Programming Machine Learning Lab

Exercise 5

General Instructions:

1. You need to submit the PDF as well as the filled notebook file.
2. Name your submissions by prefixing your matriculation number to the filename. Example, if your MR is 12345 then rename the files as **"12345_Exercise_5.xxx"**
3. Complete all your tasks and then do a clean run before generating the final pdf. (*Clear All Ouputs* and *Run All* commands in Jupyter notebook)

Exercise Specific instructions::

1. You are allowed to use only NumPy and Pandas (unless stated otherwise). You can use any library for visualizations.

Part 1

Gradient Descent for Ridge Regression

In this part of the assignment we will perform linear regression with L2 regularization using Gradient Descent. We will use the **"regression.csv"**. Remember to split the dataset into 80% for training and 20% for test, and perform standard scaling of the features.

You need to code a function which takes in X and y as input and outputs the learned beta values. Also track the loss value over the iterations and plot them. You would need to find the learning rate η by trial and error.

Coding Hints It is easier to break the whole code into small blocks, so you can

- Create a loss function that takes in X, beta and y_actual and returns the loss value at current step.
- Create a gradient calculation function that takes in X, beta and y_actual and returns the gradient direction for current step.
- Maintaining a list of loss values would help in checking the exit condition as well as help in the plotting at the end.

The algorithm for linear regression with L2 regularization is given below:

Gradient Descent for Ridge Regression

- Gradient for the objective function of ridge regression can be derived as

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2 \boxed{+\lambda\|\boldsymbol{\theta}\|_2^2}$$

$$\nabla L(\boldsymbol{\theta}) = \frac{\partial}{\partial\boldsymbol{\theta}}\frac{1}{N}\sum_{n=1}^{N}(f_{\boldsymbol{\theta}}(\mathbf{x}_n) - y_n)^2 \boxed{+\frac{\partial}{\partial\boldsymbol{\theta}}\lambda\|\boldsymbol{\theta}\|_2^2} = \frac{1}{N}(-\mathbf{X})^{\mathrm{T}}2(\mathbf{y}-\mathbf{X}\boldsymbol{\theta})\boxed{+2\lambda\boldsymbol{\theta}}$$

Derivative of regularization term:

$$\frac{\partial}{\partial\boldsymbol{\theta}}\|\boldsymbol{\theta}\|_2^2 = \frac{\partial}{\partial\boldsymbol{\theta}}\langle\boldsymbol{\theta},\boldsymbol{\theta}\rangle = 2\boldsymbol{\theta}$$

- Optimize model by gradient descent:

Gradient descent algorithm

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,\ldots,i_{max}$:
3. $\quad \boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta\nabla L(\boldsymbol{\theta}_i)$
4. $\quad$ if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$:
5. $\qquad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

In [ ]:
```
#### write your code here
# imports
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
```

In [ ]:
```
class Loss:
    def __init__(self, lamb):
        self.history_train = []
        self.history_test = []
        self.lamb = lamb

    def mean_square_loss(self, x, y, theta, x_test, y_test):
        # Mean Square Loss for Regression
        predictions = np.dot(x, theta)
        mse = np.mean((predictions - y) ** 2) + self.lamb*np.sum(theta**2)
        self.history_train.append(mse)
        self.history_test.append(
            np.mean((np.dot(x_test, theta) - y_test) ** 2) + self.lamb*np.sum(theta
        )
        return mse

    def mean_square_loss_gradient(self, x, y, theta):
        # Gradient of Mean Square Loss for Regression
        gradient = 2 * np.dot(x.T, (np.dot(x, theta) - y))  / len(y) + 2*self.lamb*
```

```python
        return gradient

class Optimization:
    def __init__(self, x, y, x_test, y_test, lamb):
        self.x = x
        self.y = y
        self.x_test = x_test
        self.y_test = y_test
        self.lamb = lamb
        self.loss = Loss(self.lamb)

    def gradient_descent(self, theta, learning_rate, epochs):
        # Gradient Descent for Mean Square Loss
        m = len(self.y)
        for epoch in range(epochs):
            loss = self.loss.mean_square_loss(
                self.x, self.y, theta, self.x_test, self.y_test
            )
            gradient = self.loss.mean_square_loss_gradient(
                self.x, self.y, theta
            )
            theta = theta - learning_rate * gradient
        return theta
```

