Binary Classification Using Logistic Regression and MLP

1st Zhiling LI
Department of System Design and Intelligent Manufacturing
Southern University of Science and Technology
Shenzhen, China
12112748@mail.sustech.edu.cn

***Abstract*—This study delves into the development and evaluation of binary classification models for solving binary classification problems. Specifically, the Logistic Regression model and the Multi-Layer Perceptron (MLP) model are implemented using Python and Numpy. The MLP model offers flexibility in configuring the number of hidden layers and neurons. The impact of varying hyperparameters on model performance is observed, alongside the evaluation of these models using metrics such as Recall, Precision, and F1 score.**

***Index Terms*—Logistic Regression, MLP, Binary Classification**

I. Introduction

Binary classification is a fundamental task in the field of machine learning, with applications spanning across various domains. The ability to categorize data into one of two classes is a fundamental building block for more complex machine learning tasks. This research delves into the development and evaluation of binary classification models, with a specific focus on two prominent approaches: Logistic Regression and Multi-Layer Perceptron (MLP) neural networks.

The Logistic Regression model is a well-established method for binary classification, known for its simplicity and interpretability. It models the probability of an instance belonging to a specific class and provides insights into the relationship between input features and the binary outcome.

On the other hand, the MLP neural network represents a more flexible and expressive approach to binary classification. MLPs consist of multiple layers, including input, hidden, and output layers, allowing for nonlinear relationships to be captured within the data. The number of hidden layers and neurons within these layers can be adjusted, making MLPs highly adaptable to various classification tasks.

In this research, we not only implement these models but also examine their performance under different configurations. We create binary classification datasets. We employ key metrics such as Recall, Precision, and F1 score to assess the models' ability to correctly classify positive and negative instances.

Eventually, we explore the impact of hyperparameters, such as learning rate and model frame, on the training process and final model performance. We analyze how adjustments to these hyperparameters affect the stability of the models.

II. Problem formulation

A. Performance Metrics

Precision:

$$P = \frac{TP}{TP + FP} \tag{1}$$

Recall:

$$R = \frac{TP}{TP + FN} \tag{2}$$

F1 Score:

$$F1 = \frac{2 \cdot P \cdot R}{P + R} \tag{3}$$

where, TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives.

B. Activation Functions

Sigmoid:

$$f(z) = \frac{1}{1 + e^{-z}} \tag{4}$$

Relu:

$$f(z) = \max(0, z) \tag{5}$$

III. The Implementation of Logistic Regression Model

A. Logistic Regression Principles

Logistic Regression is a widely used technique in machine learning, particularly for binary classification problems. It differs from linear classification mainly due to its use of the sigmoid function as a link function, which transforms the linear combination of input features into a probability between 0 and 1.

The sigmoid function (σ) is defined as equation (4). This function offers differentiability, making it suitable for gradient-based optimization, and it ensures that the output is confined to the range of (0, 1), allowing it to represent probabilities.

In Logistic Regression, the likelihood function ($L(w)$) is central to the loss calculation. Assuming that the training examples are sampled independent and identically distributed, we can write the likelihood function:

$$L(w) = \prod_{i=1}^{N} p(t^{(i)}|x^{(i)}; w)$$

We can convert the maximization problem into minimization so that we can write the loss function:

$$\begin{aligned} l_{log}(w) &= -logL(w) \\ &= -\sum_{i=1}^{N} t^{(i)} log(1 - p(C = 0|x^{(i)}; w)) \\ &\quad -\sum_{i=1}^{N} (1 - t^{(i)}) log\ p(C = 0|x^{(i)}; w) \end{aligned}$$

which is a convex function of w.

This loss function penalizes discrepancies between predicted probabilities and actual labels.

To optimize the model, gradient-based optimization algorithms are typically used. The gradient of the loss with respect to model parameters (w_j) is computed as:

$$\frac{\partial l}{\partial w_j} = \sum_i x_j^{(i)} (t^{(i)} - p(C = 1|x^{(i)}; w))$$

This gradient guides parameter updates during training, ensuring convergence to a configuration that minimizes the loss. And the gradient descent for logistic regression:

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \sum_i x_j^{(i)} \left(t^{(i)} - p(C = 1|x^{(i)}; w) \right)$$

Logistic Regression's simplicity and interpretability make it a valuable tool and the most basic implementation for binary classification tasks.

B. Code Implementation

1) Model Class Implementation: In this section, we will provide an overview of the code implementation of Logistic Regression. The code structure diagram of the class implementation is as follows:

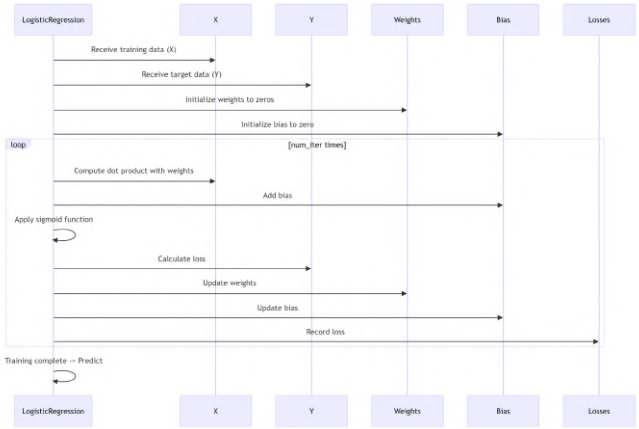

Fig. 1. Logistic Regression Class

a) The constructor `__init__` initializes an instance of the `LogisticRegression` class with the following parameters:

- `self.lr`: Learning rate, with a default value of 0.01.
- `num_iter`: Number of iterations for training the model.
- `weights`: The weight coefficients for features.
- `bias`: The bias term is initially set to 'None'.
- `losses`: An empty list to store the loss values during training.

```python
def __init__(self, lr=0.01, num_iter=2333):
    self.lr = lr
    self.num_iter = num_iter
    self.weights = None
    self.bias = None
    self.losses = []
```

Fig. 2. Init Function

b) Sigmoid Function

The `_sigmoid` method implements the sigmoid activation function, defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

This function maps the input z to a value in the range $[0, 1]$ and is used to model the probability of the positive class in logistic regression.

c) Loss Function

The `loss` method calculates the binary cross-entropy loss between predicted values y_{pred} and true labels y. It clips y_{pred} to ensure stability, computes the loss using the binary cross-entropy formula, and returns the mean of the calculated loss.

```python
def loss(self, y_pred, y):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)  # avoid log(0)
    return -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
```

Fig. 3. Loss Method

d) Model Training

The `fit` function in the logistic regression class is responsible for training the model using the input data X and target labels y. It iteratively updates these parameters to minimize the logistic loss function. In each iteration:

It computes the predicted values `y_predicted` using the sigmoid activation function. Then, calculates the gradients `dw` and `db` of the logistic loss. After that, Updates the weights and bias using gradient descent with a learning rate of `self.lr` and records the loss for monitoring during training.

e) Prediction

The `predict` method makes predictions for new data points represented by the feature matrix X. It follows these steps: Compute the weighted sum of input features and bias: $z\_pred = X \cdot \text{weights} + \text{bias}$. And then, Apply the sigmoid activation function to z_pred to obtain the predicted probabilities: $y\_predicted = \sigma(z\_pred)$. Last, threshold the predicted probabilities at 0.5 to classify data points into binary classes (0 or 1).

```python
def fit(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)
    self.bias = 0

    for _ in range(self.num_iter):
        z_pred = np.dot(X, self.weights) + self.bias
        y_predicted = self._sigmoid(z_pred)

        dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
        db = (1 / n_samples) * np.sum(y_predicted - y)

        self.weights -= self.lr * dw
        self.bias -= self.lr * db

        self.losses.append(self.loss(y_predicted, y))
```

Fig. 4. Fit Function

Overall, this class provides a simple implementation of logistic regression for binary classification tasks, including model initialization, training, and prediction.

```python
def predict(self, X):
    z_pred = np.dot(X, self.weights) + self.bias
    y_predicted = self._sigmoid(z_pred)
    y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
    return np.array(y_predicted_cls)
```

Fig. 5. Predict Function

2) Dataset Construction: To test the Logistic Regression, we need to create appropriate data to test various performance. We use regular normalized discrete point sets and assign them binary values:

```python
def create_LogisticRegression_data(num_points = 233):
    X1 = np.random.multivariate_normal([0, 0], [[1, .75],[.75, 1]], num_points)
    X2 = np.random.multivariate_normal([1, 4], [[1, .75],[.75, 1]], num_points)
    X = np.vstack((X1, X2)).astype(np.float32)
    y = np.hstack((np.zeros(num_points), np.ones(num_points)))
    return X, y
```

Fig. 6. Dataset Construction

3) Training, Prediction and Testing: The code begins by setting a random seed for reproducibility and generating random data for a Logistic Regression experiment. Key hyperparameters, such as the learning rate (`lr`) and the number of iterations (`num_iter`), are configured for the Logistic Regression model. The model is trained on the generated data, and predictions are made. Subsequently, essential performance metrics, including precision, recall, and F1 score, are calculated using appropriate functions. Finally, the results, comprising precision, recall, and F1 score, are displayed in the console output for evaluation.

4) Results Visualization: A complex set of codes was used to visualize the results to facilitate analysis and iterative optimization.

```python
LogisticRegression_key = True

# Choose Your Seed!
seed_now = random.randint(0, 23333)
seed_now = 12324
np.random.seed(seed_now)
print(f'[+] Random seed of LR: {seed_now}')

if LogisticRegression_key:
    # Create Data
    X, y = create_LogisticRegression_data()

    # Set Hyper-Parameters
    lr = 0.3
    num_iter = 1600
    model = LogisticRegression(lr, num_iter)
    model.fit(X, y)
    predictions = model.predict(X)

    # Calculate Precision, Recall, F1
    precision = precision_score(y, predictions)
    recall = recall_score(y, predictions)
    f1 = f1_score(y, predictions)

    print("=============================================")
    print("+     The result of Logistic Regression     +")
    print("=============================================")
...
```

Fig. 7. Training of Logistic Regression

```python
# Plot Decision Boundary
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    xx, yy, Z, x_values, y_values = plot_decision_boundary_log2d(X, mode
    plt.contourf(xx, yy, Z, alpha=0.8, cmap='Set3')
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='seagreen', cmap='Set3
    plt.plot(x_values, y_values, label='Decision Boundary', color='mediu
    plt.title('Logistic Regression Test')
    plt.legend()

# Plot Loss Curve
    plt.subplot(1, 2, 2)
    plt.plot(range(len(model.losses)), model.losses, label='Loss Curve',
    plt.title(f'Loss Curve of lr={lr}, epoch={num_iter}')
    plt.legend()
    plt.savefig(f'./output/LogisticRegression_Seed_{seed_now}_lr_{lr}_ep
    plt.show()
```

Fig. 8. Results Visualization

C. Output Analysis

1) Output Image and Loss Curve: The code aims to visualize the results of the Logistic Regression model. It generates two subplots. The first subplot illustrates the decision boundary of the model by contouring the classification regions. Data points are displayed as scatter points, with different colors representing their respective classes. The decision boundary is plotted as a distinct line. The second subplot depicts the loss curve, showcasing how the loss function evolves over training iterations (epochs). Both visualizations provide insights into the model's performance and training progress.

D. Model Optimization

1) Activation Function Selection: Activation Function Selection Analysis: The choice of the sigmoid activation function in our Logistic Regression (LR) model is a critical aspect

TABLE I
THE RESULT OF LOGISTIC REGRESSION

Precision	Recall	F1 Score	Loss
0.9915	1.0000	0.9957	0.0180

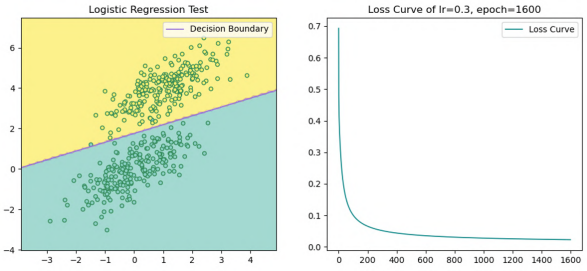

Fig. 9. Output Image and Loss Curve

of our design. Here, we elaborate on the advantages and drawbacks of using the sigmoid function:

- Advantages of Sigmoid Function:
  - Smoothness: The sigmoid function is smooth and differentiable, making it suitable for gradient-based optimization algorithms like gradient descent. This property enables efficient weight updates during training.
  - Normalization: The sigmoid function outputs values in the range of (0, 1), effectively normalizing the predictions as probabilities. This is particularly useful for binary classification tasks.
- Drawback of Sigmoid Function:
  - Vanishing Gradient: Sigmoid can suffer from the vanishing gradient problem, especially in deep neural networks with many layers. This can slow down or hinder the convergence of the model during training.

The selection of the sigmoid function in our model is motivated by its suitability for binary classification. However, it's essential to be aware of its limitations, particularly in the context of deep neural networks, where alternatives like ReLU and its variants are often preferred.

2) Hyperparameters Selection:

- Learning Rate (lr):
  - Analysis: A learning rate was chosen because it provides a good balance between fast convergence and stability. It allows the model to make reasonably large updates to the parameters in each iteration without diverging. In this simple test, we tried dynamically processing the number of iterations, and the diff was small.
- Number of Iterations (epoch):
  - Analysis: The choice of 1000 iterations was made based on experimentation. It was found that this number of iterations was sufficient for the model to converge and achieve good performance on the dataset without overfitting. Further increasing the number of iterations did not significantly improve results.

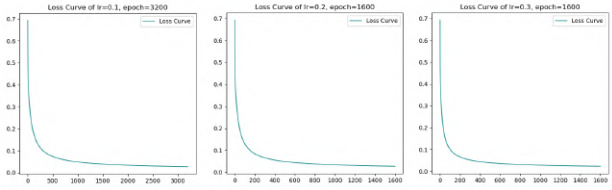

Fig. 10. Loss Comparison for lr Chose

3) Tolerance: The tolerance parameter, denoted as ϵ, is a crucial hyperparameter in the training process of machine learning models, including Logistic Regression and Multi-Layer Perceptron (MLP). It plays a significant role in determining when the training process should terminate and directly affects the training time, stability and convergence behavior.

The tolerance parameter represents the acceptable change in the loss function between two consecutive iterations, below which the training process is considered to have converged. In other words, if the change in the loss function falls below ϵ, the training stops, as the model's parameters are not changing significantly.

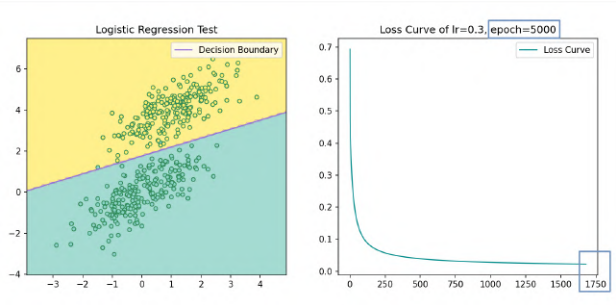

Fig. 11. Tolerance

As shown in the figure, when the number of iterations is set to 5000, due to the low loss reduction rate, the training is terminated early(), effectively saving resources.

```
[+] Random seed of LR: 12324
============================================
+     The result of Logistic Regression     +
============================================
[Precision]     [recall]        [f1_score]
0.9915          1.0             0.9957
```

Fig. 12. Test Output

IV. THE IMPLEMENTATION OF MLP MODEL

A. Multilayer Perceptron Principles

The Multilayer Perceptron (MLP) is a fundamental type of neural network that serves as the basis for more complex architectures. It simulates the human brain's functionality by

constructing artificial "neurons" with activation functions and connecting them in a structured manner.

In an MLP, neurons are organized into multiple layers:

- **Input Layer**: This is the first layer, which receives the raw input data. Each neuron corresponds to a feature or input dimension.
- **Hidden Layers**: Intermediate layers between the input and output layers are referred to as hidden layers. These layers extract and transform features from the input data.
- **Output Layer**: The final layer, known as the output layer, produces the network's predictions or classifications.

Neurons within a layer are fully connected to all neurons in the subsequent layer, creating a dense network of connections.

For binary classification problems, the cross-entropy error is commonly used to measure the model's performance:

$$E = -\sum_{n=1}^{N} t^{(n)} \log(o^{(n)}) + (1 - t^{(n)}) \log(1 - o^{(n)})$$

$$o^{(n)} = (1 + \exp(-z^{(n)}))^{-1}$$

Training an MLP involves updating numerous parameters, including the weights and biases. To accelerate training, the backpropagation algorithm is employed. Backpropagation efficiently computes gradients using the chain rule of calculus. The specific process is as follows:

1) Activation Functions: The choice of activation function is crucial in neural networks, as it determines the network's ability to learn complex patterns. Two commonly used activation functions are:

- **Sigmoid Function**: Defined as $\sigma(z) = \frac{1}{1+e^{-z}}$, the sigmoid function is often used in the output layer for binary classification tasks.
- **ReLU Function**: Defined as $ReLU(z) = \max(0, z)$, the Rectified Linear Unit (ReLU) function is preferred in hidden layers due to its computational efficiency and its ability to mitigate the vanishing gradient problem.

2) Forward Propagation: Forward propagation involves the following steps: 1. Input data is fed into the input layer. 2. Data flows through hidden layers, where neurons apply activation functions. 3. Output layer produces predictions or classifications.

3) Backward Propagation: Backpropagation is a fundamental algorithm for training neural networks. It efficiently computes gradients of the loss function with respect to each weight by propagating errors backward through the network. This process is accomplished using the chain rule of calculus, making it a cornerstone of neural network training.

The batch gradient descent method is as follows:

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta \sum_{n=1}^{N} \frac{\partial E(o^{(n)}, t^{(n)}; w)}{\partial w_{ki}}$$

These processes repeat iteratively during training to optimize the MLP for accurate predictions.

B. Code Implementation

1) Overview: In this section, we will provide an overview of the code implementation of MLP. The code structure diagram of the class implementation is as follows:

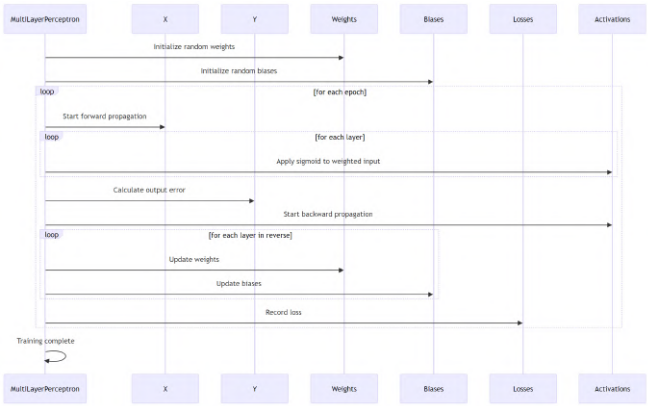

Fig. 13. MLP Class

a) The constructor `__init__` initializes an instance of the `MLP` class with the following parameters:

- `self.layer_sizes`: A list specifying the sizes of the neural network's layers, including the input and output layers.
- `self.weights`: An empty list that will store weight matrices for connections between layers during model initialization.
- `self.biases`: An empty list that will store bias vectors for each layer during model initialization.
- `self.losses`: An empty list used to record the loss values during training.
- `self.num3loss`: A variable that will be used to store the number of iterations when the loss drops to the initial 30%.
- `self.hasNum3loss`: A boolean variable initially set to 'False' indicating whether 'num3loss' is none.

```python
def __init__(self, layer_sizes):
    self.layer_sizes = layer_sizes
    self.weights = []
    self.biases = []
    self.losses = []
    self.num3loss = None
    self.hasNum3loss = False

    for i in range(len(layer_sizes) - 1):
        self.weights.append(np.random.randn(layer_sizes[i], \
            layer_sizes[i+1]))
        self.biases.append(np.random.randn(layer_sizes[i+1]))
```

Fig. 14. Init Function

b) Sigmoid Function

The `_sigmoid` method implements the sigmoid activation function, defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

This function maps the input z to a value in the range $[0, 1]$ and is used to model the probability of the positive class.

c) Loss Function

The `loss` method calculates the binary cross-entropy loss between predicted values y_{pred} and true labels y. It clips y_{pred} to ensure stability, computes the loss using the binary cross-entropy formula, and returns the mean of the calculated loss.

```python
def loss(self, y_pred, y):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)  # avoid log(0)
    return -np.mean(y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
```

Fig. 15. Loss Method

d) Model Training

The `fit` function in the MLP class is responsible for training the neural network using the given input data X and target labels y. It iteratively updates the network's weights and biases to minimize the loss function. In each iteration:

It computes the output values `y_predicted` using the sigmoid function for each neuron. Then, it calculates the gradients `dw` and `db` of the loss function with respect to the network's parameters.

Afterwards, it updates the weights and biases using gradient descent with a learning rate of `self.lr` to adjust the network's parameters towards a more optimal configuration. Throughout the training process, it records the loss values to monitor the convergence and learning progress.

```python
def _forward(self, X):
    output = X
    for weight, bias in zip(self.weights, self.biases):
        output = self._sigmoid(np.dot(output, weight) + bias)
    return output

def fit(self, X, y, epochs, learning_rate):
    for _ in range(epochs):
        activations = [X]
        for weight, bias in zip(self.weights, self.biases):
            activations.append(self._sigmoid(np.dot(activations[-1], weight) + bias))

        y = y.reshape(y.shape[0], 1)
        error = activations[-1] - y
        deltas = [error * activations[-1] * (1 - activations[-1])]  # sigmoid

        for i in range(len(self.weights) - 1, 0, -1):
            delta = np.dot(deltas[-1], self.weights[i].T) * activations[i] * \
                    (1 - activations[i])  # sigmoid
            deltas.append(delta)

        deltas.reverse()

        for i in range(len(self.weights)):
            self.weights[i] -= learning_rate * np.dot(activations[i].T, deltas[i])
            self.biases[i] -= learning_rate * np.sum(deltas[i], axis=0)

        self.losses.append(self._loss(activations[-1], y))
```

Fig. 16. Fit Function

e) Prediction

The `predict` method in the MLP class computes predictions for new data points represented by the feature matrix X. It follows these steps:

1) Compute the weighted sum of input features and biases: $z\_pred = X \cdot \text{weights} + \text{bias}$.
2) Apply the sigmoid activation function to z_pred to obtain predicted probabilities: $y\_predicted = \sigma(z\_pred)$.
3) Threshold the predicted probabilities at 0.5 to classify data points into binary classes (0 or 1).

This method offers a straightforward way to perform binary classification predictions using the trained MLP model, assigning data points to classes based on predicted probabilities.

```python
def predict(self, X):
        output =  self._forward(X)
        output_cls = [1 if i > 0.5 else 0 for i in output]
        return np.array(output_cls)
```

Fig. 17. Predict Function

2) Dataset Construction: To test the Logistic Regression, we need to create appropriate data to test various performance. We use regular normalized discrete point sets and assign them binary values:

```python
def create_MLP_data(num_points, x_range, y_range):
    X = np.random.rand(num_points, 2)
    X[:, 0] = X[:, 0] * (x_range[1] - x_range[0]) + x_range[0]
    X[:, 1] = X[:, 1] * (y_range[1] - y_range[0]) + y_range[0]

    y = np.random.randint(0, 2, num_points)
    return X, y
```

Fig. 18. Dataset Construction

3) Training, Prediction and Testing: The implementation of training is similar to logistic regression testing and will not be described again. Here's how to generate the desired model structure:

4) Results Visualization: A complex set of codes was used to visualize the results to facilitate analysis and iterative optimization. For the generation method of contour grid, please refer to the attached code source file.

C. Output Analysis

Our MLP model implements arbitrary settings of the model hierarchy. The output sample of a test sample is as follows:

The part in the box in the figure is the hierarchical structure of the neural network that can be adjusted arbitrarily.

In addition to this, iteratively adjust the learning rate, the number of iterations, and try new seeds to explore problems that different structures may hide.

1) Width Test: `[+] lr=0.025, epoch=250000`

TABLE II
THE RESULT OF LOGISTIC REGRESSION

Precision	Recall	F1 Score	Final Loss
0.7000	0.5224	0.5983	0.6342
0.8200	0.6119	0.7009	0.7484
0.8947	0.7612	0.8226	0.5845
0.9365	0.8806	0.9077	0.6618

where, `neuron=[3, 6, 20, 50]`

```python
print(">> Width Test")

    # Set Hyper-Parameters
    lr = 0.02
    num_iter = 300000
    print(f'[+] lr={lr}, epoch={num_iter}')

    neuron = [3, 6, 20, 50]
    print(f'[+] single layer: neuron={neuron}')

print(">> Depth Test")

    # Set Hyper-Parameters
    lr = 0.015
    num_iter = 160000
    print(f'[+] lr={lr}, epoch={num_iter}')

    layer = [2, 3, 6, 10]
    unit_of_layer = 20
    layer_frame = []
    for i in range(len(layer)):
        if not Diff_key:
            # each layer's units num is the same, to better compare
            layer_frame.append([2] + [unit_of_layer] * layer[i] + [1])
        else:
            # different units of each layer, to test the model
            layer_frame.append([2])
            for j in range(layer[i]):
                layer_frame[i].append(unit_of_layer - j)
            layer_frame[i].append(1)
        print(f'[+] layer: {layer[i]} \tframe={layer_frame[i]}')
```

Fig. 19. Generate Model Structure

```
==========================================
+            The result of MLP           +
==========================================
>> Width Test
[+] lr=0.025, epoch=200000
[+] single layer: neuron=[3, 6, 20, 50]
[Precision]     [recall]        [f1_score]      [loss0.3]       [final_loss]
0.7             0.5224          0.5983          None            0.6352459924768561
0.8163          0.597           0.6897          None            0.6887690079592104
0.8364          0.6866          0.7541          None            0.4966119853779868
0.9153          0.806           0.8571          901             0.5577339063988684
>> Depth Test
[+] lr=0.015, epoch=160000
[+] layer: 2    frame=[2, 20, 20, 1]
[+] layer: 3    frame=[2, 20, 20, 20, 1]
[+] layer: 6    frame=[2, 20, 20, 20, 20, 20, 20, 1]
[+] layer: 10   frame=[2, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 1]
[Precision]     [recall]        [f1_score]      [loss0.3]       [final_loss]
0.9853          1.0             0.9926          90792           0.056361886864611896
1.0             1.0             1.0             39705           0.017224414155406768
1.0             1.0             1.0             13732           0.0028352035185489 49
1.0             1.0             1.0             901             0.0011048621686611834
```

Fig. 20. Output Sample

2) Depth Test: The example in the picture is `[+] lr=0.01, epoch=160000`

TABLE III
DEPTH TEST RESULTS

Precision	Recall	F1 Score	Final Loss
0.9552	0.9552	0.9552	0.0346
1.0000	0.9701	0.9848	0.0127
1.0000	0.9851	0.9925	0.0043
1.0000	1.0000	1.0000	5.11e-06

where, `layer = [2, 3, 6, 10]`

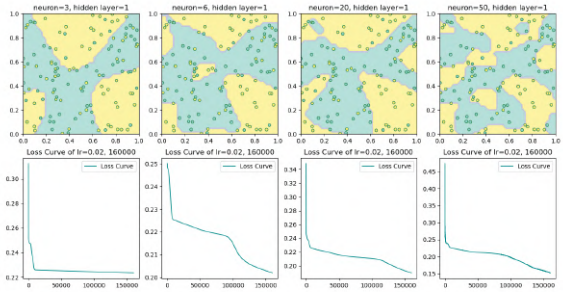

Fig. 21. Width Test Output

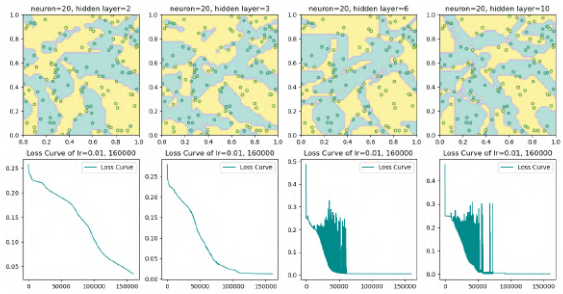

Fig. 22. Depth Test Output

D. Model Optimization

1) Hyperparameters Analysis: In order to better analyze and test the impact of model construction on the results, we selected a more appropriate random number seed (there will not be too much overlap between random points) and fixed the seed to maintain the unity of irrelevant variables.

In addition, in order to test the impact of the number of layers of the deep neural network, the number of neurons in different layers is tentatively set to be the same when compared.

- Learning Rate:
  - Attempts: Select a learning rate of 0.015 for the single-layer model, and 0.01 for the multiple-layer model.
  - Analysis: A learning rate of 0.01 was chosen because it provides a good balance between fast convergence and stability. It allows the model to make reasonably large updates to the parameters in each iteration without diverging.
  - We record the number of iterations when the loss drops to 30% of the initial value in the model, and observe the loss curve to choose a better learning rate. As shown in the figure:
    In the width test as follow, the loss does not decrease but increases when lr=0.3. After continuous testing, gradually reduce the learning rate to prevent the model overfitting.

- Tolerance:
  - In the training of MLP, according to the actual situation, we found that the loss may fluctuate greatly during the training process. Some processing attempts (such as using regularization methods, etc.) cannot reduce this phenomenon well.
  - However, when the hyperparameters are appropriate, loss will eventually drop to a value close to 0. Therefore, we removed the restriction on loss tolerance to better monitor the model effect.

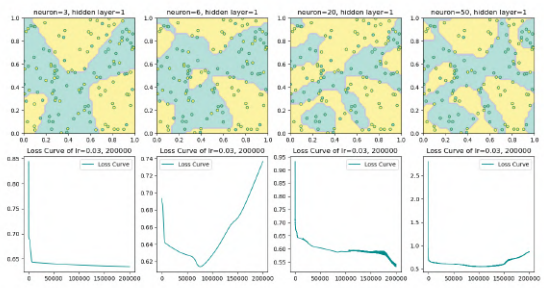

Fig. 23. Loss Curve Rising Example

2) Speculative Analysis of Width and Depth in Neural Networks: Neural networks' architecture, specifically their width (number of neurons in each layer) and depth (number of layers), significantly influences their learning capacity and performance. Here we analyze how these dimensions affect the network:

- **Representational Power:** A wider network can learn a greater variety of linear feature combinations, while a deeper network can capture more complex hierarchical structures and abstractions. Increasing the width may prove more effective, especially when the task requires capturing a large number of low-level features.
- **Vanishing/Exploding Gradients:** Deep neural networks may encounter vanishing or exploding gradients, impacting their training efficiency and ultimate performance. In contrast, wider, shallower networks are less likely to face these issues.
- **Optimization Difficulty:** Optimizing deep networks is generally more complex and challenging than shallower ones. The interaction between layers in deep networks can result in more complex loss surfaces, making it harder to find good local minima.
- **Overfitting Risk:** As the width of the network increases, so does the model's capacity, which can lead to overfitting, particularly when the number of training samples is limited.

Although deep neural networks appear to be superior. Sometimes, a wider, shallower network may be easier to train and provide similar or better performance than a deeper one.

3) Another Speculative Analysis: As the width of the neural network increases, the probability of the loss surface reaching a local optimum decreases. There is a novel perspective to look at this problem:

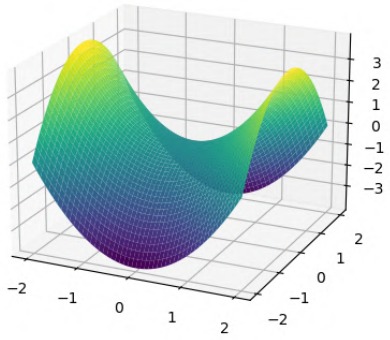

Fig. 24. Saddle Surface in 2-D

As shown in the figure, when the dimensionality increases, the probability that the loss function reaches the local optimal value in all dimensions simultaneously decreases. That is, there is a greater probability that it is not local optimal in a certain dimension, forming an elliptical parabola. Instead, it forms a high-dimensional saddle surface.

V. Conclusion and Future Problems

A. Conclusion

In this research, we explored the development and evaluation of binary classification models, focusing on two distinct approaches: Logistic Regression and Multi-Layer Perceptron (MLP) neural networks. We investigated the principles, code implementation, and optimization of both models to gain insights into their performance and behavior.

Logistic Regression, with its simplicity and interpretability, serves as a foundational binary classification technique. We discussed its principles, including the sigmoid activation function and loss function, and provided a detailed overview of its code implementation. Through experiments, we observed the impact of hyperparameters on the model's training process and final performance.

The MLP neural network, on the other hand, represents a more common approach. We delved into its principles, covering aspects such as forward pass and backward propagation. The code implementation of MLP showcased its adaptability, allowing for the customization of model architecture, including the number of layers and neurons in each layer. We conducted experiments to explore the effects of varying model width and depth on performance.

Results indicate that both models can achieve high precision, recall, and F1 scores when appropriately configured. The choice between Logistic Regression and MLP depends on the specific problem and dataset characteristics. Logistic

Regression excels in simplicity and efficiency. In contrast, MLP offers greater flexibility and can better deal with complex data.

Additionally, our experiments have displayed the importance of hyperparameter tuning, including the learning rate, tolerance settings, and model architecture, in achieving optimal model performance. These findings underscore the significance of careful experimentation and parameter selection in machine learning model development.

In conclusion, this research provides a comprehensive understanding of Logistic Regression and MLP models for binary classification.

B. Future Problems

As our study delved into the development and evaluation of binary classification models, there are several promising directions for future research and exploration in this domain:

1) Other Activation Functions: Future research can investigate the use of advanced activation functions like ReLU, Leaky ReLU, and Parametric ReLU (PReLU). These functions may offer improved convergence properties and mitigate issues like the vanishing gradient problem.

2) Ensemble Learning: Exploring ensemble learning methods, such as Random Forests, Gradient Boosting, and AdaBoost, in the context of binary classification can lead to improved model accuracy and stability. Future research can evaluate the effectiveness of ensemble techniques compared to single-model approaches.

These future research directions hold the potential to contribute to the development of more robust and accurate models for a wide range of applications.

REFERENCES

[1] Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer.