Lab Report: 3 &4

Name: **Prem Vikram Mokal**

Roll no.: **B24BB1028**

1. Introduction

This report provides a comprehensive analysis of the Python code presented in the python files Q1.py and Q2.py. The code implements two fundamental machine learning models: Linear Regression and Logistic Regression.

The primary objectives of this report are:
• To explain the step-by-step execution of the code for both models.
• To detail the data preprocessing and dataset splitting procedures.
• To provide a clear, in-depth explanation of the ***object-oriented*** implementation of the Gradient Descent algorithm for model training.

2. Question 1: Linear Regression (Q1.py)

2.1. Code Execution and Data Processing

• Library Importation: The script begins by importing essential libraries: NumPy for numerical operations, pandas for data manipulation, and matplotlib.pyplot for data visualization.
• Data Loading and Preparation:
 - The dataset is loaded from Q1.csv into a pandas DataFrame.
 - The data is then randomly shuffled to ensure that any inherent ordering does not bias the model training process. A random_state is used for reproducibility.
 - The feature matrix (X) is created by dropping the target variable column (Y). The target vector (Y) is selected as a separate column.
 - A column of ones is appended to the feature matrix X. This is a standard practice in linear regression to account for the intercept term.
• Data Splitting: The dataset is partitioned into three subsets:
 - Training Set (70%)
 - Validation Set (15%)
 - Testing Set (15%)

2.2. Model Implementation: Closed-Form Solution

The code implements the closed-form solution, also known as the Normal Equation, to find the optimal model parameters (θ).

Formula:
$\theta^* = (X^T X)^{-1} X^T Y$

np.linalg.pinv: This function calculates the **Moore-Penrose pseudo-inverse**. It is a generalization of the inverse that works for **any matrix**, including singular or non-square matrices. If a true inverse exists, pinv will calculate it. If the matrix is singular, pinv will still compute the best possible solution that minimizes the error.

This calculation directly computes the optimal theta values that minimize the Mean Squared Error (MSE) loss function without requiring an iterative optimization algorithm like Gradient Descent.

Final Training Error: 8.4737

Final Validation Error: 8.1844

Final Testing Error: 9.4695

2.3. Model implementation: Gradient decent

The **linear_regression_model** class provides a well-structured, object-oriented implementation for training a linear regression model.

The training logic is contained within the fit method. It implements the **Batch Gradient Descent** algorithm to find the optimal model weights.

- **Initialization:** Weights are initialized to zeros before training begins.
- **Iterative Updates:** For each epoch, the method updates the model's weights by taking a step in the opposite direction of the gradient of the Mean Squared Error (MSE) cost function.
- **Weight Update Formula:** The line self.weights-=(self.alpha*2/(m))*(xt.T@(xt@self.weights-yt)) is the vectorized implementation of this update rule, which is both computationally efficient and mathematically precise.

Built-in Visualization: The **errorplot** method provides a convenient way to visualize the training and validation error curves, making it easy to see how the model converged over the 30,000 epochs.

This class is a robust and complete implementation of a linear regression model. By training it for **30,000 epochs** with a learning rate (alpha) of **0.31**, you ensured that the Gradient Descent algorithm had sufficient iterations to converge very close to the optimal solution, as demonstrated by the comparison with the Normal Equation results.

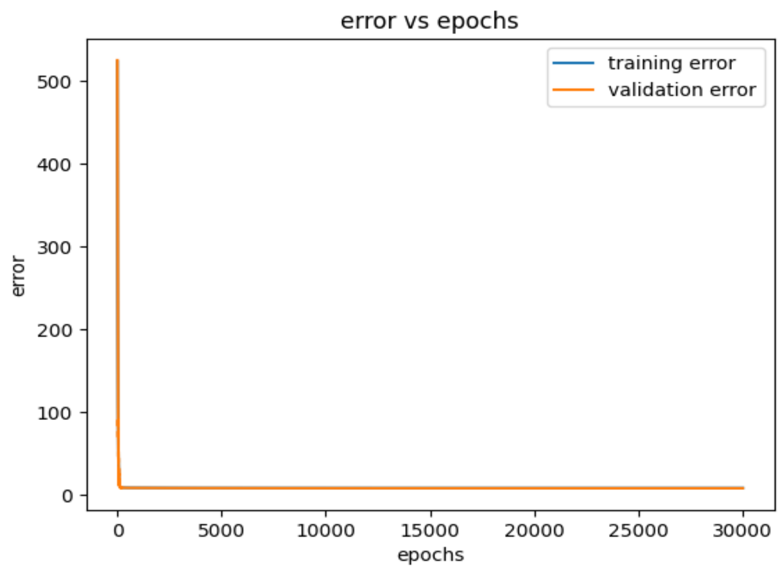
Final Training Error: 8.4737

Final Validation Error: 8.1866

Final Testing Error: 9.4695

2.4. Evaluation and Visualization

• Predictions: The trained model is used to make predictions on the training, validation, and testing sets.
• Error Calculation: The Mean Squared Error (MSE) is calculated for all three sets.



2.5. Comparison of Optimal Parameters: Normal Equation vs. Gradient Descent

- **Theta 1 (Normal Equation):** This method provides a direct, analytical solution.

[[-22.3568], [-10.6230], [2.1745], [-9.4712], [15.3864], [0.0732], [7.4290], [1.4995], [28.7283]]

- **Theta 2 (Gradient Descent):** This is an iterative method that approximates the solution. The result below was obtained after 30,000 epochs with a learning rate of 0.31.

[[-22.1687], [-10.4920], [2.2084], [-9.3764], [15.4331], [0.0731], [7.4293], [1.5002], [28.4876]]

3. Question 2: Logistic Regression (Q2.py)

3.1. Code Execution and Data Processing

• Library Importation: The script imports NumPy, pandas, and matplotlib.pyplot.
• Data Loading and Preparation:
 - The dataset is loaded from Q2.csv.
 - The data is shuffled, and columns are named Y, x1, and x2.
 - The target variable Y is mapped from -1 and 1 to 0 and 1.
 - The feature matrix X is created, with a column of ones appended.
• Data Splitting: The data is split into training (70%), validation (15%), and testing (15%) sets.

3.2. Gradient Descent Implementation: Logistic_regression_model Class

Gradient Descent is an iterative optimization algorithm used to minimize the cost function. The update rule is:

$\theta j := \theta j - \alpha\, \partial J(\theta)/\partial\theta j$

Class Object Analysis:
• __init__(self): Initializes learning rate and iterations.
• fit(self, X, Y, X_val, Y_val,Learning_rate,epochs): Main training loop, updates parameters and stores loss/accuracy.
• predict(self, X,Y, threshold): Generates predictions after training.

The primary training logic is housed within the fit method, which uses the **Gradient Descent** algorithm to learn the optimal model weights.

- **Initialization:** Before training, the model weights are initialized to a vector of zeros.
- **Iterative Optimization:** The method iterates for a specified number of epochs. In each epoch, it updates the weights by descending the gradient of the log-loss (binary cross-entropy) cost function.
- **Weight Update Formula:** The update rule :

  self.weights-=(self.lr/(m))*(xt.T@(H(xt,self.weights)-yt)) is a vectorized implementation that efficiently computes the gradient across the entire training batch (xt) and adjusts the weights. **H(xt, self.weights)** represents the sigmoid function applied to the linear combination of inputs and weights.

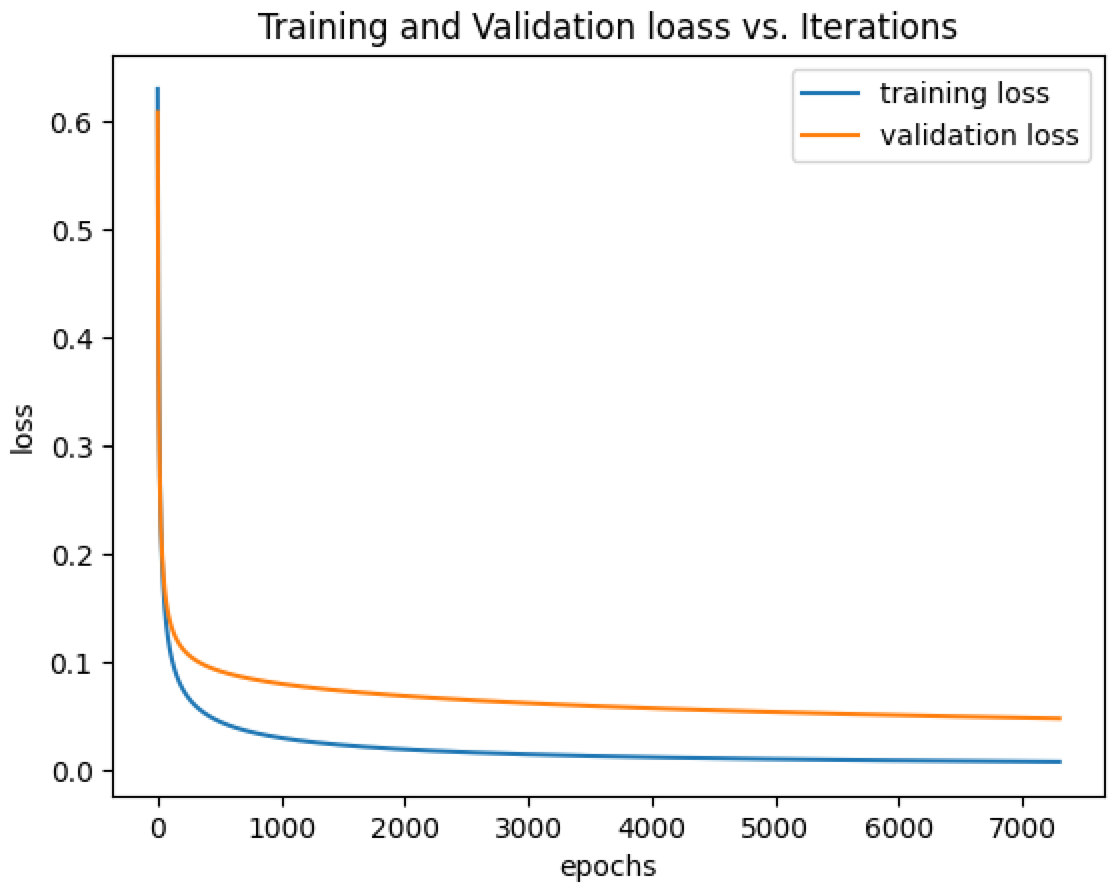
The class is a complete toolkit, featuring a suite of methods for robust model evaluation:

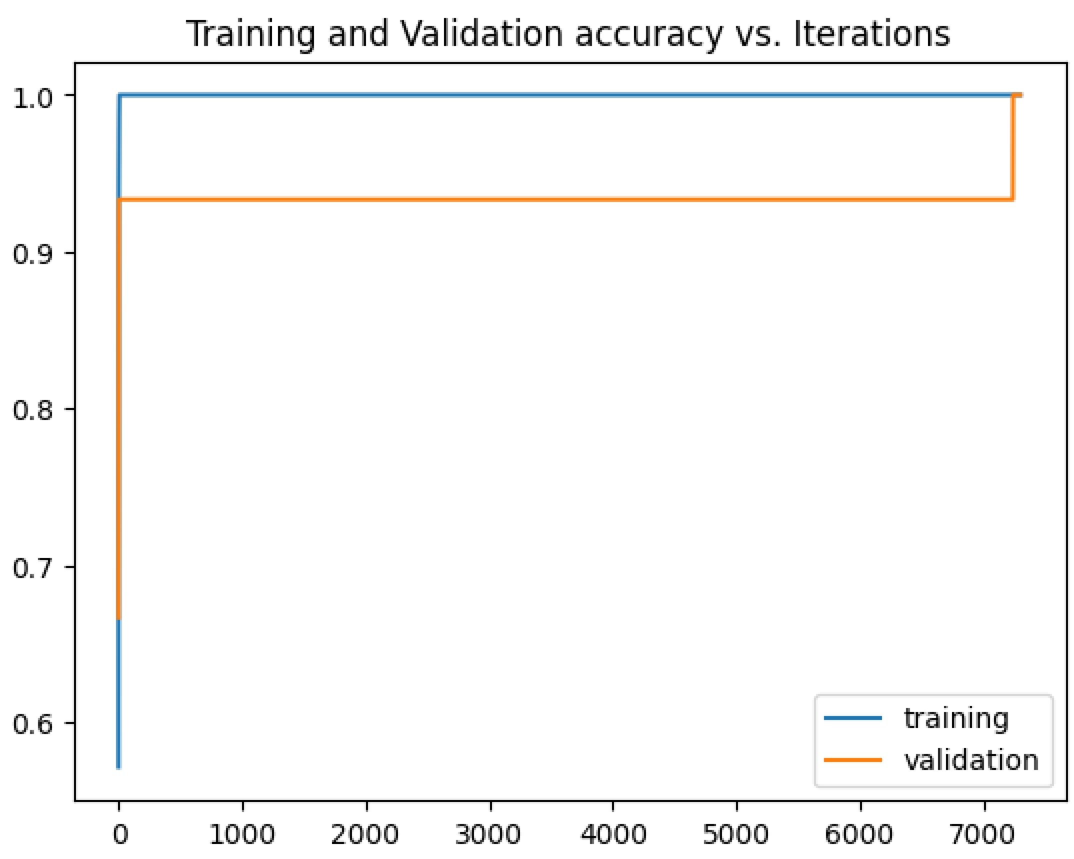
- **Accuracy Calculation:** The accuracy_score method computes the classification accuracy by comparing the thresholded predictions against the true labels.

- **Confusion Matrix:** The confusion_matrixfn method provides a detailed breakdown of classification performance into true positives, true negatives, false positives, and false negatives.
- **Precision and Recall:** The class includes dedicated methods (precision_for_binary_classification and recall_for_binary_classification) to calculate these essential classification metrics directly from the confusion matrix.
- **Built-in Visualization:** The lossplot and plot_training_validation_accuracy methods offer a straightforward way to generate plots of the key performance metrics over epochs, making it easy to visualize model convergence and check for overfitting.

3.3. Model Training and Evaluation

• Instantiation and Training: GradientDescent object created with learning rate 0.1 and 7300 iterations.





• Final Evaluation: Predictions made on test set. Metrics include:

Accuracy :

1. Training accuracy : 100%
2. Validation accuracy: 100%
3. Testing accuracy :100%

Confusion matrix :

1. training set confusion matrix:

	TRUE 0	TRUE 1
PRED 0	37	0
PRED 1	0	33

2. validation set confusion matrix

	TRUE 0	TRUE 1
PRED 0	9	0
PRED 1	0	6

3. testing set confusion matrix

	TRUE 0	TRUE 1
PRED 0	4	0
PRED 1	0	11

percision/recall/F1:

training set :

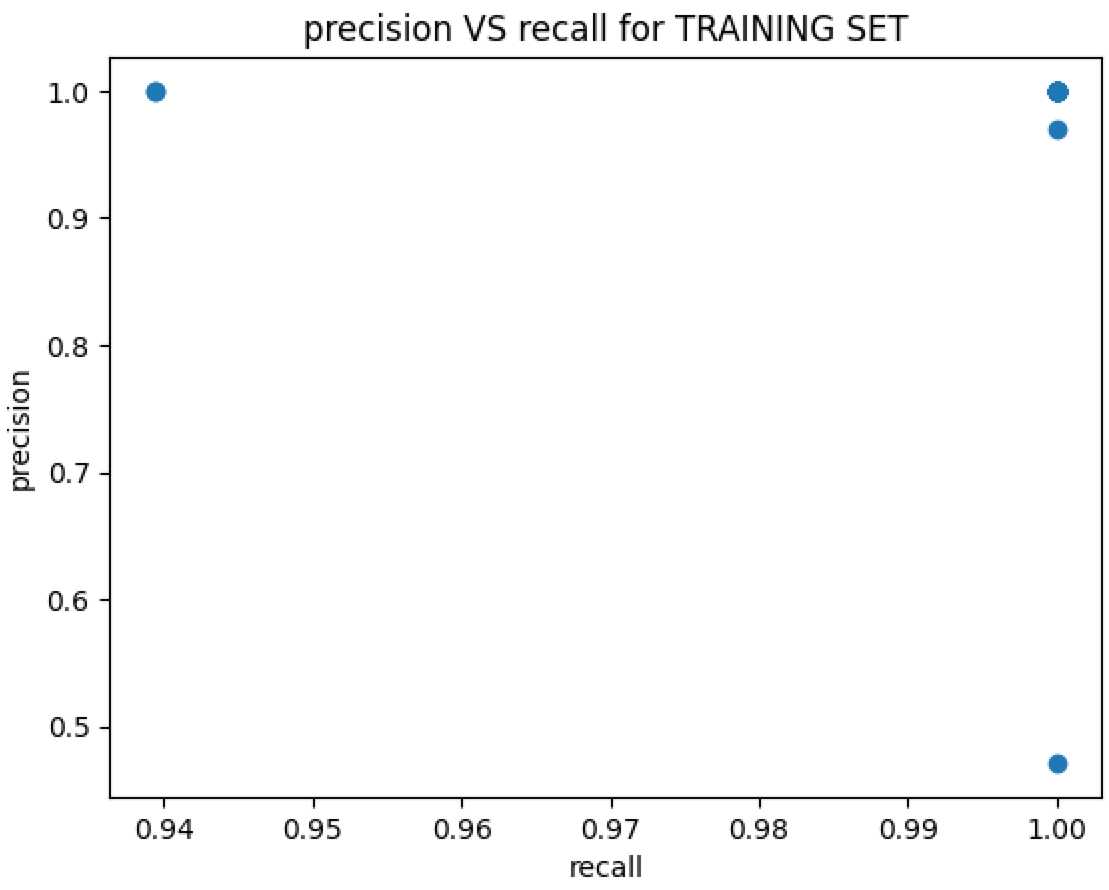
- precision : 1.0
- recall 1.0
- F1 SCORE: 1.0

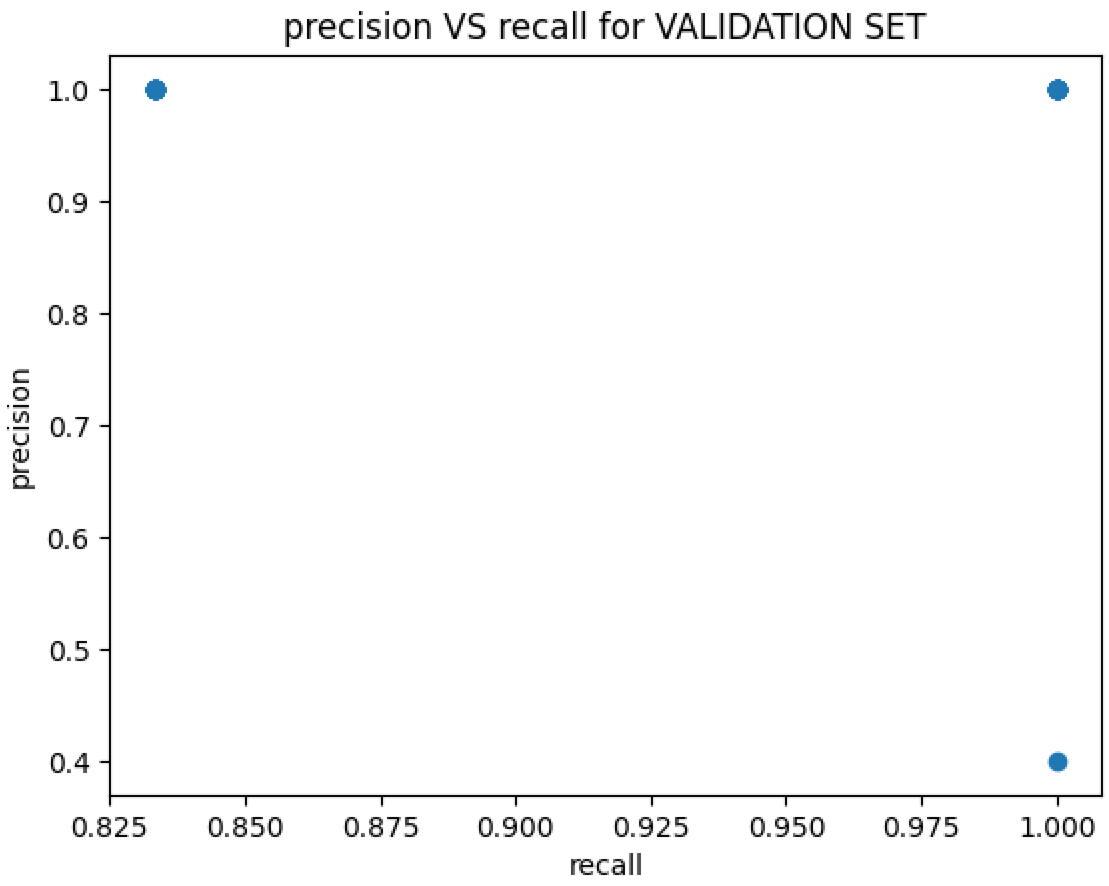
Validation set :

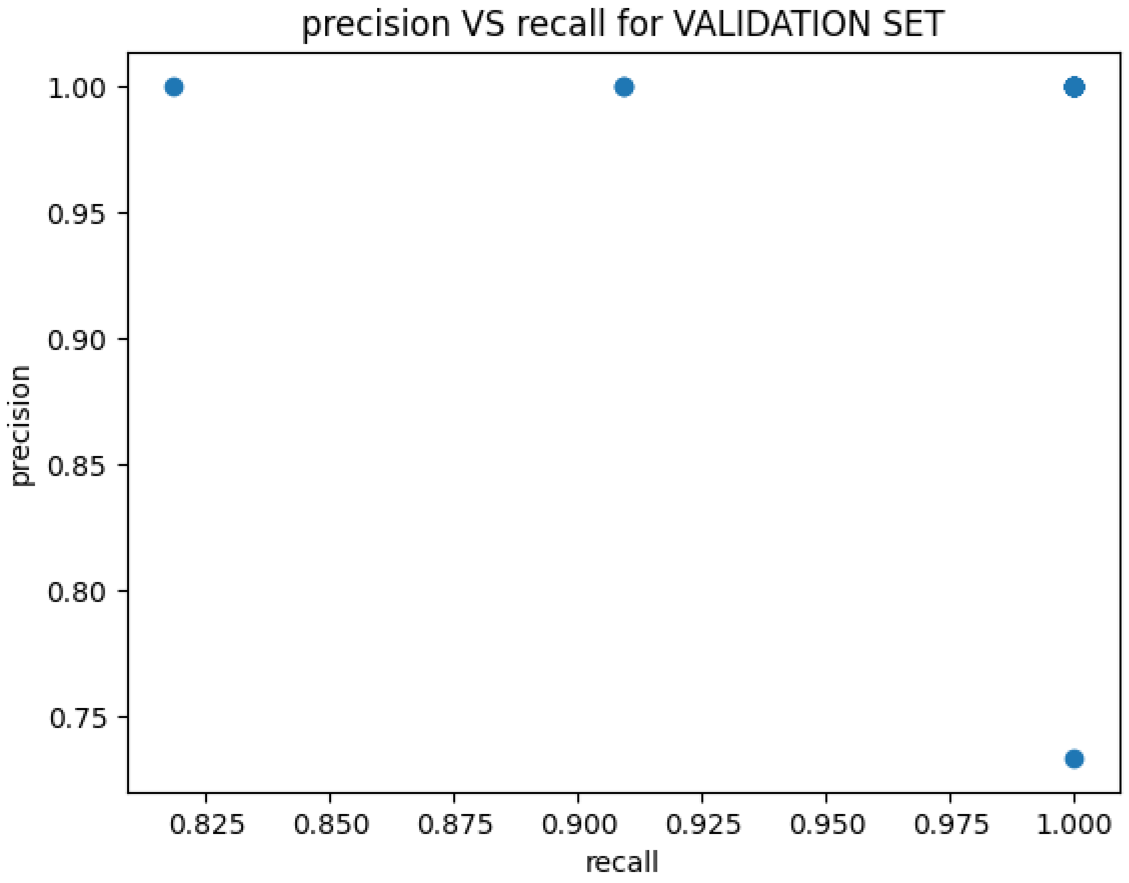
- precision : 1.0
- recall 1.0
- F1 SCORE: 1.0

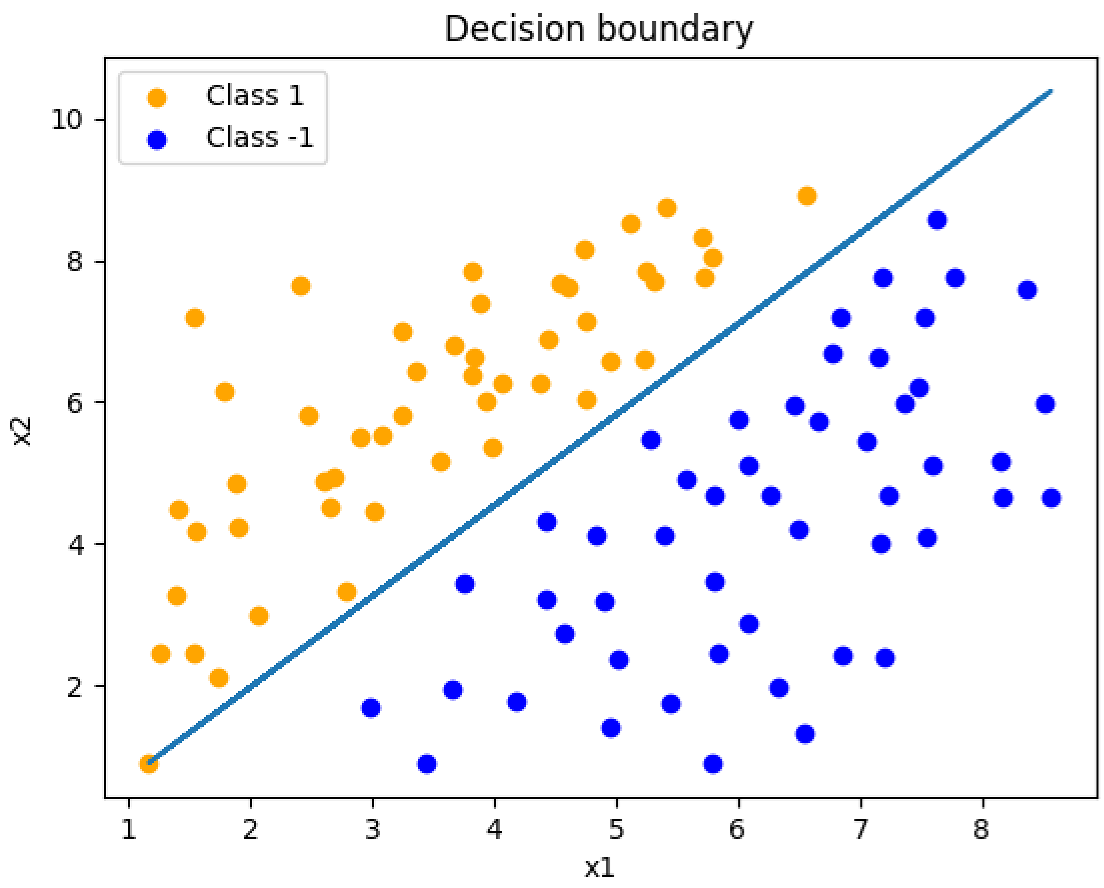
Testing set :

- precision : 1.0
- recall 1.0
- F1 SCORE: 1.0


precision VS recall for TRAINING SET
precision
recall
1.0
0.9
0.8
0.7
0.6
0.5
0.94
0.95
0.96
0.97
0.98
0.99
1.00


precision VS recall for VALIDATION SET
precision
recall
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.825
0.850
0.875
0.900
0.925
0.950
0.975
1.000


precision VS recall for VALIDATION SET
precision
1.00
0.95
0.90
0.85
0.80
0.75
recall
0.825
0.850
0.875
0.900
0.925
0.950
0.975
1.000



4. Conclusion

The provided code successfully implements both a linear regression model using a closed-form solution and a logistic regression model using an object-oriented Gradient Descent algorithm. The implementations follow standard practices, including preprocessing, splitting, training, and evaluation. The GradientDescent class is modular, encapsulating all training and prediction logic, making the code easy to understand and extend.