In [ ]:

```python
df = pd.read_csv('regression.csv', header=0, index_col=None)

# Split the dataset into 80% for training and 20% for test
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)

scaler = StandardScaler()
train_df.iloc[:, :-1] = scaler.fit_transform(train_df.iloc[:, :-1])
test_df.iloc[:, :-1] = scaler.transform(test_df.iloc[:, :-1])

# Linear Regression class
class LinearRegression:
    def __init__(self, x, y, x_test, y_test, learning_rate=0.01, epochs=100, lamb =
        self.x = np.hstack((np.ones((x.shape[0], 1)), x))  # Add a column of ones f
        self.y = y.reshape(-1, 1)
        self.x_test = np.hstack((np.ones((x_test.shape[0], 1)), x_test))
        self.y_test = y_test.reshape(-1, 1)
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.lamb = lamb

    def fit(self):
        initial_theta = np.zeros((self.x.shape[1], 1))
        self.optimization = Optimization(self.x, self.y, self.x_test, self.y_test,
        self.theta = self.optimization.gradient_descent(initial_theta, self.learnin

    def predict(self, x):
        x = np.hstack((np.ones((x.shape[0], 1)), x))
        return np.dot(x, self.theta)

# Train a linear regression model
X_train = train_df.iloc[:, :-1].values
y_train = train_df['Y'].values
```

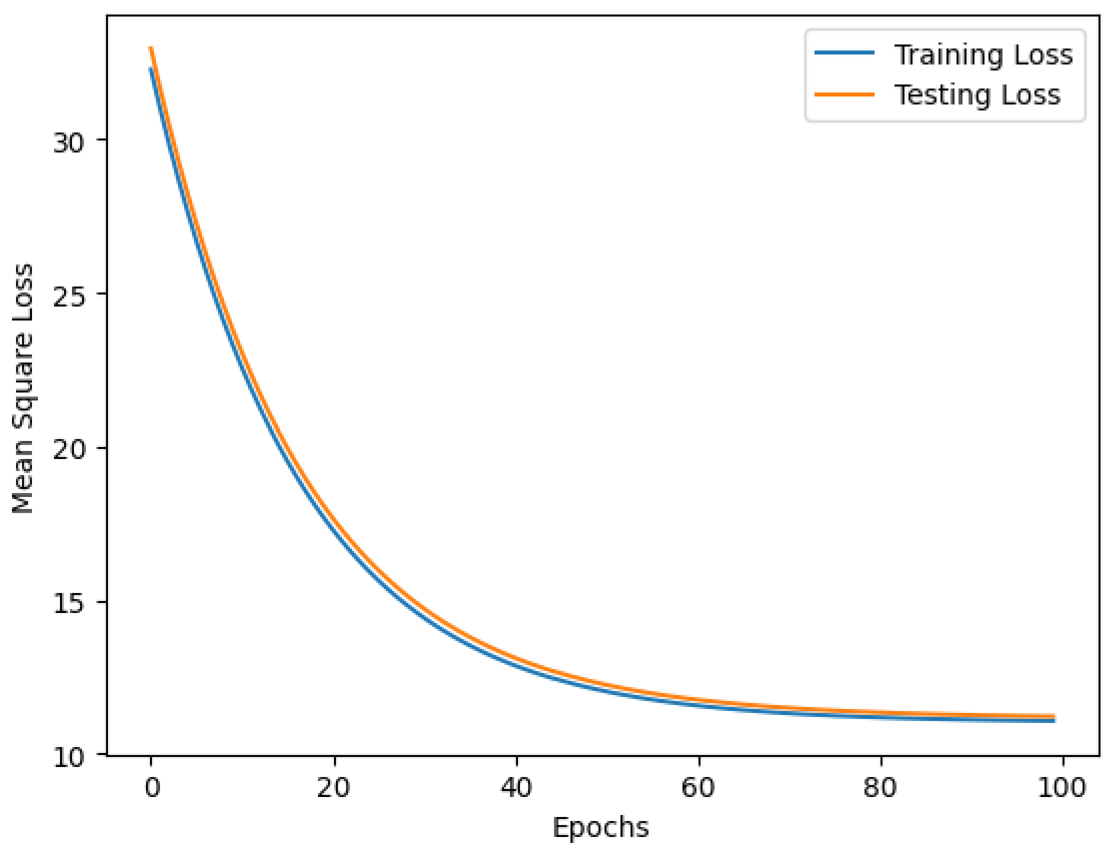
```python
X_test = test_df.iloc[:, :-1].values
y_test = test_df['Y'].values

linear_reg = LinearRegression(X_train, y_train, X_test, y_test)
linear_reg.fit()

print(f'Final validation loss: {linear_reg.optimization.loss.history_test[len(linea
print(f'Betas: {linear_reg.theta}')

# Plot the loss trajectory for both training and testing datasets
plt.plot(linear_reg.optimization.loss.history_train, label='Training Loss')
plt.plot(linear_reg.optimization.loss.history_test, label='Testing Loss')
plt.xlabel('Epochs')
plt.ylabel('Mean Square Loss')
plt.legend()
plt.show()
```

```
Final validation loss: 11.223106937577153
Betas: [[ 3.57099547]
 [ 0.03933986]
 [-0.1265984 ]
 [ 0.04298742]
 [ 0.01272058]
 [-0.0604303 ]
 [ 0.00915962]
 [-0.07201057]
 [-0.05919468]
 [-0.011193  ]
 [ 0.09916581]
 [ 0.18634124]]
```



Evaluation

For evaluation, try at least 3 different values of λ (one of them must be 0) and report the test loss. Also comments on the results.

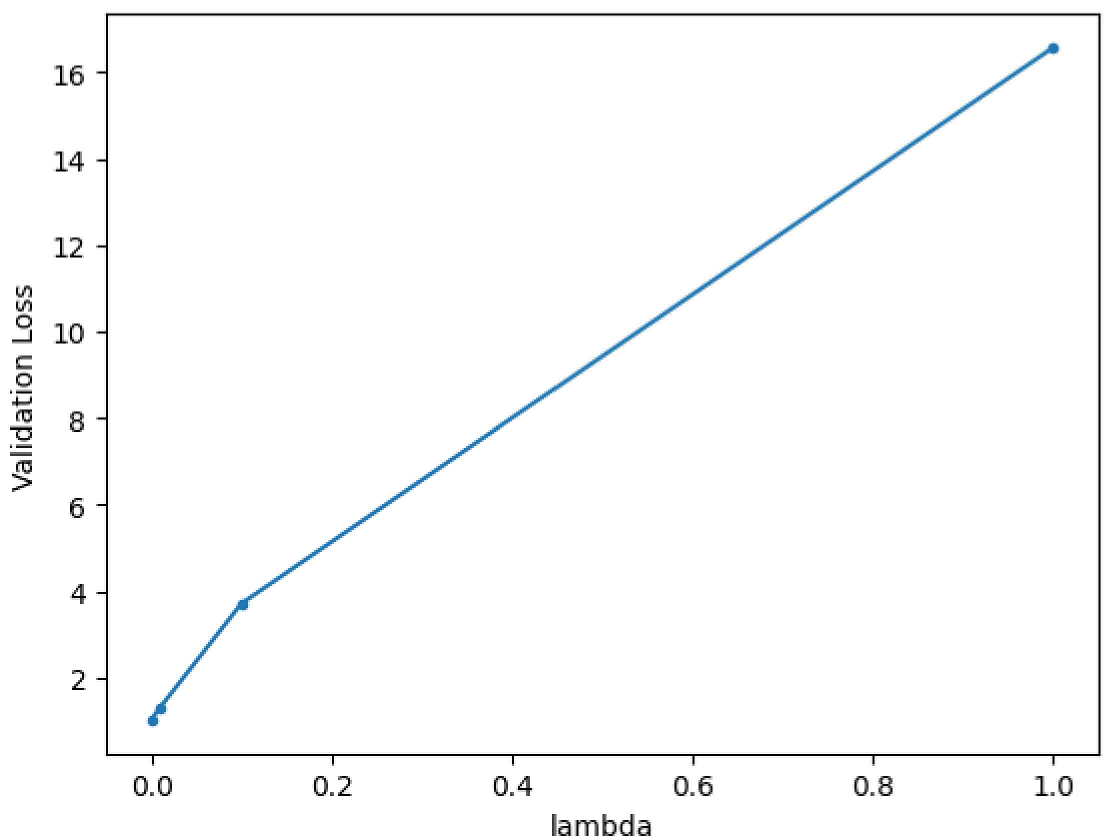
In [ ]:
```
#### write your code here
linear_reg0 = LinearRegression(X_train, y_train, X_test, y_test, lamb=0)
linear_reg0.fit()

linear_reg1 = LinearRegression(X_train, y_train, X_test, y_test, lamb=0.01)
linear_reg1.fit()

linear_reg2 = LinearRegression(X_train, y_train, X_test, y_test, lamb=0.1)
linear_reg2.fit()

linear_reg3 = LinearRegression(X_train, y_train, X_test, y_test, lamb=1)
linear_reg3.fit()

plt.plot([0, 0.01, 0.1, 1 ] , [
    linear_reg0.optimization.loss.history_test[len(linear_reg0.optimization.loss.hi
    linear_reg1.optimization.loss.history_test[len(linear_reg1.optimization.loss.hi
    linear_reg2.optimization.loss.history_test[len(linear_reg2.optimization.loss.hi
    linear_reg3.optimization.loss.history_test[len(linear_reg3.optimization.loss.hi
], '.-')
plt.xlabel('lambda')
plt.ylabel('Validation Loss')
plt.show()
```



Part 2

L2-Regularized Logistic Regression

In this part of the assignment we will perform logistic regression with L2 regularization using Gradient Descent. We will use the **"logistic.csv"**. Remember to split the dataset into 80% for training and 20% for test, and perform standard scaling of the features.

You need to code a function which takes in X and y as input and outputs the learned beta values. Also track the loss value over the iterations and plot them. You would need to find the learning rate η by trial and error.

Coding Hints It is easier to break the whole code into small blocks, so you can

- Create a loss function that takes in X, beta and y_actual and returns the loss value at current step.
- Create a gradient calculation function that takes in X, beta and y_actual and returns the gradient direction for current step.
- Maintaining a list of loss values would help in checking the exit condition as well as help in the plotting at the end.

The algorithm for logistic regression with L2 regularization is given below:

L2-Regularized Logistic Regression

- Add L2 regularization to logistic regression: because we are maximizing objective function, need to substract the penalty term

$$L_{cll}(\boldsymbol{\theta}) = \log p(y_1,\ldots,y_N \mid \mathbf{x}_1,\ldots,\mathbf{x}_N,\boldsymbol{\theta}) \boxed{-\lambda\|\boldsymbol{\theta}\|_2^2}$$

- The gradient then becomes

$$\nabla L_{cll}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathbf{x}_n (y_n - f(\mathbf{x}_n)) \boxed{-2\lambda\boldsymbol{\theta}}$$

- Learn parameters by gradient ascent as before

Gradient ascent algorithm

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,\ldots,i_{max}$:
3. $\quad \boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + \eta \nabla L_{cll}(\boldsymbol{\theta}_i)$
4. $\quad$ if $L_{cll}(\boldsymbol{\theta}_{i+1}) - L_{cll}(\boldsymbol{\theta}_i) < \epsilon$:
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in i_{max} iterations")

Lecture „Machine Learning“ 38

In [ ]:
```python
class LogisticOptimization:
    def __init__(self, x, y, x_test, y_test, learning_rate=0.01, epochs=100, lamb =
        self.x = np.hstack((np.ones((x.shape[0], 1)), x))  # Add a column of ones f
        self.y = y.reshape(-1, 1)
        self.x_test = np.hstack((np.ones((x_test.shape[0], 1)), x_test))
        self.y_test = y_test.reshape(-1, 1)
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.lamb = lamb
        self.theta = np.random.randn(self.x.shape[1], 1)  # Add 1 for the bias term
        self.train_historyLoss = []
        self.test_historyLoss = []

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def compute_loss(self, X, y, X_test, Y_test):
        # Compute scores
        scores = X.dot(self.theta)
        # Calculate cross-entropy loss for binary classification
        loss = (y * np.log(self.sigmoid(scores)) + (1 - y) * np.log(1 - self.sigmoi

        # Add L2 regularization term
        reg_term = self.lamb * np.sum(self.theta ** 2)
        total_loss = loss - reg_term
        self.train_historyLoss.append(total_loss)
        self.test_historyLoss.append(
            (y_test * np.log(self.sigmoid(X_test.dot(self.theta))) + (1 - y_test) *
        )
        return total_loss

    def compute_gradient(self, X, y):
        num_samples = X.shape[0]


        # Calculate gradient for binary classification
        gradient = (X.T.dot(self.sigmoid(y -np.dot(X, self.theta))) / num_samples)

        return gradient

    def gradient_step(self, gradient):
        # Update weights and bias using gradient ascent
        self.theta += self.learning_rate * gradient

    def fit(self):
        for epoch in range(self.epochs):
            # Compute loss and gradient
            loss = self.compute_loss(self.x, self.y, self.x_test, self.y_test)
            gradient = self.compute_gradient(self.x, self.y)

            # Perform a gradient step (ascent)
            self.gradient_step(gradient)
```

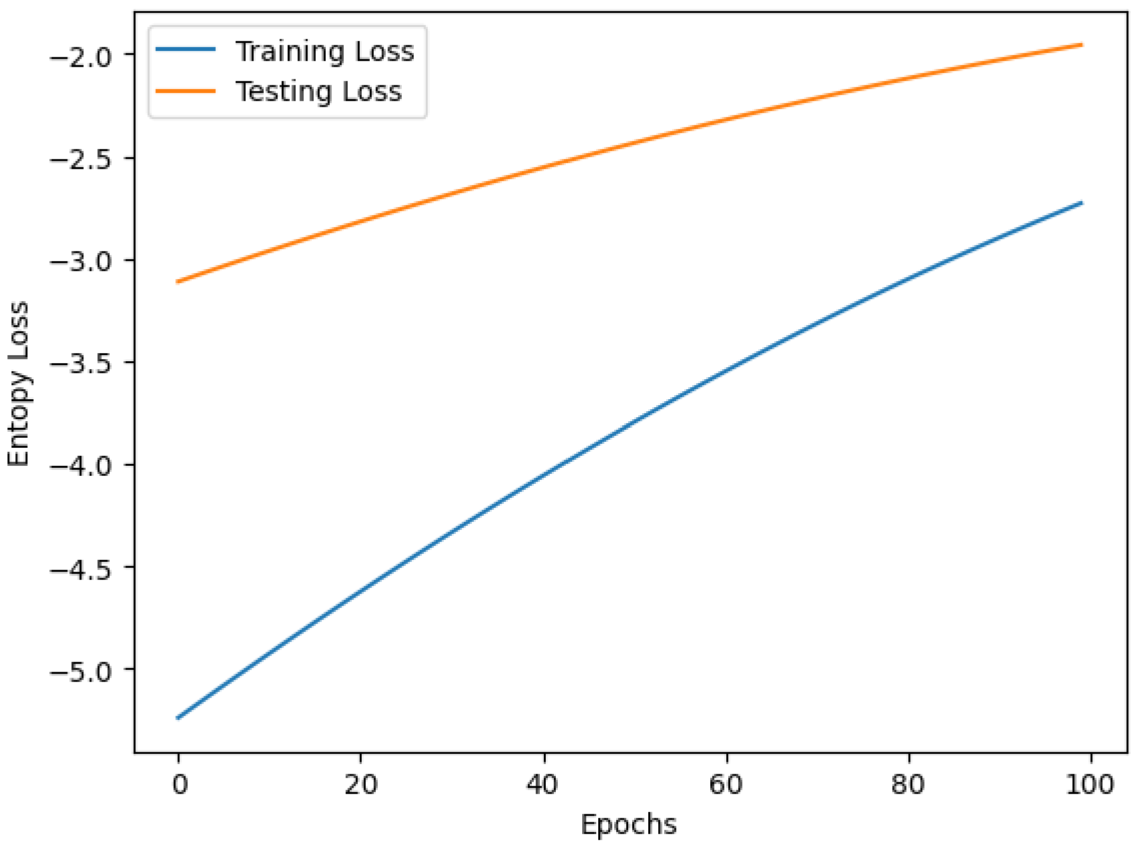
In [ ]:
```
df = pd.read_csv('logistic.csv', header=0, index_col=None)

# Split the dataset into 80% for training and 20% for test
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)

label_encoder = LabelEncoder()
train_df['Y'] = label_encoder.fit_transform(train_df['Y'])
test_df['Y'] = label_encoder.transform(test_df['Y'])
scaler = StandardScaler()
X_train = train_df.iloc[:, 1:].values
y_train = train_df['Y'].values
X_test = test_df.iloc[:, 1:].values
y_test = test_df['Y'].values
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Instantiate Logistic Regression model
logistic_reg = LogisticOptimization(X_train, y_train, X_test, y_test, lamb=0.01)
logistic_reg.fit()

plt.plot(logistic_reg.train_historyLoss, label='Training Loss')
plt.plot(logistic_reg.test_historyLoss, label='Testing Loss')
plt.xlabel('Epochs')
plt.ylabel('Entopy Loss')
plt.legend()
plt.show()
```



Evaluation

For evaluation, try at least 3 different values of λ (one of them must be 0) and report the test losses. Also comments on the results.

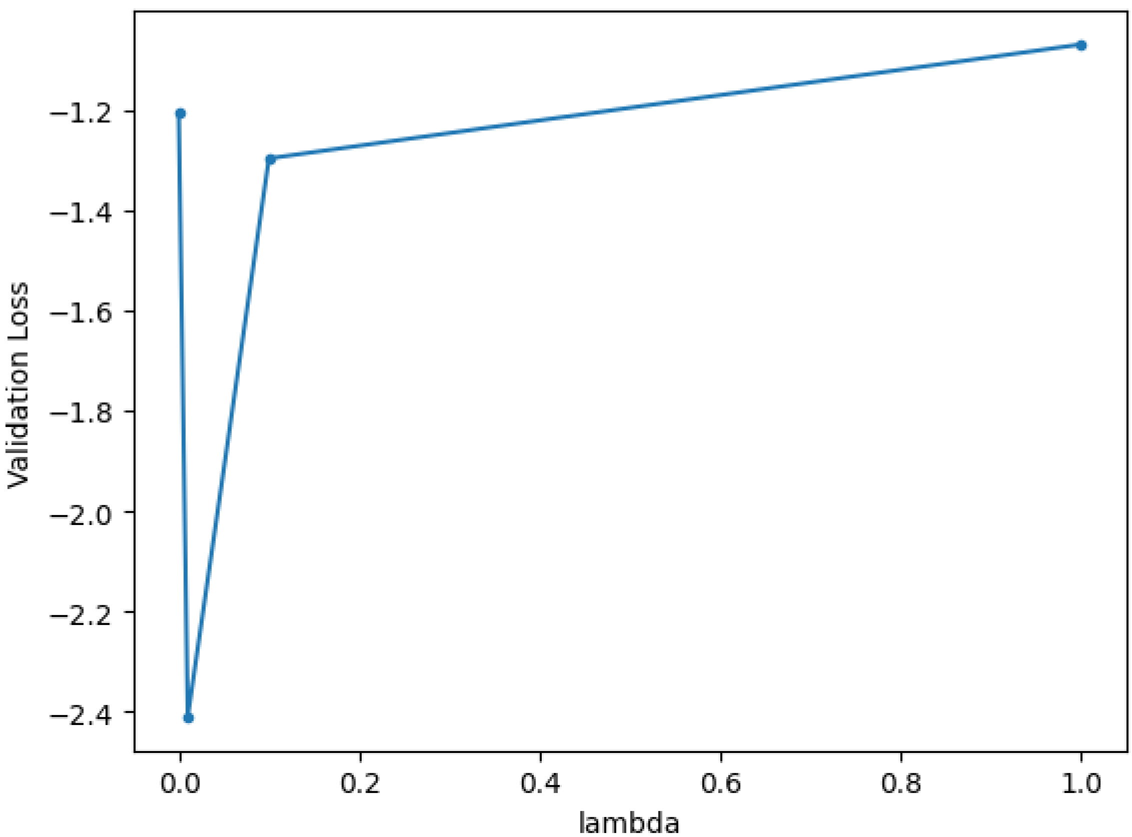
In [ ]:
```
#### write your code here
#### write your code here
logistic_reg = LogisticOptimization(X_train, y_train, X_test, y_test, lamb=0)
logistic_reg.fit()

logistic_reg1 = LogisticOptimization(X_train, y_train, X_test, y_test, lamb=0.01)
logistic_reg1.fit()

logistic_reg2 = LogisticOptimization(X_train, y_train, X_test, y_test, lamb=0.001)
logistic_reg2.fit()

logistic_reg3 = LogisticOptimization(X_train, y_train, X_test, y_test, lamb=1)
logistic_reg3.fit()

plt.plot([0, 0.01, 0.1, 1 ] , [
    logistic_reg.test_historyLoss[len(logistic_reg.test_historyLoss) - 1],
    logistic_reg1.test_historyLoss[len(logistic_reg.test_historyLoss) - 1],
    logistic_reg2.test_historyLoss[len(logistic_reg.test_historyLoss) - 1],
    logistic_reg3.test_historyLoss[len(logistic_reg.test_historyLoss) - 1],
], '.-')
plt.xlabel('lambda')
plt.ylabel('Validation Loss')
plt.show()
```



In [ ]: