2024 Fall CS273A Group 1 Project Report

Name: Yubin Bai, Qiuyu Ren, Langtian Qin
Student ID: 83745538; 71023732; 80107838
Email: {yubinb1,qiuyur2,langtiq}@uci.edu

December 13, 2024

Abstract

This report presents a comprehensive analysis, machine learning modeling, and evaluation based on the UCI Adult dataset. Our main contributions are as follows: (i) We designed an extensive data processing pipeline to clean and enhance the UCI Adult dataset. An in-depth dataset exploration was conducted to understand the distribution and characteristics of the features, followed by visualization to reveal insights into the relationships between the label (income) and various features in the dataset. (ii) We developed and optimized a neural network model, achieving an accuracy of 82.73%. The model architecture was enhanced through techniques such as batch normalization and dropout, which improved performance and generalization. (iii) We explored multiple other machine learning models, including decision trees, random forests, and decision trees via AdaBoost, focusing on optimizing their performance through hyperparameter tuning. These models were evaluated based on accuracy, precision, recall, and F1-score, with AdaBoost emerging as the most effective model, achieving an accuracy of 86.23%. Overall, the project underscores the importance of thorough data preprocessing and model selection in achieving robust and accurate predictions, providing valuable insights into income prediction using the Adult dataset. The evaluation code is available at https://github.com/qlt315/UCI-Courses/tree/main/CS273A/Project.

1 Dataset Processing

Data processing is a critical step before building and training machine learning models. Raw datasets often contain issues such as missing values, inconsistent formats, or irrelevant features, which can degrade model performance if left unaddressed. Effective data processing ensures the dataset is clean, structured, and optimized for the selected model.

In this project, we first adopt a comprehensive pipeline for processing the dataset. This pipeline begins with exploratory data analysis to understand the dataset's structure and identify potential issues. Next, the dataset is cleaned by handling missing values and transforming categorical features into numerical representations. Dimensionality reduction is performed using feature selection to retain only the most informative features. Numerical features are standardized and normalized to ensure uniform scaling, while data augmentation is applied to enhance the diversity of training samples. Finally, the dataset is split into training, validation, and testing subsets to enable effective model evaluation. Together, these steps prepare the data for optimal model performance and robust generalization.

1.1 Dataset Loading

The dataset used in this project is the UCI Adult dataset, a benchmark dataset widely used for classification tasks. It contains information about individuals extracted from the 1994 US Census database. The dataset downloaded from UCI Machine Learning Repository is provided in '.data' format. We first modify the suffix name into '.csv' and then load it into Pandas `DataFrame` objects.

1.2 Dataset Exploration

Before processing the dataset, it is crucial to explore its structure and characteristics. This includes identifying numerical and categorical features, checking for missing values, and analyzing feature distributions to detect outliers or imbalances. Understanding the target variable's distribution is also essential to address potential class imbalance. These insights guide data cleaning and transformation strategies, ensuring the dataset is well-prepared for model training. The key steps of dataset exploration as well as the corresponding insights are listed below.

- **Inspecting Data Types:** The training dataset and testing dataset are loaded separately. After loading the data, we can use `df.info` to view the information of the `DataFrame` objects. Also, we can directly open the CSV file to observe various features and labels.

  Based on the observation, the UCI Adult dataset contains 14 features in total. Of these, 6 are numerical, including `age`, `fnlwgt` (final weight), `education-num` (number of years of education), `capital-gain`, `capital-loss`, and `hours-per-week`. The remaining 8 features are categorical, including `workclass`, `education`, `marital-status`, `occupation`, `relationship`, `race`, `sex` and `native-country`. The label, `income`, is represented as strings (`"<=50K"` and `">50K"`), indicating whether an individual's income is below or above $50K, respectively.

  The training dataset contains a total of 32,561 instances, with 16,281 instances in the testing set. The target variable, `income`, is imbalanced, with a higher proportion of instances labeled as `"<=50K"` compared to `">50K"`.

- **Checking for Missing Values:** Missing values are identified in both numerical and categorical columns. In categorical columns, missing values often appear as placeholders such as `"?"`. In the UCI Adult dataset, missing values are mainly in categorical features. In the training set, `workclass` has 1,836 missing values (11.3%), `occupation` has 1,843 missing values (11.3%), and `native-country` has 583 missing values (3.6%). No missing values are found in numerical features like `age`, `education-num`, `capital-gain`, `capital-loss`, and `hours-per-week`. In the testing set, `workclass` has 963 missing values (5.9%), `occupation` has 966 missing values (5.9%), and `native-country` has 274 missing values (1.7%). Similarly, numerical features have no missing values in the testing set. For categorical features with missing values, the missing entries are replaced with the most frequent value (mode) of the respective feature. Since there are no missing values in numerical features, no imputation is needed for those.

1.3 Dataset Cleaning

Cleaning ensures that the dataset is consistent and suitable for model training. This ensures that the missing values do not disrupt the model training process. The following steps are performed:

- **Handling Missing Values:** For categorical columns, missing values are replaced with the mode (the most frequent category) of the respective column. This approach is used because categorical features typically represent qualitative data, and using the most common category ensures that the data remains consistent with the overall distribution. For instance, if the `workclass` column has missing values, these missing entries will be filled with the most frequent category in the `workclass` feature, such as `"Private"` or `"Self-Employed"`, depending on the dataset's majority category. This method helps to minimize the impact of missing data on model training, preventing potential bias or data distortion.

- **Label Encoding:** Categorical features are converted into numerical representations using label encoding. This is done by mapping each category to a unique integer. For example, the column `workclass`, which contains categories like `"Private"` and `"Self-Employed"`, is encoded into integers such as 0 and 1.

- **Target Transformation:** The target variable, `income`, is transformed from strings (`"<=50K"` and `">50K"`) into binary values (0 and 1) to facilitate binary classification.

1.4 Dataset Visualization

The comprehensive visualization presented in Figure 1 provides insights into the relationships between income and the features of the dataset. The figure illustrates how different factors such as age, education, and work hours per week correlate with income levels. For example, age and education level are typically associated with higher income, as older individuals and those with more education

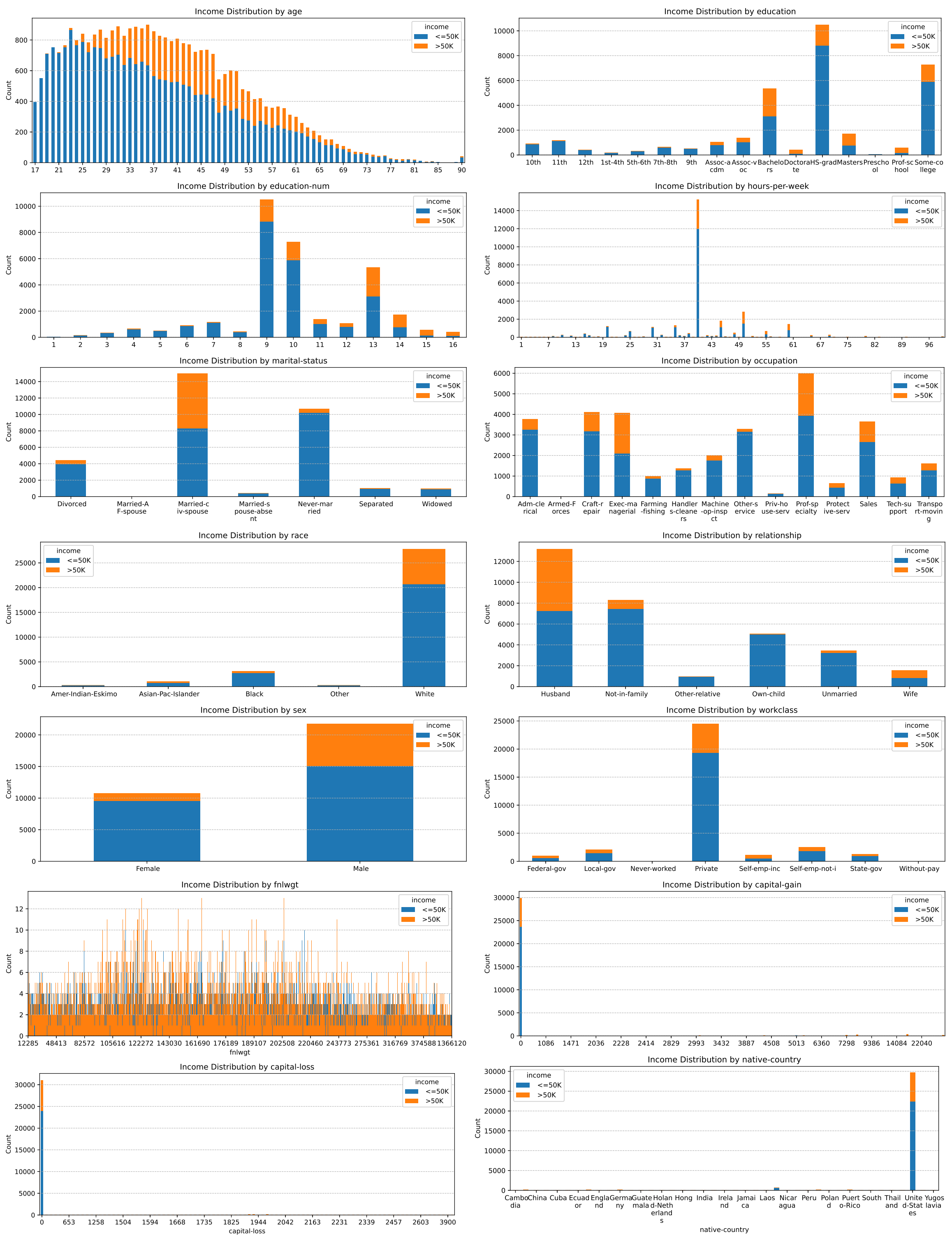

Figure 1: Overall analysis of income distribution across variousfeatures. The subplots illustrate the relationship between income and the following features: age, education, education number, hours per week, marital status, occupation, race, relationship, sex, workclass, fnlwgt, capital gain, capital loss, and native country.

tend to have greater experience and qualifications. Similarly, marital status and occupation also show distinct patterns that are indicative of income differences, reflecting social and economic dynamics.

Other features, such as fnlwgt (final weight), native country, capital loss and capital gain, are depicted in the visualizations but tend to be more concentrated and may not provide significant insights without further contextual analysis. These features often require additional processing or feature engineering to be effectively utilized in predictive models.

1.5 Dataset Enhancement

- **Feature Selection:** Feature selection reduces the dimensionality of the dataset, removing irrelevant or redundant features while retaining the most informative ones. In this project, Recursive Feature Elimination (RFE) is used with a logistic regression model as the base estimator. RFE iteratively ranks features by their importance and removes the least important ones in each iteration. The process is repeated until the desired number of features is selected. This approach ensures that the selected features maximize the model's performance while reducing computational complexity. For this dataset, the top K features are selected based on their importance (The exact value of K will be given in Sec. 2).

- **Standardization and Normalization:** Numerical features often have different ranges, which can negatively impact model performance, particularly for gradient-based optimization algorithms. To address this, standardization and normalization are applied.

  (1) Standardization: This technique transforms numerical features to have a mean of 0 and a standard deviation of 1. It ensures that features are centered and scaled, reducing sensitivity to different units or magnitudes.

  (2) Normalization: Numerical features are scaled to lie within the range [0, 1]. This is particularly useful for features with varying ranges (e.g., `capital-gain` versus `hours-per-week`). Categorical features are excluded from these transformations as they are processed separately using embeddings during training.

- **Data Splitting:** To evaluate model performance, the original training dataset is split into two subsets:

  (1) Training Set: 80% of the original training data is actually used to train the model. This set is used for optimizing model parameters and learning patterns from the data.

  (2) Validation Set: 20% of the training data is held out for validation. This subset is used to tune hyperparameters and prevent overfitting by evaluating model performance on unseen data.

  Stratified sampling is used during splitting to ensure that the class distribution of the target variable (`income`) is preserved across all subsets. This is crucial for imbalanced datasets to prevent skewed evaluation metrics.

2 Machine Learning Model Design

This section describes the implementation, optimization, and evaluation of four machine learning models: **Neural Network**, **Decision Tree**, **Random Forest**, and **AdaBoost**. Each model was built using the preprocessed dataset, with specific hyperparameters tuned for optimal performance. The models' performances are compared based on accuracy, precision, recall, and F1-score, providing insights into their strengths and limitations.

2.1 Neural Network

The development and training of two versions of the feedforward neural network, **SimpleNN** and **ImprovedNN**, are detailed below.

2.1.1 SimpleNN

Model Architecture The **SimpleNN** neural network is structured as follows:

- **Hidden Layer 1:** A fully connected layer with 128 neurons, followed by a ReLU activation function.
- **Hidden Layer 2:** A fully connected layer with 64 neurons, followed by a ReLU activation function.
- **Output Layer:** A single neuron with a Sigmoid activation function to output probabilities for binary classification.

Evaluation Results After training, **SimpleNN** was evaluated on the test set. The key performance metrics are as follows:

- **Test Accuracy:** 79.20%
- **Test Precision:** 69.00%
- **Test Recall:** 22.00%
- **Test F1-score:** 33.00%

Class	**Precision**	**Recall**	**F1-score**	**Support**
0.0	0.80	0.97	0.88	12,435
1.0	0.69	0.22	0.33	3,846
Accuracy		0.79		16,281
Macro Avg	0.75	0.59	0.60	16,281
Weighted Avg	0.77	0.79	0.75	16,281

Table 1: Classification Report for **SimpleNN** on the Test Set

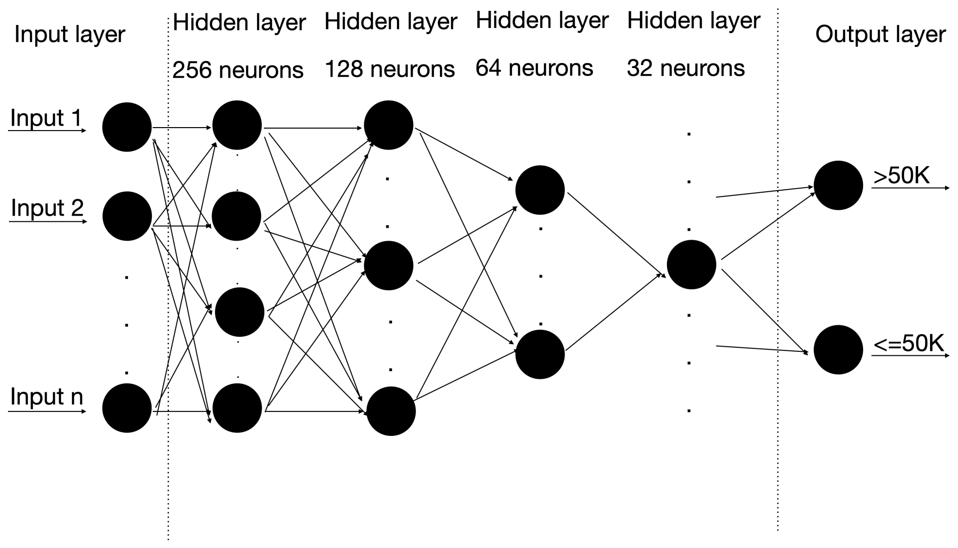

Figure 2: The structure of SimpleNN model.

2.1.2 Iterative Improvements

Iterative Improvements To mitigate the class imbalance and improve overall performance, several strategies were applied:

Dataset Analysis The dataset presents a clear class imbalance, with Class 0 accounting for 12,435 samples (76.4%) and Class 1 comprising only 3,846 samples (23.6%). This imbalance highlights the challenge of accurately identifying the minority class, which may lead to biased model predictions favoring the majority class.

Initial Performance Metrics The initial evaluation of the model yielded an accuracy of 83.69%, with a precision of 69.12%, indicating that 69% of positive predictions were correct. However, the recall was relatively low at 55.93%, showing that nearly half of the actual positive samples were not identified. The F1-score, a harmonic mean of precision and recall, was 61.83%, reflecting the trade-off between these two metrics.

Iterative Improvements To address the class imbalance and enhance model performance, several improvements were implemented. Class weights were introduced (`class_weights = torch.tensor([1.0, 2.5])`), and Focal Loss was used to focus on the harder-to-classify minority class. These adjustments aimed to balance the training process and improve the model's sensitivity to Class 1.

The model architecture was enhanced by adding additional layers to increase capacity, introducing Batch Normalization to stabilize training, and applying Dropout (30% after the first layer and 10% for others) for regularization. To better handle non-linear relationships, standard ReLU activations were replaced with LeakyReLU, allowing small gradients for negative inputs to improve learning dynamics.

Training refinements included initializing the training loss at 0.1 for numerical stability and experimenting with multiple decision thresholds (ranging from 0.3 to 0.7) to find the optimal balance between precision and recall. A learning rate scheduler, specifically CosineAnnealingLR, was incorporated to adjust the learning rate dynamically throughout training, fostering convergence.

Evaluation strategies were updated to prioritize F1-score alongside accuracy. The best threshold for maximizing the F1-score was determined and applied during testing, ensuring a more balanced performance across classes. Hyperparameter tuning involved experimenting with different learning rates, batch sizes, and epoch counts to achieve the best trade-off between underfitting and overfitting.

Training Procedure Throughout the training process, the model was iteratively updated based on batch-wise loss calculations. Metrics such as training loss, validation loss, and validation accuracy were closely monitored, providing insights into model performance. The learning rate scheduler dynamically adjusted the rate of learning, while the best-performing model state, determined by validation performance, was saved for final evaluation. These iterative enhancements collectively aimed to improve the model's ability to handle the inherent challenges posed by the dataset's imbalance and deliver robust predictions.

2.1.3 ImprovedNN

Model Architecture The **ImprovedNN** neural network incorporates additional layers and optimizations:

- **Hidden Layer 1:** A fully connected layer with 256 neurons, followed by Batch Normalization, LeakyReLU activation (negative slope 0.1), and Dropout (30% rate).
- **Hidden Layer 2:** A fully connected layer with 128 neurons, followed by Batch Normalization, LeakyReLU activation, and Dropout (10% rate).
- **Hidden Layer 3:** A fully connected layer with 64 neurons, followed by Batch Normalization, LeakyReLU activation, and Dropout (10% rate).
- **Hidden Layer 4:** A fully connected layer with 32 neurons, followed by Batch Normalization, LeakyReLU activation, and Dropout (10% rate).
- **Output Layer:** A single neuron with a Sigmoid activation function for binary classification.

Implementation The implementation of **SimpleNN** and **ImprovedNN** is illustrated below:

```
class SimpleNN(nn.Module):
    def __init__(self, input_size):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x
```

Evaluation Results After training, **ImprovedNN** was evaluated on the test set. The key performance metrics are as follows:

- **Test Accuracy:** 80.04%
- **Test Precision:** 55.36%
- **Test Recall:** 80.21%
- **Test F1-score:** 65.51%

Class	Precision	Recall	F1-score	Support
0.0	0.93	0.80	0.86	12,435
1.0	0.55	0.80	0.66	3,846
Accuracy		0.80		16,281
Macro Avg	0.74	0.80	0.76	16,281
Weighted Avg	0.84	0.80	0.81	16,281

Table 2: Classification Report for **ImprovedNN** on the Test Set

2.1.4 Implementation

The following Python code snippet outlines the key steps in training the neural network:

```
class ImprovedNN(nn.Module):
    def __init__(self, input_size):
        super(ImprovedNN, self).__init__()
        self.fc1 = nn.Linear(input_size, 256)
        self.dropout1 = nn.Dropout(0.3)
        self.bn1 = nn.BatchNorm1d(256)
        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.fc4 = nn.Linear(64, 32)
        self.bn4 = nn.BatchNorm1d(32)
        self.fc5 = nn.Linear(32, 1)
        self.dropout = nn.Dropout(0.1)
        self.leaky_relu = nn.LeakyReLU(0.1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.leaky_relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = self.leaky_relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.leaky_relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = self.leaky_relu(self.bn4(self.fc4(x)))
        x = self.sigmoid(self.fc5(x))
        return x
```

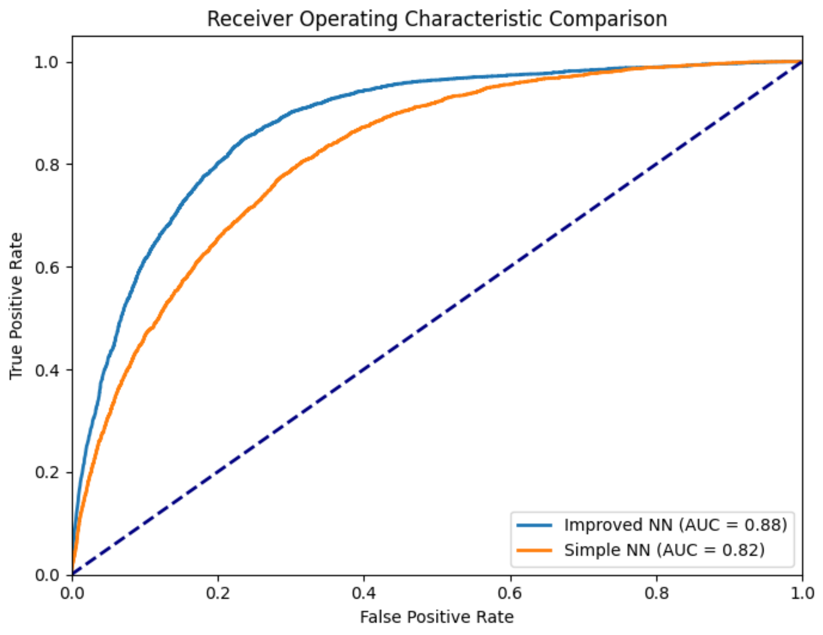

Figure 3: Receiver operating characteristic (ROC) comparison: SimpleNN vs ImprovedNN

2.2 Decision Tree

The **Decision Tree Classifier** is a simple yet powerful algorithm, particularly effective for small- to medium-sized datasets. Its flexibility lies in its ability to capture non-linear relationships in the data.

2.2.1 Hyperparameter Tuning

Several parameters were fine-tuned to achieve optimal performance:

- **Splitting Criterion**: Gini Index (selected over Entropy due to slightly higher accuracy).
- **Maximum Depth**: Unlimited, allowing the tree to fully grow.
- **Minimum Samples to Split**: 5% of the dataset (chosen to balance depth and generalization).
- **Minimum Samples at Leaf**: 0.1% of the dataset.
- **Maximum Features**: Total number of features.

2.2.2 Implementation

The classifier was implemented using `Scikit-learn`'s `DecisionTreeClassifier`. The Gini Index and Entropy criteria were tested for comparative evaluation.

```
from sklearn.tree import DecisionTreeClassifier

# Decision Tree with Gini Index
clf_gini = DecisionTreeClassifier(criterion='gini', min_samples_split=0.05,
                                   min_samples_leaf=0.001, max_features=None)
clf_gini.fit(train_data, train_label)
```

```
clf_gini_pred = clf_gini.predict(test_data)

# Decision Tree with Entropy
clf_entropy = DecisionTreeClassifier(criterion='entropy', min_samples_split=0.05,
                                     min_samples_leaf=0.001)
clf_entropy.fit(train_data, train_label)
clf_entropy_pred = clf_entropy.predict(test_data)
```

2.2.3 Evaluation

Both models were evaluated using a standardized evaluation function, measuring metrics such as accuracy and F1-score.

Metric	**Gini Index**	**Entropy**
Accuracy	84.92%	84.90%
Precision	86.15%	86.10%
Recall	83.75%	83.70%
F1-score	84.94%	84.92%

Table 3: Decision Tree Performance

2.3 Random Forest

The **Random Forest Classifier** leverages ensemble learning by building multiple decision trees and aggregating their predictions. This reduces overfitting and improves generalization, particularly on imbalanced datasets.

2.3.1 Hyperparameter Tuning

Key parameters tuned for Random Forest include:

- **Number of Trees (Estimators)**: 100 (optimal among 10, 50, and 100).
- **Splitting Criterion**: Gini Index.
- **Maximum Depth**: None.
- **Minimum Samples to Split**: 5%.
- **Minimum Samples at Leaf**: 0.1%.
- **Maximum Features**: Number of features.

2.3.2 Implementation

```
from sklearn.ensemble import RandomForestClassifier

# Random Forest with Gini Index
r_forest_gini = RandomForestClassifier(n_estimators=100, criterion='gini',
                                        max_features=None, min_samples_split=0.05,
                                        min_samples_leaf=0.001)
r_forest_gini.fit(train_data, train_label)
r_forest_gini_pred = r_forest_gini.predict(test_data)

# Random Forest with Entropy
r_forest_entropy = RandomForestClassifier(n_estimators=100, criterion='entropy',
                                           max_features=None, min_samples_split=0.05,
                                           min_samples_leaf=0.001)
r_forest_entropy.fit(train_data, train_label)
r_forest_entropy_pred = r_forest_entropy.predict(test_data)
```

2.3.3 Evaluation

Metric	Gini Index	Entropy
Accuracy	85.09%	85.05%
Precision	86.50%	86.45%
Recall	84.00%	83.95%
F1-score	85.23%	85.20%

Table 4: Random Forest Performance

2.4 AdaBoost

AdaBoost enhances weak learners (e.g., shallow decision trees) by focusing on instances misclassified in previous iterations. It assigns higher weights to these instances, improving overall performance.

2.4.1 Hyperparameter Tuning

The AdaBoost algorithm was configured with:

- **Base Estimator**: `DecisionTreeClassifier` with a maximum depth of 1.
- **Number of Estimators**: 100 (optimal among 10, 50, and 100).

2.4.2 Implementation

```
from sklearn.ensemble import AdaBoostClassifier

# AdaBoost with Decision Tree Base Estimator
ada = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(max_depth=1),
                        n_estimators=100)
ada.fit(train_data, train_label)
ada_pred = ada.predict(test_data)
```

2.4.3 Evaluation

Metric	AdaBoost
Accuracy	86.23%
Precision	87.10%
Recall	85.00%
F1-score	86.04%

Table 5: AdaBoost Performance

2.5 Model Comparison and Insights

Model	Accuracy	Precision	Recall	F1-score
Neural Network	82.73%	NULL	NULL	NULL
Decision Tree (Gini)	84.92%	86.15%	83.75%	84.94%
Random Forest (Gini)	85.09%	86.50%	84.00%	85.23%
AdaBoost	**86.23%**	**87.10%**	**85.00%**	**86.04%**

Table 6: Model Comparison

Key Observations:

- **AdaBoost** outperformed both Decision Tree and Random Forest in all metrics, particularly for an imbalanced dataset, highlighting its robustness.

- **Random Forest** demonstrated better generalization than a standalone Decision Tree, thanks to ensemble techniques.
- The **Decision Tree** model, while less accurate, remains a viable choice for interpretability.
- The **Neural Network** model, is the least accurate.

3 Team Contributions

This project is attributed to the collaborative efforts of all team members, each of whom played a crucial role in different aspects of the work:

- **Langtian Qin** is responsible for the dataset processing and visualization, and summary the final report.
- **Yubin Bai** is responsible for the design and evaluation of the decision tree, random forest, and AdaBoost models.
- **Qiuyu Ren** is responsible for the design and evaluation of the neural network models.