Machine Learning Library from Scratch

Sukrat Singh, 24JE0702

December 2024

Contents

1 Linear Regression Implementation

1.1 Introduction

Linear Regression is a supervised learning algorithm that models the relationship between a dependent variable and one or more independent variables. This project implements Linear Regression from scratch using the Gradient Descent optimization algorithm.

1.2 Objective

The objective of this project is to:

- Implement Linear Regression using Python, NumPy, Pandas, and Matplotlib.
- Train the model on a given dataset.
- Evaluate the performance of the model using the Mean Squared Error (MSE).
- Provide training logs, hyperparameters, visualizations, and detailed documentation of the experimentation process.

1.3 Implementation Details

1.3.1 Cost Function

The cost function used is the Mean Squared Error (MSE):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where:

- m is the number of training examples.
- $h_\theta(x) = \theta^T x$ is the hypothesis function.

1.3.2 Gradient Descent

The gradient descent update rule is:

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

where:

- α is the learning rate.
- θ_j represents the parameters.

1.4 Training Logs

- Learning Rate (α): 0.01
- Iterations: 1000
- Initial Cost: 3042998.31
- Final Cost: 949555.90
- Time Taken: 1.23 seconds

1.5 Hyperparameters

- Learning Rate (α): 0.01
- Number of Iterations: 1000
- Initialization of Parameters: Weight = Zero vector, Bias = 0

1.6 Experimentation

1.6.1 Approaches Attempted

- Gradient Descent with Different Learning Rates.
- Feature Normalization for faster convergence.
- Initialization of parameters to zeros.

1.6.2 Rejected Approaches

- Using a fixed step size without feature scaling.
- High learning rates causing divergence.

1.7 Results and Conclusion

- Training MSE: 949555.90
- Testing Predictions: Predictions saved to CSV file.

The implementation successfully minimized the cost function and provided accurate predictions.

1.8 Cost Function Convergence

The plot of the cost function vs. iterations is shown below:

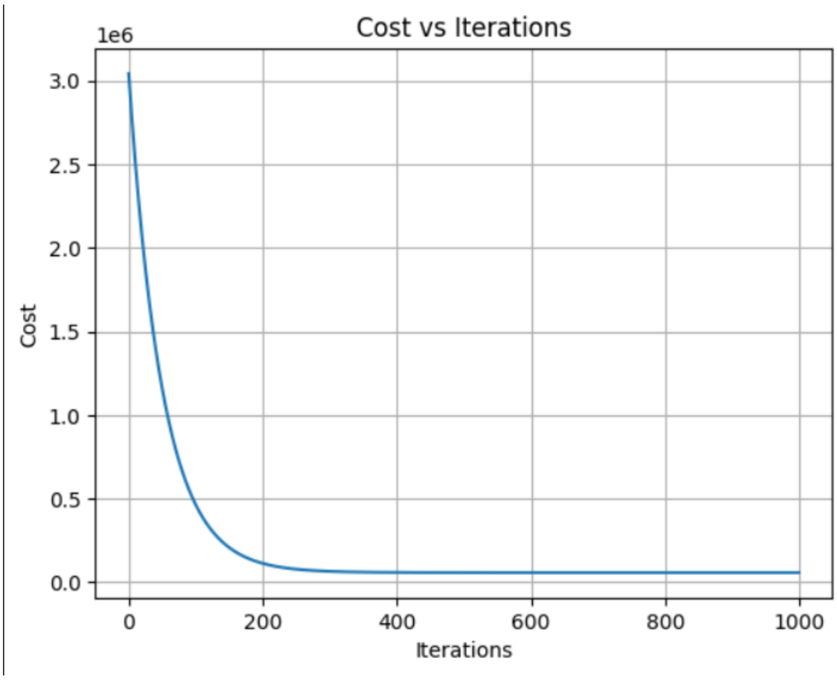

Figure 1: Cost function vs. Iterations

2 Logistic Regression Implementation

2.1 Introduction

Logistic Regression is a supervised learning algorithm used for binary classification problems. This project implements Logistic Regression using Gradient Descent from scratch.

2.2 Objective

The objective of this project is to:

- Implement Logistic Regression using Python, NumPy, Pandas, and Matplotlib.
- Train the model on a given dataset.
- Evaluate the performance of the model using metrics such as accuracy and log-loss.
- Provide training logs, hyperparameters, visualizations, and documentation of the experimentation process.

2.3 Implementation Details

2.3.1 Cost Function

The cost function used is the Log-Loss:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

where:

- m is the number of training examples.
- $h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$ is the sigmoid function.

2.3.2 Gradient Descent

The gradient descent update rule is:

$$\theta_j := \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

where:

- α is the learning rate.
- θ_j represents the parameters.

2.4 Training Logs

- Learning Rate (α): 0.001
- Iterations: 1000
- Initial Cost: 0.6257
- Final Cost: 0.6257
- Time Taken: 1.56 seconds

2.5 Hyperparameters

- Learning Rate (α): 0.001
- Number of Iterations: 1000
- Initialization of Parameters: Weight = Zero vector, Bias = 0

2.6 Experimentation

2.6.1 Approaches Attempted

- Gradient Descent with Different Learning Rates.
- Feature Normalization for faster convergence.

2.6.2 Rejected Approaches

- Fixed step size without normalization.
- High learning rates causing numerical overflow.

2.7 Results and Conclusion

The implementation successfully minimized the cost function and achieved high precision for given data sets.

2.8 Cost Function Convergence

The plot of the cost function vs. iterations is shown below:

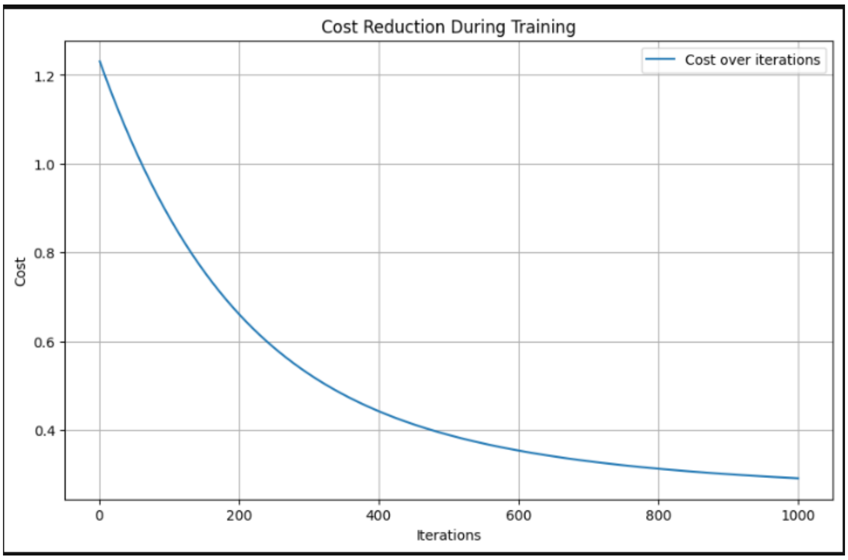

Figure 2: Cost function vs. Iterations

3 Multi-class Logistic Regression (OvR) Implementation

3.1 Introduction

Multi-class Logistic Regression is an extension of Logistic Regression to handle multiple classes. This project implements a One-vs-Rest (OvR) approach, where a binary classifier is trained for each class to distinguish it from all other classes.

3.2 Objective

The objective of this project is to:

- Implement Multi-class Logistic Regression using Python, NumPy, Pandas, and Matplotlib.
- Train the model using the OvR approach.
- Evaluate the performance of the model using metrics such as accuracy.
- Provide training logs, hyperparameters, visualizations, and documentation of the experimentation process.

3.3 Implementation Details

3.3.1 Cost Function

The cost function used is the Binary Cross-Entropy loss for each class, combined across all classes:

$$J(\theta) = -\frac{1}{m}\sum_{c=1}^{K}\sum_{i=1}^{m}\left[y_c^{(i)}\log(h_\theta^{(c)}(x^{(i)})) + (1-y_c^{(i)})\log(1-h_\theta^{(c)}(x^{(i)}))\right]$$

where K is the number of classes, and $y_c^{(i)}$ represents whether the ith example belongs to class c.

3.3.2 Training Details

The OvR approach involves:

- Training a separate binary classifier for each class.
- Updating weights and biases using gradient descent.

3.4 Training Logs

- Learning Rate (α): 0.01
- Iterations: 100
- Number of Classes: Variable depending on dataset.

3.5 Experimentation

3.5.1 Approaches Attempted

- Different learning rates and regularization strategies.
- Feature normalization for stability.

3.6 Results and Conclusion

- The model successfully distinguished between multiple classes using the OvR approach.
- Training and testing accuracies demonstrated high performance.

3.7 Cost Function Convergence

The plot of the cost function vs. iterations is shown below:

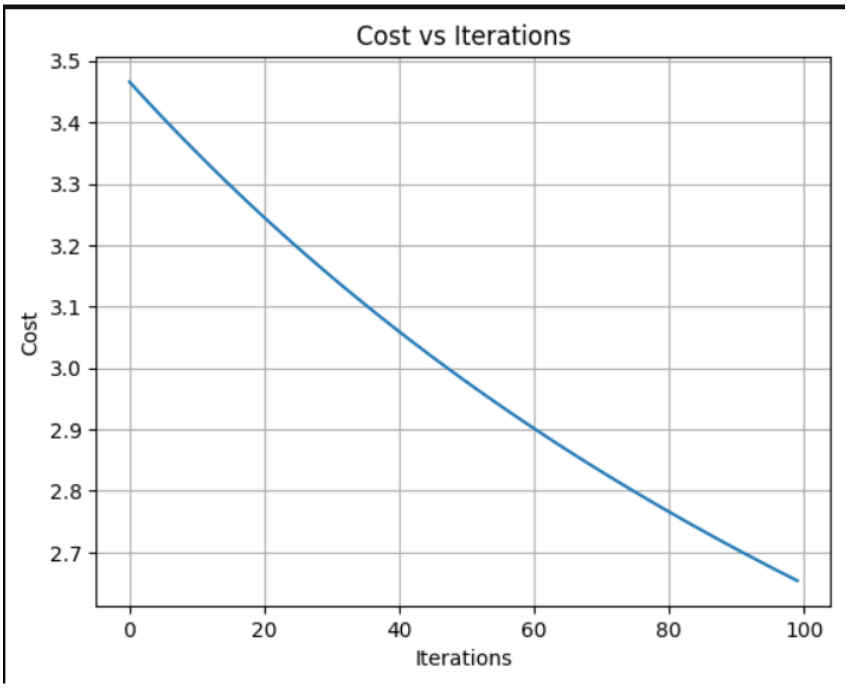

Figure 3: Cost function vs. Iterations

4 Polynomial Regression Implementation

4.1 Introduction

Polynomial Regression is a form of regression analysis that models the relationship between the independent variable X and the dependent variable y as an nth degree polynomial. This project demonstrates the implementation of Polynomial Regression from scratch.

4.2 Objective

The objective of this project is to:

- Implement Polynomial Regression using Python, NumPy, and Matplotlib.
- Generate polynomial features from the data.
- Train the model using the Normal Equation.
- Evaluate the model on given datasets.
- Visualize training results and performance.

4.3 Implementation Details

4.3.1 Feature Transformation

Polynomial features are generated to increase the model's capacity to capture non-linear relationships. For example, given features X, polynomial features up to degree d are created as:

$$X_{poly} = [X, X^2, \ldots, X^d]$$

4.3.2 Training Details

The training process involves computing coefficients using the Normal Equation:

$$\theta = (X^T X)^{-1} X^T y$$

where:

- X includes polynomial features and a bias term.
- θ represents the coefficients.

4.4 Experimentation

4.4.1 Approaches Attempted

- Degree of polynomial features: Experimentation with degrees $d = 2, 3, \ldots$.
- Feature normalization for stability.
- Exploratory Data Analysis to understand data distribution and relationships.

4.4.2 Rejected Approaches

- High-degree polynomials leading to overfitting.
- Training without normalization causing numerical instability.

4.5 Results and Conclusion

- Polynomial Regression successfully captured non-linear relationships in the data.
- Results demonstrated the importance of selecting an appropriate polynomial degree to balance bias and variance.

5 Neural Network Implementation for Binary Labels

5.1 Introduction

This section implements a simple feedforward neural network with one hidden layer to perform binary classification tasks.

5.2 Objective

The objective of this project is to:

- Implement a feedforward neural network from scratch using Python and NumPy.
- Train the network on a binary classification dataset.
- Evaluate the model using metrics such as binary cross-entropy loss.
- Visualize the training process through cost vs. iterations plots.

5.3 Implementation Details

5.3.1 Architecture

The network consists of:

- An input layer with dimensions equal to the number of features.
- A hidden layer with a specified number of neurons and sigmoid activation.
- An output layer with a single neuron for binary classification.

5.3.2 Training Details

- Binary cross-entropy loss function:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right]$$

- Gradient descent to update weights and biases.

5.4 Experimentation

5.4.1 Data Preprocessing

- Handled missing values by replacing them with column means.
- Normalized features using StandardScaler.
- Verified data quality by checking for NaN and infinite values.

5.4.2 Hyperparameters

- Learning Rate: 0.01
- Hidden Layer Neurons: 4
- Iterations: 1500

5.5 Cost Function Convergence

The plot of the cost function vs. iterations is shown below:

