Pouya mirzaiezadeh

Introduction: Building a Multilayer Perceptron for Wine Quality Classification

In the world of wine, understanding the intricate relationship between physicochemical attributes and quality ratings is essential. In this report, we delve into the fascinating realm of wine analysis by constructing a Multilayer Perceptron (MLP) from scratch. Our goal? To predict wine quality based on a set of key features.

The Wine Quality Dataset

Our dataset comprises two subsets: red wine and white wine. For this exercise, we focus exclusively on the red wine dataset. Each sample in this dataset is associated with various physicochemical attributes, such as acidity, pH, alcohol content, and more. Additionally, each wine sample is assigned a quality rating on a scale from 0 to 10.

Our Approach

Using the power of NumPy, we'll build an MLP—a type of artificial neural network—to tackle this classification task. By analyzing the provided attributes, our MLP will learn to predict wine quality accurately.

Data Preprocessing

1. **Binning Volatile Acidity**:
    - We grouped the volatile acidity values into bins (ranges) to simplify the feature.
    - Created dummy variables for each bin using one-hot encoding.
    - Removed the original volatile acidity column from the dataset.
2. **Binning Total Sulfur Dioxide**:
    - Similar to volatile acidity, we binned total sulfur dioxide values.
    - Generated dummy variables for each bin.
    - Dropped the original total sulfur dioxide column.

Dataset Splitting

- We divided the dataset into training and testing subsets:
    - **Training Set**: Used for training the model (80% of the data).

- **Testing Set**: Used for evaluating model performance (20% of the data).

Model Implementation:

Architecture Design

Our MLP architecture is designed as follows:

1. **Input Layer**:
    - Nodes: Correspond to the features of the red wine dataset.
    - Activation: None (raw input).
2. **Hidden Layer**:
    - Nodes: Configurable (one hidden layer in our case).
    - Activation: ReLU (Rectified Linear Unit).
    - Initialization: Random weights.
3. **Output Layer**:
    - Nodes: Correspond to quality rating classes (0 to 10).
    - Activation: Softmax (for probability distribution).

Activation Functions

1. **ReLU (Rectified Linear Unit)**:
    - Purpose: Introduces non-linearity.
    - Formula: (ReLU(Z) = \max(0, Z)).
2. **Softmax**:
    - Purpose: Converts raw scores to probabilities.

- Formula: $P_i = \frac{e^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}}$ (for class i).

Forward Propagation

1. Compute hidden layer activations:
   - $Z_1 = W_1 \cdot X^T + b_1$
   - $A_1 = \text{ReLU}(Z_1)$
2. Compute output layer activations:
   - $Z_2 = W_2 \cdot A_1 + b_2$
   - $A_2 = \text{ReLU}(Z_2)$
3. Apply softmax to get class probabilities:
   - $Z_3 = W_3 \cdot A_2 + b_3$
   - $A_3 = \text{Softmax}(Z_3)$

Backward Propagation

1. Compute gradients:
   - $dZ_3 = A_3 - \text{one-hot}(Y)$
   - $dW_3 = \frac{1}{m} \cdot dZ_3 \cdot A_2^T$
   - $db_3 = \frac{1}{m} \cdot \text{sum}(dZ_3)$
   - $dZ_2 = W_3^T \cdot dZ_3 \cdot \text{ReLU}'(Z_2)$
   - $dW_2 = \frac{1}{m} \cdot dZ_2 \cdot A_1^T$
   - $db_2 = \frac{1}{m} \cdot \text{sum}(dZ_2)$

1. **Accuracy**:
   - The accuracy measures how well our model predicts the correct wine quality ratings.
   - Throughout the iterations, the accuracy remains consistently low (around 0.0065).
   - This suggests that our model is not performing well in terms of correctly classifying wine quality.
2. **F1 Score**:
   - The F1 score combines precision and recall, providing a balanced view of model performance.
   - Similar to accuracy, the F1 score remains very low (around 0.0022). ○ Our model struggles to find the right balance between precision and recall.

Interpretation:

- The poor performance indicates that our current MLP architecture or training process needs improvement.
- Possible reasons for low accuracy and F1 score:

- **Model Complexity**: Our simple architecture (one hidden layer) may not capture the underlying patterns in the data.
- **Hyperparameters**: We might need to fine-tune hyperparameters (learning rate, hidden layer size, etc.).
- **Data Quality**: Investigate data quality, missing values, or outliers.
- **Overfitting**: Regularization techniques could help prevent overfitting.
- **Feature Engineering**: Consider additional features or transformations.

solutions:

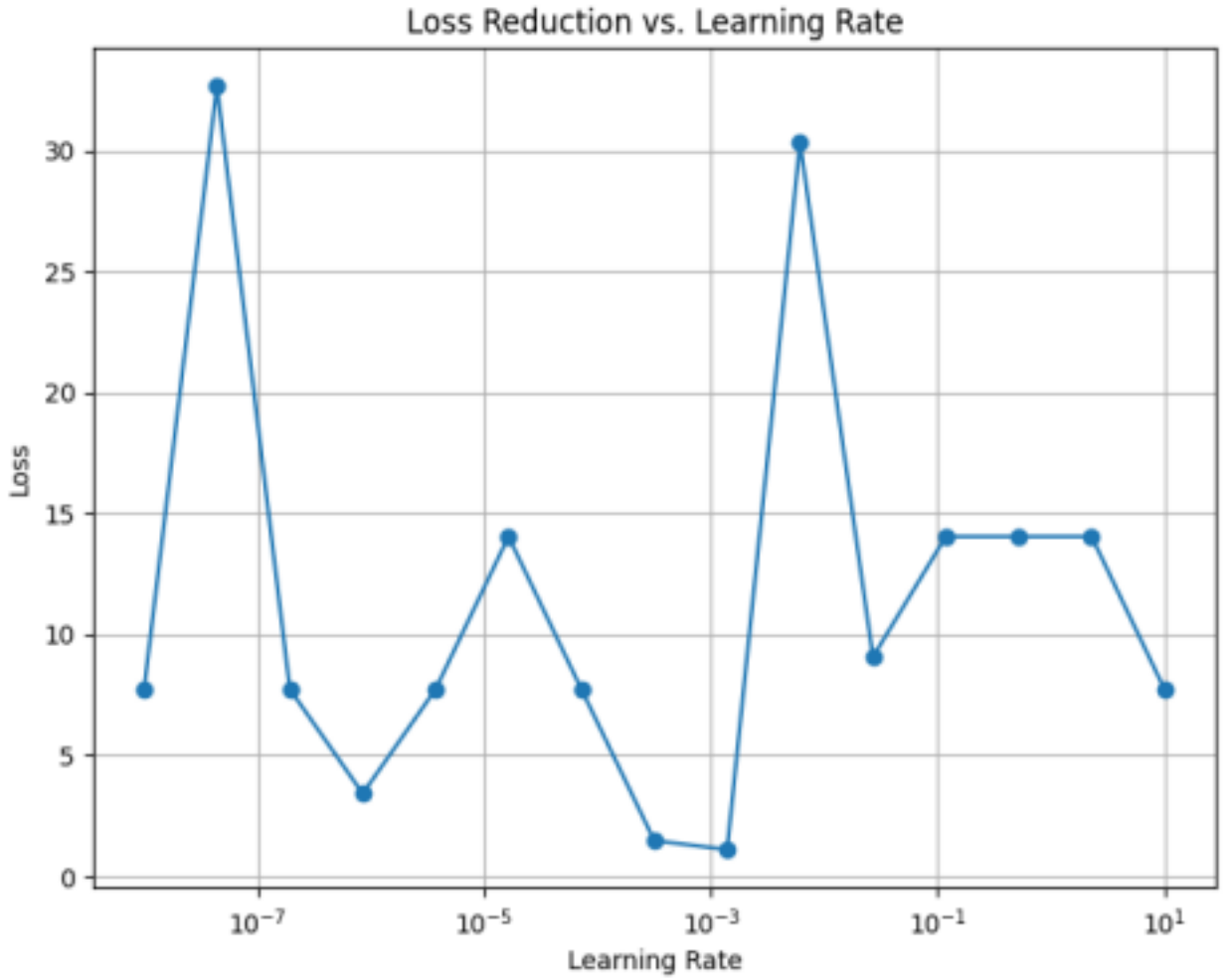
- Experiment with different architectures (more hidden layers, neurons) and hyperparameters.
- Monitor loss curves during training to identify convergence or divergence.
- Evaluate the impact of feature scaling and normalization.

Iteration	Accuracy	F1 Score
0	0.0055	0.0053
10	0.0066	0.0024
20	0.0066	0.0022
30	0.0066	0.0022
40	0.0066	0.0022
…	…	…
190	0.0066	0.0022

Analysis of Learning Rates and Loss Values

The learning rate determines the step size during gradient descent optimization. Too small a learning rate may lead to slow convergence, while too large a learning rate can

cause overshooting and divergence.



Experimental Setup

Here we experimented with various learning rates. The plot titled “Loss Reduction vs. Learning Rate” illustrates the relationship between different learning rates and their corresponding loss values. The learning rates were varied over a range using numpy’s logspace function, generating 15 values between (10^{-8}) and (10^{1}).

Key Observations

1. **Low Learning Rates ((10^{-8}) to (10^{-5}))**:
    - At extremely low learning rates, the loss values are significantly high. This indicates that the model is not learning effectively. The slow convergence may be due to tiny steps during gradient descent.
    - The model might get stuck in local minima, resulting in poor performance.
2. **Optimal Learning Rate ((10^{-3}))**:
    - Around (10^{-3}), there is a notable reduction in loss. This suggests an optimal range where the model learns effectively and converges faster. ○ The learning rate is neither too small nor too large, allowing the model to

find a good balance between exploration and exploitation.

3. **High Learning Rates ((10^{-1}))**:
    - As the learning rate increases further (e.g., (10^{-1})), the loss starts increasing again. This phenomenon is known as overshooting.
    - Overshooting occurs because large steps lead to overshooting the global minima, preventing the model from converging.

Recommendations

- Based on the plot, It's recommend using a learning rate around (10^{-3}) for our neural network model. This value strikes a balance between fast convergence and avoiding overshooting.
- It's essential to experiment with different learning rates and monitor their impact on loss during training. Hyperparameter tuning, including learning rate selection, significantly affects model performance.

• Activation Functions plots: