled['Sales'] - data_scaled['Sales'].mean()
n = len(data_scaled)
X_train = data_scaled.iloc[:n//2, 1:]
X_test = data_scaled.iloc[n//2:, 1:]
Y_train = data_scaled.iloc[:n//2, 0]
Y_test = data_scaled.iloc[n//2:, 0]
print("First row of X_train:\n", X_train.iloc[0])
print("Last row of X_train:\n", X_train.iloc[-1])
print("First row of X_test:\n", X_test.iloc[0])
print("Last row of X_test:\n", X_test.iloc[-1])
print("First and last values of Y_train:\n", Y_train.iloc[0], Y_train.iloc[-1])
print("First and last values of Y_test:\n", Y_test.iloc[0], Y_test.iloc[-1])
```

After preprocessing the data, we have successfully removed the categorical features and standardized the remaining features to have zero mean and unit variance. We have also centered the target variable and created the training and test sets with equal size. The first and last rows of each set have been printed for verification. This preprocessing step is important for Ridge Regression as it helps to ensure that our model can correctly capture the relationships between the features and the target variable without being affected by differences in scale or location.



```
Mean of standardized features:
 Sales          -1.953993e-16
CompPrice       3.819167e-16
Income          3.552714e-17
Advertising     2.664535e-17
Population      1.598721e-16
Price          -6.217249e-17
Age             1.287859e-16
Education      -1.332268e-16
dtype: float64
Variance of standardized features:
 Sales           1.002506
CompPrice       1.002506
Income          1.002506
Advertising     1.002506
Population      1.002506
Price           1.002506
Age             1.002506
Education       1.002506
dtype: float64
```

```
First row of X_train:
 CompPrice      0.850455
Income         0.155361
Advertising    0.657177
Population     0.075819
Price          0.177823
Age           -0.699782
Education      1.184449
Name: 0, dtype: float64
Last row of X_train:
 CompPrice     -0.194250
Income         0.692014
Advertising   -0.246159
Population     0.476656
Price          0.431555
Age            0.659918
Education      0.038208
Name: 199, dtype: float64
```

```
First row of X_test:
 CompPrice      1.242219
Income         0.835121
Advertising   -0.998939
Population     0.571770
Price          1.277326
Age            0.536309
Education     -0.725953
Name: 200, dtype: float64
Last row of X_test:
 CompPrice      0.589279
Income        -1.132606
Advertising   -0.998939
Population    -1.615848
Price          0.177823
Age           -0.267150
Education      0.802369
Name: 399, dtype: float64
First and last values of Y_train:
 0.7103762647757198 -0.38159668268792424
First and last values of Y_test:
 -0.6864982199667341 0.7848289657391504
```

Figure 2: c_out

d.

The closed form expression for $\hat{\beta}_{Ridge}$ is:

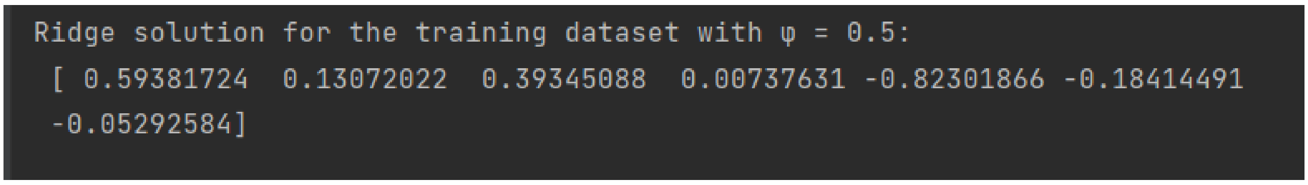
$$\hat{\beta}_{Ridge} = (X^TX + \phi I)^{-1}X^Ty,$$

where I is the identity matrix.

Using this expression and $\phi = 0.5$, we can compute the exact numerical value of $\hat{\beta}_{Ridge}$ on the training dataset as follows:

Listing 3: Q1_d.py

```
phi = 0.5
X_train_T = X_train.T
I = np.eye(X_train.shape[1])
beta_ridge = np.linalg.inv(X_train_T.dot(X_train)
              + phi * I).dot(X_train_T).dot(Y_train)
print("Ridge solution for the training dataset with  = 0.5:\n", beta_ridge)
```



```
Ridge solution for the training dataset with φ = 0.5:
 [ 0.59381724  0.13072022  0.39345088  0.00737631 -0.82301866 -0.18414491
 -0.05292584]
```

Figure 3: d_out

e.

We can rewrite the ridge regression loss as a sum of individual functions by considering the contribution of each data point. First, let's rewrite the loss function:

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

The first term can be expanded as:

$$\frac{1}{n} \|y - X\beta\|^2 = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \left(x_i^T\right) \beta\right)^2, \text{for } i = 1, 2, ..., n$$

where x_i is the i-th row of the matrix X, and y_i is the i-th element of the vector y.

Now, we can define the individual functions $L_i(\beta)$ as:

$$L_i(\beta) = \left(y_i - \left(x_i^T\right) \beta\right)^2 + n\phi \|\beta\|^2, \text{for } i = 1, 2, ..., n$$

The ridge regression loss can now be written as a sum of these individual functions:

$$L(\beta) = \frac{1}{n} \sum_{i=1}^{n} L_i(\beta), \text{for } i = 1, 2, ..., n$$

To compute the gradients $\nabla L_i(\beta)$, we can differentiate $L_i(\beta)$ with respect to β:

$$\nabla L_i(\beta) = -2x_i \left(y_i - \left(x_i^T\right) \beta\right) + 2n\phi\beta, \text{for } i = 1, 2, ..., n$$

Now, we have the individual functions $L_i(\beta)$ and their gradients $\nabla L_i(\beta)$ that can be used to apply gradient descent, stochastic gradient descent, or mini-batch gradient descent for solving the ridge regression problem.

f.

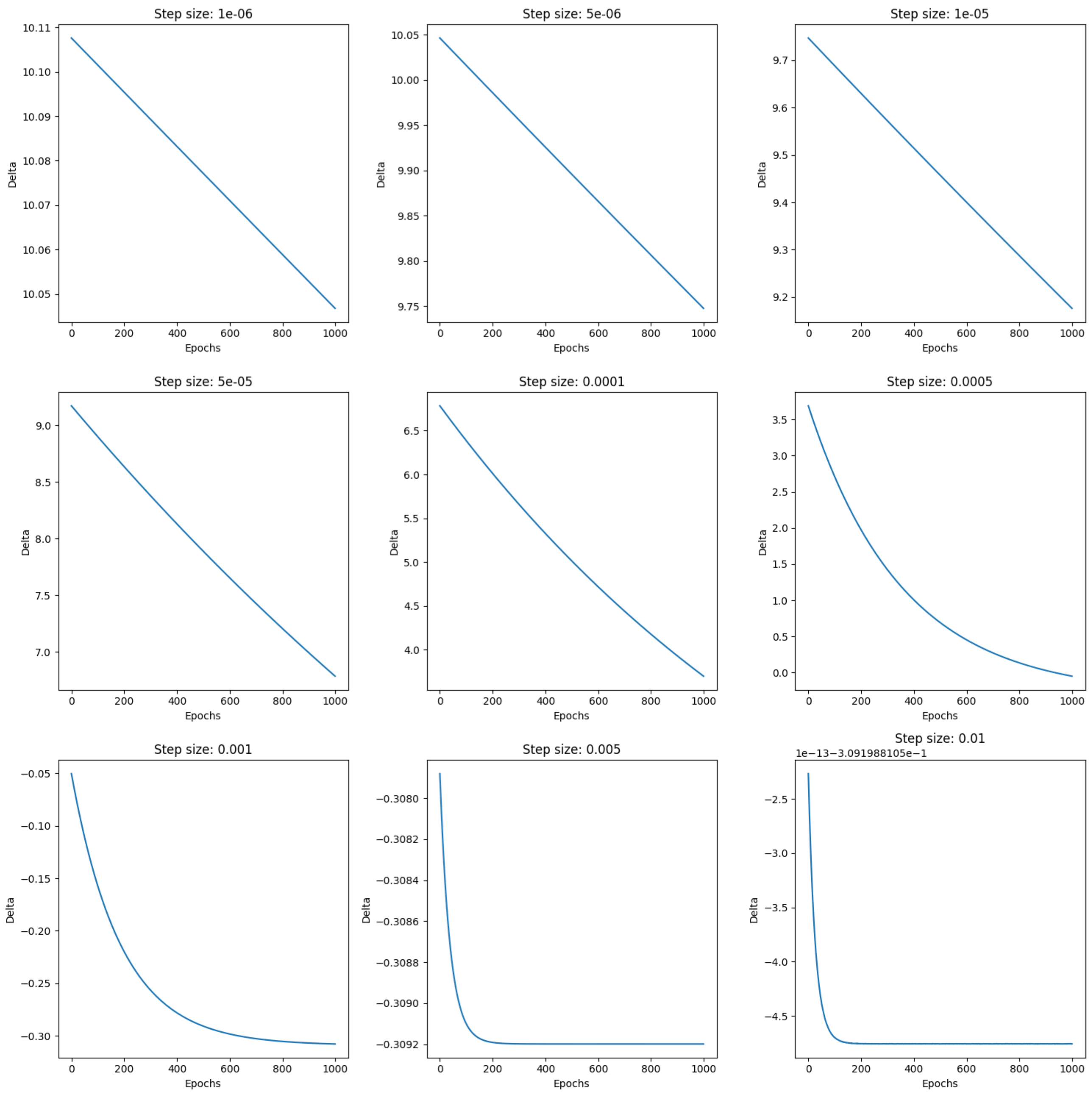
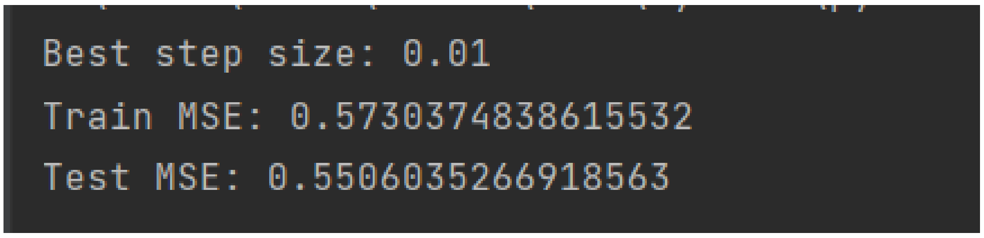


Figure 4: f_out1

Listing 4: Q1_f.py

```
def ridge_regression_loss(X, y, beta, phi):
    n = X.shape[0]
    return (1 / n) * np.sum((y - X.dot(beta)) ** 2) + phi * np.sum(beta ** 2)
def ridge_regression_gradient(X, y, beta, phi):
    n = X.shape[0]
```

```python
    return (-2 / n) * X.T.dot(y - X.dot(beta)) + 2 * phi * beta
def batch_gradient_descent(X, y, beta_init, step_size, epochs, phi):
    beta = beta_init
    loss_history = []
    for _ in range(epochs):
        beta -= step_size * ridge_regression_gradient(X, y, beta, phi)
        loss_history.append(ridge_regression_loss(X, y, beta, phi))
    return beta, loss_history
def train_test_MSE(beta, X_train, y_train, X_test, y_test):
    train_mse = (1 / X_train.shape[0]) * np.sum((y_train - X_train.dot(beta)) ** 2)
    test_mse = (1 / X_test.shape[0]) * np.sum((y_test - X_test.dot(beta)) ** 2)
    return train_mse, test_mse
# Load train and test data here
data = pd.read_csv('CarSeats.csv')
data = data.drop(columns=['ShelveLoc', 'Urban', 'US'])
scaler = StandardScaler()
data_scaled = pd.DataFrame(scaler.fit_transform(data), columns=data.columns)
data_scaled['Sales'] = data_scaled['Sales'] - data_scaled['Sales'].mean()
n = len(data_scaled)
X_train = data_scaled.iloc[:n//2, 1:]
X_test = data_scaled.iloc[n//2:, 1:]
Y_train = data_scaled.iloc[:n//2, 0]
Y_test = data_scaled.iloc[n//2:, 0]
phi = 0.5
epochs = 1000
step_sizes = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005,
    0.01]
beta_init = np.ones(X_train.shape[1])
# Calculate the closed form ridge solution
beta_closed_form = np.linalg.inv(X_train.T.dot(X_train) + phi * np.eye(X_train.shape
    [1])).dot(X_train.T).dot(Y_train)
fig, axes = plt.subplots(3, 3, figsize=(15, 15))
axes = axes.ravel()
best_beta = None
best_step_size = None
lowest_test_mse = float('inf')
for i, step_size in enumerate(step_sizes):
    beta_gd, loss_history = batch_gradient_descent(X_train, Y_train, beta_init,
        step_size, epochs, phi)
    delta = np.array(loss_history) - ridge_regression_loss(X_train, Y_train,
        beta_closed_form, phi)
    train_mse, test_mse = train_test_MSE(beta_gd, X_train, Y_train, X_test, Y_test)
    if test_mse < lowest_test_mse:
        best_beta = beta_gd
        best_step_size = step_size
        lowest_test_mse = test_mse
    ax = axes[i]
    ax.plot(delta)
    ax.set_title(f"Step size: {step_size}")
    ax.set_xlabel("Epochs")
    ax.set_ylabel("Delta")
plt.tight_layout()
plt.savefig('f_plot.png')
plt.show()
train_mse, test_mse = train_test_MSE(best_beta, X_train, Y_train, X_test, Y_test)
print(f"Best step size: {best_step_size}")
print(f"Train MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```



```
Best step size: 0.01
Train MSE: 0.5730374838615532
Test MSE: 0.5506035266918563
```

Figure 5: f_out2

g.

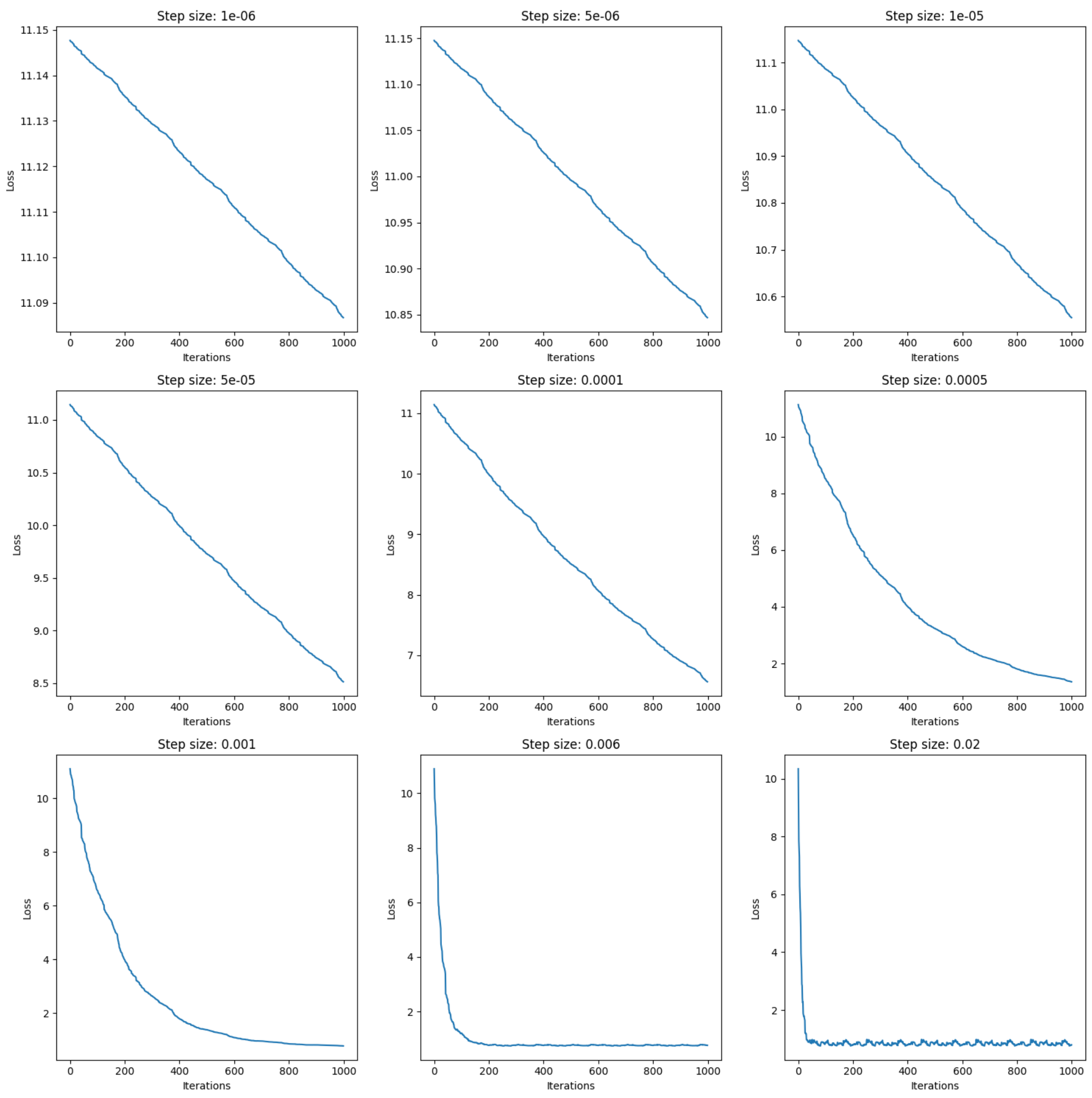
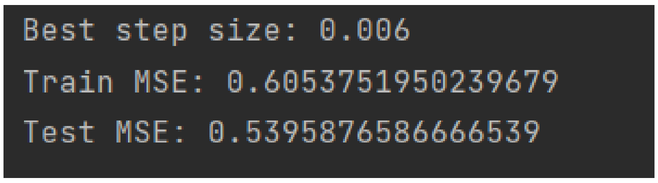