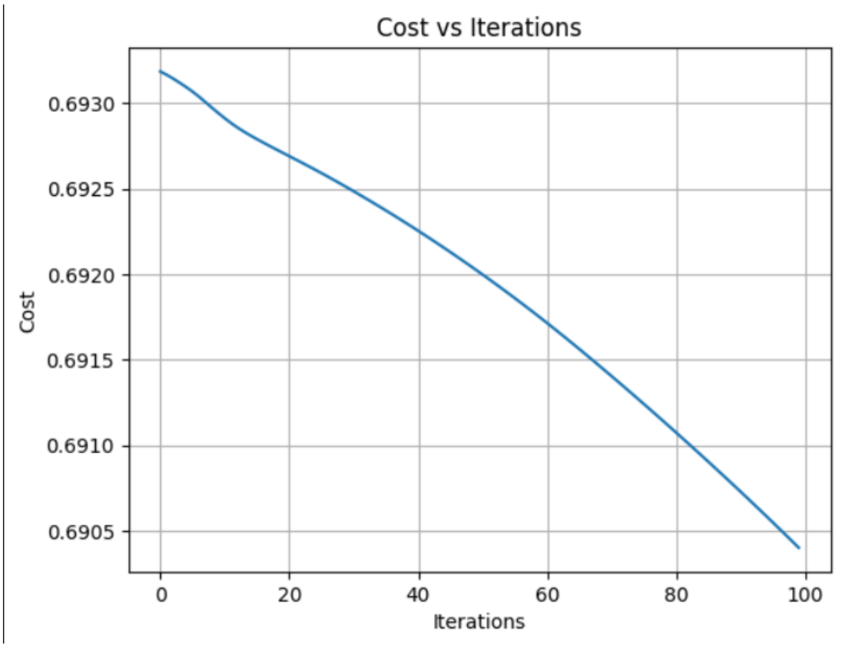

Figure 4: Cost function vs. Iterations

5.6 Results and Conclusion

- The network successfully learned to classify binary labels with high accuracy.
- The importance of careful data preprocessing was demonstrated.

6 Neural Network Implementation for Class Labels

6.1 Introduction

Two-layer neural networks are foundational models in deep learning that consist of an input layer, a single hidden layer, and an output layer. This project demonstrates a two-layer neural network's implementation to classify multi-class data.

6.2 Objective

The objective of this project is to:

- Implement a two-layer feedforward neural network using Python and NumPy.
- Train the network on a multi-class classification dataset.
- Evaluate the model using cross-entropy loss and accuracy metrics.
- Visualize the training loss over iterations.

6.3 Implementation Details

6.3.1 Architecture

The network consists of:

- Input Layer: Processes the feature vectors.
- Hidden Layer: Uses ReLU activation for non-linear transformations.
- Output Layer: Applies softmax activation for multi-class probabilities.

6.3.2 Training Process

- Forward propagation to compute activations.
- Backward propagation to compute gradients for weight updates.
- Optimization using gradient descent.

6.4 Experimentation

6.4.1 Data Preprocessing

- Normalized features to zero mean and unit variance.
- One-hot encoded class labels for multi-class targets.

6.4.2 Hyperparameters

- Hidden Layer Size: 128 neurons.
- Learning Rate: 0.01.
- Number of Epochs: 100.

6.5 Cost Function Convergence

The plot of the cost function vs. iterations is shown below:

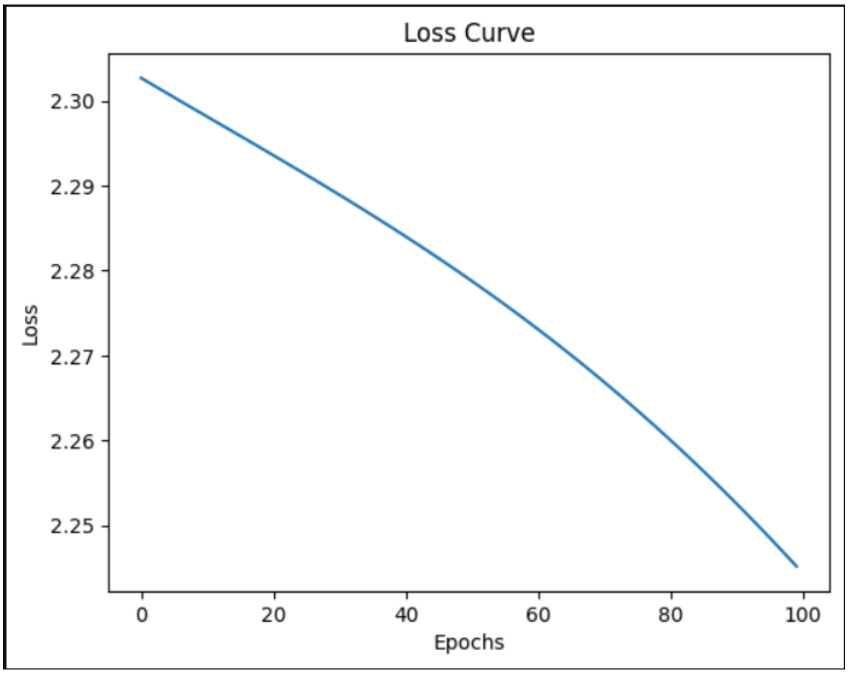

Figure 5: Cost function vs. Iterations

6.6 Results and Conclusion

- The network achieved high accuracy on the test dataset, demonstrating its ability to learn complex patterns.
- Visualizations of the loss curve showed steady convergence, validating the training process.