Machine Learning Project - 1

Team: Nangara

Ullas G
(IM22022125)
Mallikarjun Chakoti
(IMT2022116)
Adithya Nangarath
(IMT2022024)

Problem Statement: Lend or Lose - Loan Default Prediction

Dataset Overview

This dataset is used to predict loan default risk based on various borrower characteristics and loan details. It contains 255,347 rows and 18 columns, each representing a specific feature of the borrower or the loan. Below is a description of each feature:

- **LoanID**: Unique identifier for each loan.
- **Age**: Age of the borrower.
- **Income**: Annual income of the borrower.
- **LoanAmount**: Amount of money being borrowed.
- **CreditScore**: Credit score of the borrower, indicating their creditworthiness.
- **MonthsEmployed**: Number of months the borrower has been employed.
- **NumCreditLines**: Number of credit lines the borrower has open.
- **InterestRate**: Interest rate for the loan.
- **LoanTerm**: Loan term length in months.
- **DTIRatio**: Debt-to-Income ratio, comparing debt to income.
- **Education**: Highest level of education attained by the borrower.
- **EmploymentType**: Employment status (Full-time, Part-time, etc.).
- **MaritalStatus**: Marital status of the borrower.
- **HasMortgage**: Whether the borrower has a mortgage (Yes/No).
- **HasDependents**: Whether the borrower has dependents (Yes/No).
- **LoanPurpose**: Purpose of the loan (e.g., Auto, Business, Education).
- **HasCoSigner**: Whether the loan has a co-signer (Yes/No).
- **Default**: Target variable indicating whether the loan defaulted (1) or not (0).

Preprocessing

For the pre-processing and analysis of the dataset, we used the **pandas**, **matplotlib**, and **seaborn** libraries in Python.

1. Checking Null Values

We used the **isna()** function from the **pandas** library to check for null values in each column. We found that there were no null values in any of the columns in both the train and test datasets. Hence, there was no need to impute any values in the data frame.

2. Looking for Dirty/Messy Data

To check for dirty or messy data, we manually inspected the dataset and used **value_counts()** for each column to identify any inconsistencies, such as spelling errors in categorical variables. We found that the data was clean, with no issues in any of the columns.

3. Outliers

For all the numerical columns in the dataset (`Age`, **Income**, **LoanAmount**, **CreditScore**, **MonthsEmployed**, **NumCreditLines**, **InterestRate**, **LoanTerm**, **DTIRatio**), we plotted box plots to check for any specific outliers. We found that all numerical columns were within a fixed range, meaning they were inside the interquartile range, with no significant outliers deviating from this range (Figure 1).

For the categorical columns, we plotted count plots (Figure 2) using seaborn and observed that all classes in each categorical column were balanced, which is beneficial as it prevents bias in the training process. Therefore, we did not need to perform any further operations on them.

4. Dropping Duplicate Entries

We removed duplicate entries before starting our training process.

5. Columns Chosen for Training

We used all columns for training except **LoanID** and **Default**. The **LoanID** is a unique identifier for each person, so it is not useful for training. The **Default** column is the label we are trying to predict, so it is excluded as well. All other features are potentially related to the target label, so we included them for training since they might logically affect the outcome.

This was the basic preprocessing required for the dataset. For each model, we experimented with different combinations of encoding for categorical variables (such as using nominal encoding for some variables and ordinal encoding for others). For some models, we also created new features to assist in training. Details on these adjustments are provided in the respective model sections of the report and are documented in the python notebook. This approach allows all preprocessing to be run initially, so any model can be trained independently by running its respective cell.

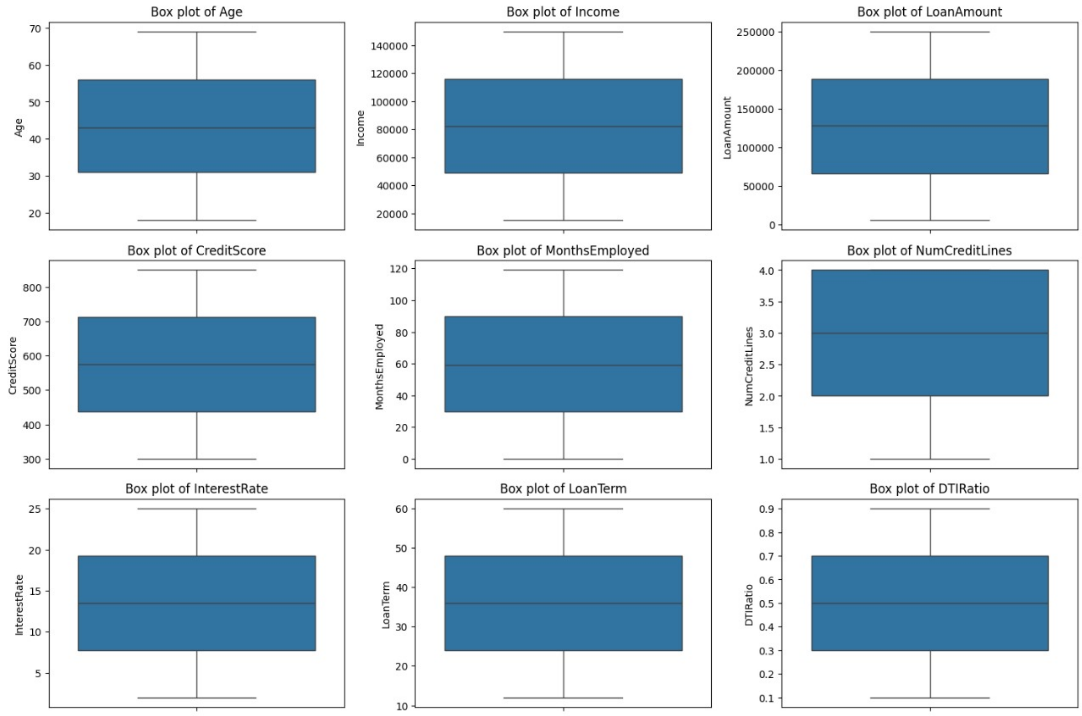


Figure 1: Box plots of numerical columns showing .

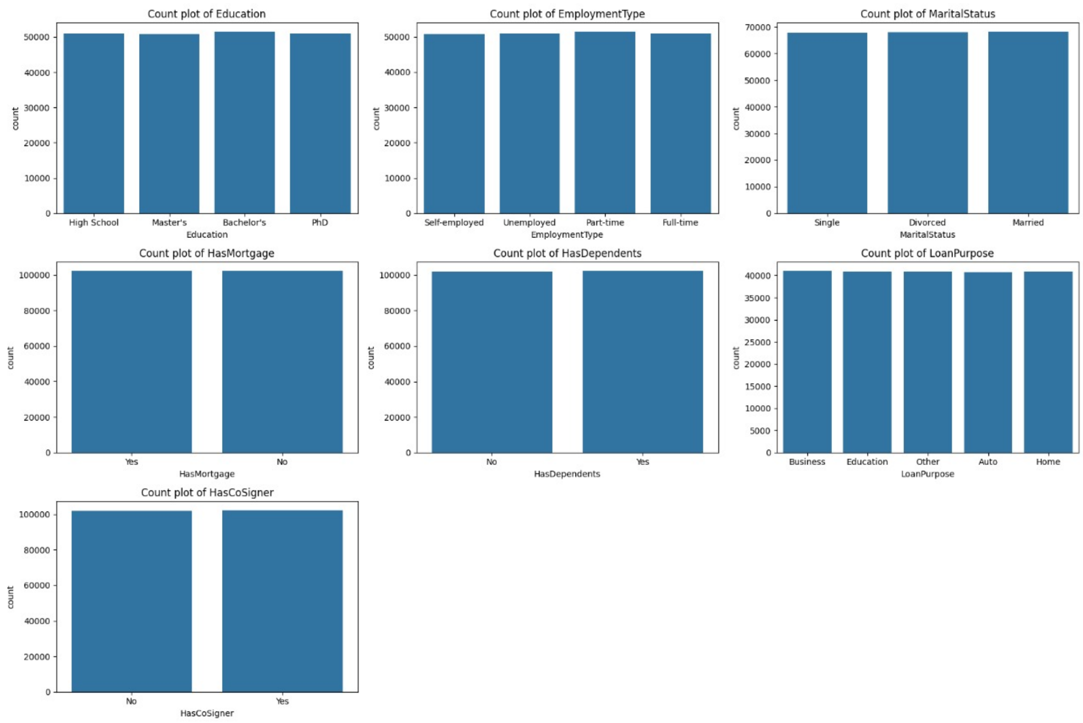


Figure 2: Count plots of categorical columns.

Instructions for Running the Notebook

1. Libraries required: **pandas**, **matplotlib**, **seaborn**, and **scikit-learn** (for modules like train-test split, encoders, classifiers, GridSearchCV, accuracy score, and StandardScaler).

2. First, run all the basic preprocessing steps provided at the beginning of the notebook.

3. Next, go to the respective cell (following the markdown headings) and run that cell to execute a specific model.

4. Each model-specific cell includes specific preprocessing, such as encoding categorical variables, splitting data into training and validation sets, and other model-specific adjustments.

5. After completing the execution of a model cell, rerun the basic preprocessing steps at the beginning before running other models. This ensures that any previous transformations, like encoding or variable drops, are reset before running a new model.

Reasoning: For each model, we experimented with different parameters, encoding combinations, and feature additions. Some preprocessing and feature adjustments were beneficial for one model but not as effective for others. By structuring the notebook this way, we could customize preprocessing for each model, which allowed us to test the all feature combinations and parameter settings that work best for each specific model.

Model Selection and Analysis

As this is a classification problem, our main approach was to use ensemble models based on decision trees(Random Forest). We also tried **K-Nearest Neighbors (KNN)** with different values of k, and **XGBoost**, a gradient boosting technique. Among these, our best-performing model was XGBoost. While the other models performed well, **XGBoost achieved the highest accuracy.**

1) Random Forest Classifier

Initially, we used the basic Random Forest Classifier without tuning parameters and encoded all categorical data as ordinal. We also experimented with one-hot encoding marital status and education while keeping other variables as ordinal. We achieved good accuracy of about 88.6%, and changing encoding methods did not significantly impact this result.

To improve, we used Grid Search CV to find the best parameters, including the number of estimators, samples, and leaf nodes. We tried expanding the parameter grid by increasing the number of estimators, but it was computationally expensive. Our system crashed, and even on Google Colab, no results were obtained after 3 hours. So, we used a smaller grid with 3-4 values for each parameter and found the best parameters.

Additionally, we created a new feature, **EmploymentLoan Ratio** (Months Employed / Loan Term), to assess job stability and repayment ability. However, this feature did not significantly enhance the model, possibly due to weak correlation or overlapping information. The accuracy remained around 88.6%.

2) K-Nearest Neighbors (KNN)

Similar to Random Forest, we experimented with feature engineering for KNN. We created a feature combining **has_cosigner, has_mortgage, and has_dependents**. However, this reduced accuracy slightly by 0.2%, as the feature's impact was inconsistent (e.g., having dependents is not always a negative indicator for loan approval).

Without additional features and fine-tuning, KNN initially achieved an accuracy of 87.6%, which was lower than Random Forest. After scaling numerical columns (important as KNN is distance-based) and adding features like **income_stability** (income / age), **credit_utilization** (LoanAmount/CreditScore), we improved KNN's performance. Using Grid Search CV with 3-fold cross-validation, the best value for k was found to be 12, yielding an accuracy of 88.6%.

This model achieved a cross-validation accuracy of 88.4% and a test accuracy of 88.6%.

3) XGBoost

We used XGBoost and started by encoding **marital status** nominally while treating other categorical variables as ordinal. This choice seemed suitable, as factors like education and employment type generally influence loan repayment ability, while marital status is less clear and better encoded nominally.
Using Grid Search CV, we found the best parameters to be:

- `colsample_bytree` = 0.6
- `learning_rate` = 0.1
- `max_depth` = 3
- `n_estimators` = 150
- `subsample` = 0.8

XGBoost achieved the highest test accuracy of 88.789%, making it our best model. Although increasing the number of estimators to 400-500 might slightly improve accuracy by 0.03-0.04%, this would be computationally expensive and would add 3-4 hours of training time.

The need to balance accuracy and training time informed our decision, as it is a trade-off between training time and accuracy. We prioritized a model that offers the best balance for large datasets, ensuring efficiency without sacrificing accuracy.

Conclusion

In this project, we explored multiple machine learning models to tackle a classification problem, including **Random Forest**, **K-Nearest Neighbors (KNN)**, and **XGBoost**. We performed preprocessing, such as handling missing values(none in our case), checking for outliers, and ensuring no class imbalance. Each model was fine-tuned using techniques like GridSearchCV to find the optimal parameters.

Among the models, **XGBoost** emerged as the best performer, achieving the highest accuracy. Random Forest also delivered strong results due to its ensemble nature and ability to capture feature importance, but it was slightly outperformed by XGBoost in terms of both accuracy and efficiency. KNN, on the other hand, initially struggled due to its sensitivity to scaling and distance metrics, but its performance improved significantly after scaling the features and optimizing the value of k. Despite these improvements, KNN's reliance on distance-based calculations made it less effective than the tree-based models on this dataset.

The superior performance of XGBoost can be attributed to its gradient boosting framework, which optimizes feature interactions more effectively and efficiently handles large datasets. Random Forest, while powerful, lacks the boosting mechanism of XGBoost, and KNN is more suited for smaller datasets or simpler classification problems.

This study highlights the importance of model selection and tuning, especially in datasets where slight feature interactions can influence predictions(0.1-0.2%).

PART -2

Data Preprocessing:

1. Handling Missing Values:

Identification: Missing values were identified using `isnull().sum()` to count the null values in each column.We found that there were no null values in any of the columns in both the train and test datasets. Hence, there was no need to impute any values in the data frame.

2. Outlier Detection and Removal:

Detection: Outliers in numerical columns were visualized using boxplots, which highlighted data points beyond the typical interquartile range (IQR).We found that all numerical columns were within a fixed range, meaning they were inside the interquartile range, with no significant outliers deviating from this range

3. Splitting Features and Target

- **Feature Selection**: The target variable (`Default`) was separated from the features (`X`). Non-informative columns, such as `LoanID`, were dropped as they do not contribute to model learning.
- **Categorical and Numerical Separation**: Features were categorized into numerical (e.g., income, age) and categorical (e.g., marital status, education) columns for tailored preprocessing.

4. Encoding Categorical Variables

- **Technique Used**: One-Hot Encoding was applied to categorical variables to convert them into binary vectors using `OneHotEncoder` from `sklearn.compose.ColumnTransformer`.
- **Reasoning**: One-Hot Encoding ensures the machine learning models interpret categorical data without assuming any ordinal relationships.

5. Standardization of Numerical Features

- **StandardScaler**: Standardization was applied to numerical features to scale them to a mean of 0 and a standard deviation of 1. This step prevents features with large scales from dominating the model's training process.
- **ColumnTransformer**: Both encoding and scaling were combined in a `ColumnTransformer` for efficient preprocessing.

6. Dataset Splitting

- **Training and Testing Split**: The dataset was split into training (80%) and testing (20%) sets using `train_test_split` with `stratify=y` to maintain the class distribution in both sets.
- **Class Distribution Check**: The class distribution in the target variable was verified before and after splitting to ensure balanced representation of both classes in training and testing sets.

7. Handling Class Imbalance

- **SMOTE (Synthetic Minority Oversampling Technique)**: SMOTE was employed on the training data to address the issue of class imbalance. This technique generates synthetic samples for the minority class, ensuring the model learns patterns from both classes effectively.
- **Effectiveness**: SMOTE improved the class balance, reducing bias toward the majority class and enhancing model performance on minority class predictions.

8. Final Preprocessing Pipeline

The entire preprocessing pipeline was implemented using the following steps:

1. **Numerical Scaling**: StandardScaler applied to numerical columns.
2. **Categorical Encoding**: One-Hot Encoding for categorical columns.
3. **Data Transformation**: Both transformations applied in a single pipeline for efficiency using `ColumnTransformer`.

This comprehensive preprocessing ensured that the data was clean, well-organized, and suitable for model training and evaluation.

Running the Notebook:

To use any specific model, start by running the preprocessing steps to prepare the dataset. Once preprocessing is complete, you can independently run the block for the desired model. If you wish to try another model, simply rerun the preprocessing steps and then execute the block for the next model.

Models Used and Analysis:

1. Logistic Regression

- **Description**: Logistic Regression was used as the baseline model for binary classification to predict whether a loan defaults (`Default = 1`) or not (`Default = 0`). It works by modeling the log-odds of the target variable as a linear combination of the features.
- **Implementation**:
    - The model was trained on standardized numerical features and one-hot encoded categorical variables after preprocessing.
    - The dataset's imbalance was not specifically addressed for this model.
- **Dataset Relevance**:
    - Logistic Regression helped establish a baseline accuracy and provided insight into how well the features linearly separate the target classes.
    - It was observed that the imbalanced dataset led to biased predictions favoring the majority class.

submission.csv
Complete (after deadline) · Mallikarjun Chakoti · 16h ago

Score: 0.88572
Public score: 0.88572

UPLOADED FILES

submission.csv (648 KiB)

ACCURACY:88.572%

2. Support Vector Machine (SVM)

- **Description**: The Support Vector Machine classifier was employed to separate classes by constructing a hyperplane with the maximum margin in the transformed feature space.
- **Implementation**:
  - A linear kernel was used to balance computational efficiency with model interpretability.
  - The preprocessed data was directly used for training.
  - The model was configured with `probability=True` to allow for probabilistic interpretation.
- **Dataset Relevance**:
  - The SVM model attempted to classify loans by finding the optimal boundary in the feature space.
  - However, like Logistic Regression, SVM faced challenges with the dataset's inherent imbalance, often biasing towards non-defaulting loans.

submission1.csv
Complete (after deadline) · Mallikarjun Chakoti · 14h ago

Score: 0.88447
Public score: 0.88447

UPLOADED FILES

submission1.csv (648 KiB)

ACCURACY:88.447%

3. Neural Network (MLPClassifier) Without Balancing

- **Description**: A multi-layer perceptron neural network was utilized to learn complex, non-linear relationships in the dataset.
- **Implementation**:
  - The neural network had one hidden layer with 64 neurons, using ReLU activation and the Adam optimizer.
  - The data imbalance was not addressed for this model, relying solely on the original class distribution.
- **Dataset Relevance**:
  - The neural network aimed to uncover intricate patterns in the features, which simpler models like Logistic Regression or SVM might miss.
  - It captured complex relationships between numerical and categorical variables but overfit to the majority class due to imbalance.

submission2.csv
Complete (after deadline) · Mallikarjun Chakoti · 14h ago

Score: 0.88729
Public score: 0.88729

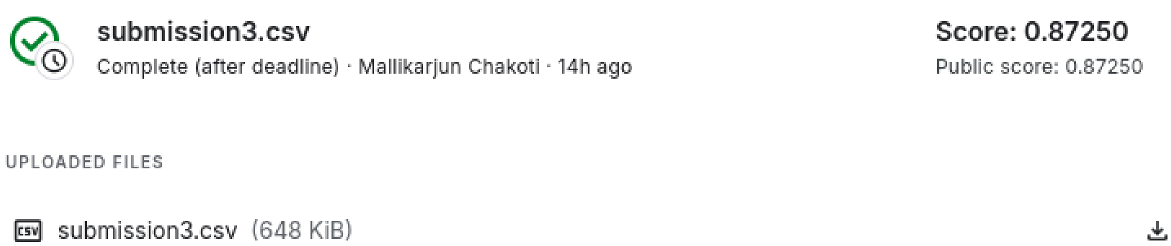
UPLOADED FILES

submission2.csv (648 KiB)

ACCURACY:88.729%

4. Neural Network (MLPClassifier) With SMOTE

- **Description**: To address class imbalance, Synthetic Minority Oversampling Technique (SMOTE) was applied to the training data before training a neural network.
- **Implementation**:
  - The same neural network architecture was used as above.
  - SMOTE balanced the dataset by generating synthetic samples for the minority class (defaulting loans).
- **Dataset Relevance**:
  - Balancing the dataset significantly improved the model's ability to detect defaulting loans.
  - This model better captured relationships that were underrepresented in the original imbalanced dataset.



ACCURACY:87.25%

Conclusion:

In the evaluation of various machine learning models for predicting loan defaults, the following results were obtained:

- **Logistic Regression** achieved an accuracy of **88.572%**, establishing a strong baseline for comparison.

- **Support Vector Machine (SVM)** attained an accuracy of **88.447%**, slightly below Logistic Regression, but it required the **highest training time** due to the complexity of the algorithm and the size of the dataset.
- **Neural Network (MLPClassifier)** without class balancing delivered an accuracy of **88.729%**, making it the **best-performing model** in terms of accuracy.
- **Neural Network (MLPClassifier) with SMOTE** yielded an accuracy of **87.250%**, which was slightly lower due to the added complexity introduced by synthetic samples. However, it significantly improved the recall for detecting defaulting loans, highlighting its utility in addressing the imbalanced dataset.

Overall, the **Neural Network without SMOTE** was identified as the best-performing model for this dataset due to its high accuracy and ability to capture complex patterns. While SMOTE-based balancing improved recall, it slightly compromised overall accuracy. Among the models, SVM was computationally the most intensive, requiring significant time to train. This analysis underscores the importance of selecting the right balance between model performance and computational efficiency based on the specific problem requirements.

Github repo link