Figure 6: g_out1

Listing 5: Q1_g.py

```python
def ridge_regression_loss(X, y, beta, phi):
    n = X.shape[0]
    return (1 / n) * np.sum((y - X.dot(beta)) ** 2) + phi * np.sum(beta ** 2)
def ridge_regression_gradient(x_i, y_i, beta, phi):
    return (-2) * x_i.T.dot(y_i - x_i.dot(beta)) + 2 * phi * beta
def sgd(X, y, beta_init, step_size, epochs, phi):
    beta = beta_init.copy()
    loss_history = []
    n = X.shape[0]
```



```
Best step size: 0.006
Train MSE: 0.6053751950239679
Test MSE: 0.5395876586666539
```

Figure 7: g_out2

```
    for _ in range(epochs):
        for i in range(n):
            x_i = X.iloc[i].values.reshape(1, -1)
            y_i = y[i]
            beta -= step_size * ridge_regression_gradient(x_i, y_i, beta, phi)
            loss_history.append(ridge_regression_loss(X, y, beta, phi))
    return beta, loss_history
def train_test_MSE(beta, X_train, y_train, X_test, y_test):
    train_mse = (1 / X_train.shape[0]) * np.sum((y_train - X_train.dot(beta)) ** 2 -
        np.mean(y_train - X_train.dot(beta)))
    test_mse = (1 / X_test.shape[0]) * np.sum((y_test - X_test.dot(beta)) ** 2 - np.
        mean(y_test - X_test.dot(beta)))
    return train_mse, test_mse
# Load train and test data here
data = pd.read_csv('CarSeats.csv')
data = data.drop(columns=['ShelveLoc', 'Urban', 'US'])
scaler = StandardScaler()
data_scaled = pd.DataFrame(scaler.fit_transform(data), columns=data.columns)
data_scaled['Sales'] = data_scaled['Sales'] - data_scaled['Sales'].mean()
n = len(data_scaled)
X_train = data_scaled.iloc[:n//2, 1:]
X_test = data_scaled.iloc[n//2:, 1:]
y_train = data_scaled.iloc[:n//2, 0]
y_test = data_scaled.iloc[n//2:, 0]
phi = 0.5
epochs = 5
step_sizes = [0.000001, 0.000005, 0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.006,
    0.02]
beta_init = np.ones(X_train.shape[1])
fig, axes = plt.subplots(3, 3, figsize=(15, 15))
axes = axes.ravel()
best_beta = None
best_step_size = None
lowest_test_mse = float('inf'
for i, step_size in enumerate(step_sizes):
    beta_sgd, loss_history = sgd(X_train, y_train, beta_init, step_size, epochs, phi)
    train_mse, test_mse = train_test_MSE(beta_sgd, X_train, y_train, X_test, y_test)
    if test_mse < lowest_test_mse:
        best_beta = beta_sgd
        best_step_size = step_size
        lowest_test_mse = test_mse
    ax = axes[i]
    ax.plot(loss_history)
    ax.set_title(f"Step size: {step_size}")
    ax.set_xlabel("Iterations")
    ax.set_ylabel("Loss")
plt.tight_layout()
plt.savefig('g_plot.png')
plt.show()
train_mse, test_mse = train_test_MSE(best_beta, X_train, y_train, X_test, y_test)
print(f"Best step size: {best_step_size}")
print(f"Train MSE: {train_mse}")
print(f"Test MSE: {test_mse}")
```

The value of Δ(k) jumping up and down in SGD is because, unlike in batch gradient descent, the gradients are estimated using only one training example at a time. Since each example may have different features and noise, the direction of the gradient update may vary, causing the fluctuations in the loss function.

h.

In general, the choice between Gradient Descent (GD) and Stochastic Gradient Descent (SGD) depends on the specific problem and the size of the dataset. Here are some factors to consider when choosing between GD and SGD:

1. Dataset size: If the dataset is large, GD can be computationally expensive, as it calculates the gradient using the entire dataset in each iteration. In such cases, SGD is preferred, as it updates the model parameters more frequently using only a single data point (or a small batch) in each iteration, leading to faster convergence.

2. Convergence speed: While SGD can converge faster than GD, the convergence can be noisy due to the use of only a single data point (or a small batch) in each iteration. This can lead to oscillations around the optimal solution, without necessarily achieving the exact minimum. GD converges more smoothly due to the use of the entire dataset in each update.

3. Local minima: SGD can sometimes escape local minima due to its noisy convergence, whereas GD can get stuck in local minima since it follows the exact negative of the gradient.

4. Memory requirements: GD requires more memory to store the entire dataset, whereas SGD can work with a smaller memory footprint, especially if using mini-batch SGD.

Based on the results of your GD and SGD implementations, you should compare the final loss, the number of iterations, and the convergence speed. If the SGD converges faster and provides a similar or better final loss, you might prefer SGD. However, if the GD provides a better loss and the dataset size is not too large, GD could be the better choice.

In summary:

- Use GD when the dataset size is manageable, and you need a smoother convergence to the optimal solution.

- Use SGD when dealing with large datasets, requiring faster convergence, or when you have limited memory resources.

i.

To find the solution for the optimization problem of updating β_1, we minimize the Ridge regression loss function with respect to β_1 while keeping $\beta_2, \beta_3, ..., \beta_p$ fixed:

$$L(\beta_1, \beta_2, ..., \beta_p) = \frac{1}{n}\|y - X\beta\|_2^2 + \phi(\|\beta\|_2^2) = \frac{1}{n}\|y - X_1\beta_1 - X_2\beta_2 - ... - X_p\beta_p\|_2^2 + \phi(\|\beta_1\|_2^2 + \|\beta_2\|_2^2 + ... + \|\beta_p\|_2^2)$$

where X_j denotes the j-th column of X and X_{-j} denotes the matrix X but with the j-th column removed.

Expanding the squared term in the above expression and collecting the terms that depend on β_1 only, we get:

$$L(\beta_1, \beta_2, ..., \beta_p) = \frac{1}{n}\|y - X_2\beta_2 - ... - X_p\beta_p - X_1\beta_1\|_2^2 + \phi(\|\beta_1\|_2^2) + const$$

where "const" denotes the terms that do not depend on β_1 and can be ignored while minimizing the loss function.

Differentiating the loss function with respect to β_1 and setting the resulting expression to zero, we get:

$$-\frac{2}{n}X_1'(y - X_2\beta_2 - ... - X_p\beta_p - X_1\beta_1) + 2\phi\beta_1 = 0$$

Solving for β_1, we have:

$$\beta_1 = (X_1'X_1 + n\phi I)^{-1}X_1'(y - X_2\beta_2 - ... - X_p\beta_p)$$

where I is the identity matrix of size $p - 1$.

Similarly, we can derive expressions for $\hat{\beta}_j$ for $j = 2, 3, ..., p$ by minimizing the loss function with respect to β_j while keeping $\beta_1, ..., \beta_{j-1}, \beta_{j+1}, ..., \beta_p$ fixed:

$$\beta_j = (X_j'X_j + n\phi I)^{-1}X_j'(y - X_1\beta_1 - ... - X_{j-1}\beta_{j-1} - X_{j+1}\beta_{j+1} - ... - X_p\beta_p)$$

We can then cycle through these coordinate updates until convergence is achieved.

j.

Train MSE (CGD): 0.6909

Test MSE (CGD): 0.6224

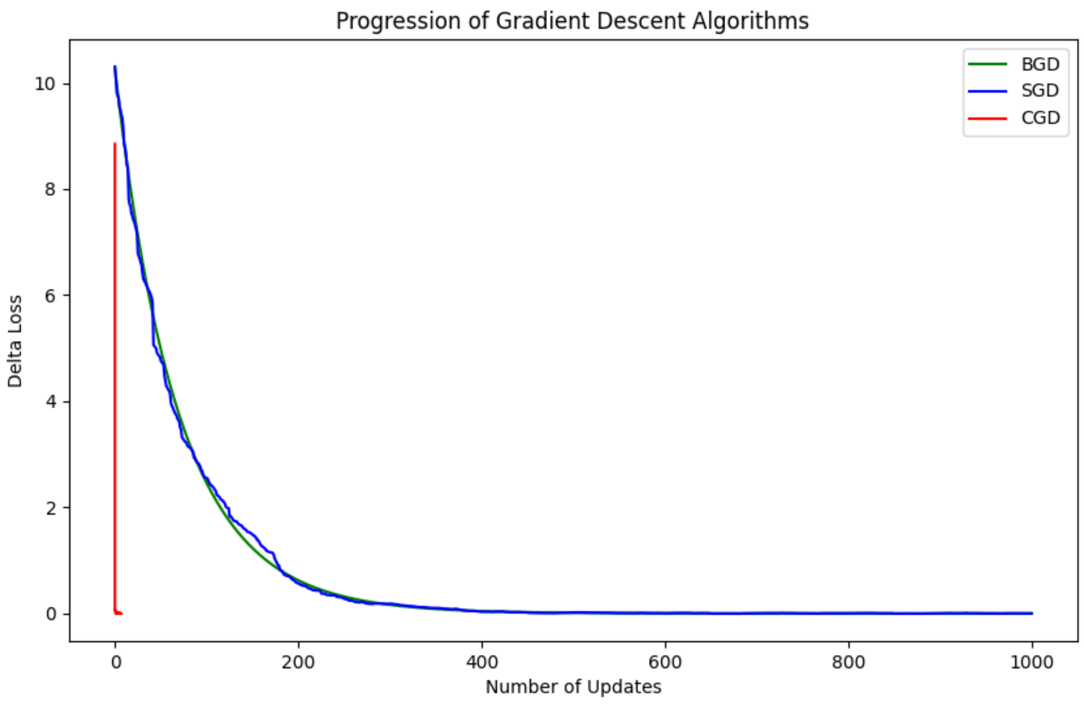
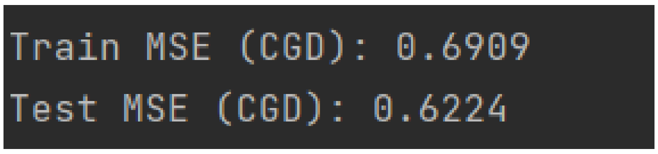


Figure 8: j_out1



```
Train MSE (CGD): 0.6909
Test MSE (CGD): 0.6224
```

Figure 9: j_out2

2 Q2

a.

This optimization problem is a Lasso regression problem with a single variable, where the objective function combines L1 regularization and a quadratic loss function. The solution can be found using the soft-thresholding operator.

To find the expression for β^, we can differentiate the objective function with respect to β and set it to zero:

$$\frac{d}{d\beta}\left(|\beta| + \frac{1}{2\lambda}(\beta - v)^2\right) = 0$$

The derivative of the absolute value $|\beta|$ is not defined at $\beta = 0$, but we can consider the cases where $\beta > 0$, $\beta < 0$, and $\beta = 0$ separately.

1. If $\beta > 0$:

$$\frac{d}{d\beta}\left(\beta + \frac{1}{2\lambda}(\beta - v)^2\right) = 1 + \frac{1}{\lambda}(\beta - v) = 0 \Rightarrow \beta = v - \lambda$$

2. If $\beta < 0$:

$$\frac{d}{d\beta}\left(-\beta + \frac{1}{2\lambda}(\beta - v)^2\right) = -1 + \frac{1}{\lambda}(\beta - v) = 0 \Rightarrow \beta = v + \lambda$$

3. If $\beta = 0$ and $|v| \leq \lambda$: The objective function is minimized when $\beta = 0$. Putting the results together, we get the following expression for $\beta^\wedge$:

$$\beta^\wedge = T_\lambda(v) := \begin{cases} v - \lambda & \text{if } v > \lambda, \\ 0 & \text{if } |v| \leq \lambda, \\ v + \lambda & \text{if } v < -\lambda. \end{cases}$$

b.

1. Rewrite the objective function:

The given minimization problem is:

$$\min_{\beta \in \mathbb{R}^p} (\|\beta\|_1 + \frac{1}{2\lambda}\|\beta - v\|_2^2)$$

where $v = (v_1, ..., v_p) \in \mathbb{R}^p$ and $\lambda > 0$.

We can rewrite the objective function by splitting the L1-norm ($\|\beta\|_1$) and the L2-norm ($\|\beta - v\|_2^2$) into their individual components:

$$f(\beta) = \sum_i |\beta_i| + \frac{1}{2\lambda}\sum_i (\beta_i - v_i)^2$$

for $i = 1, ..., p$.

2. Recognize the separable nature of the objective function:

The objective function is separable because it is a sum of functions that depend only on one component (β_i) at a time. This means we can minimize the function with respect to each β_i independently, instead of solving for all β_i simultaneously.

3. Apply the single-variable solution to each component:

For each component $i = 1, ..., p$, we have the following single-variable optimization problem:

$$\min_{\beta_i \in \mathbb{R}} (|\beta_i| + \frac{1}{2\lambda}(\beta_i - v_i)^2)$$

From the previous single-variable result, we know that the solution to this problem is:

$$\beta_i^\wedge = T_\lambda(v_i) = \begin{cases} v_i - \lambda & \text{if } v_i > \lambda, \\ 0 & \text{if } |v_i| \leq \lambda, \\ v_i + \lambda & \text{if } v_i < -\lambda \end{cases}$$

4. Combine the solutions for each component:

Since the single-variable solution holds for each component i, we can combine these solutions to obtain the solution for the original multi-variable problem:

$$\beta^\wedge = T_\lambda(v) = (T_\lambda(v_1), T_\lambda(v_2), ..., T_\lambda(v_p))$$

So, for any $\lambda > 0$ and $v = (v_1, ..., v_p) \in \mathbb{R}^p$, the solution of the given minimization problem is $\beta^\wedge = T_\lambda(v) = (T_\lambda(v_1), T_\lambda(v_2), ..., T_\lambda(v_p))$.

c.

Let's compute $T_\lambda(v)$ for the given vector v and the specified values $\lambda = 1, 3, 6, 9$:

$$v = (1, 2, 4, -7, 2, 4, -1, 8, 4, -10, -5)$$

Recall the definition of $T_\lambda(v_i)$:

$$T_\lambda(v_i) = \begin{cases} v_i - \lambda & \text{if } v_i > \lambda, \\ 0 & \text{if } |v_i| \leq \lambda, \\ v_i + \lambda & \text{if } v_i < -\lambda \end{cases}$$

For $\lambda = 1$, we get:

$$T_1(v) = (0, 1, 3, -6, 1, 3, 0, 7, 3, -9, -4)$$

For $\lambda = 3$, we get:

$$T_3(v) = (0, 0, 1, -4, 0, 1, 0, 5, 1, -7, -2)$$

For $\lambda = 6$, we get:

$$T_6(v) = (0, 0, 0, -1, 0, 0, 0, 2, 0, -4, 0)$$

For $\lambda = 9$, we get:

$$T_9(v) = (0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0)$$

As we increase the value of λ, we observe that more elements in the resulting vector $T_\lambda(v)$ become zero or closer to zero. This is because the shrinkage effect of T_λ on the original vector v increases as λ increases.

In practice, this means that when using this function to solve a Lasso regression problem, a larger λ results in more coefficient estimates being shrunk towards zero, effectively promoting sparsity in the solution. In other words, a larger λ leads to a simpler model with fewer non-zero coefficients, potentially reducing the risk of overfitting.