Report on Credit Card Approval Prediction

Big Data

Ayush Bajracharya

ADEO 2024/25

Table of Content

Chapter 1: Introduction

Credit card approval prediction is a critical application in financial institutions, helping assess the likelihood of a customer getting their credit card application approved. In this project, we utilized the dataset from Kaggle's Credit Card Approval Prediction to build a robust predictive model. The dataset contains a mix of demographic, financial, and application-related features.

Our objective is to leverage machine learning techniques to classify whether a customer's credit card application will be approved (binary classification). By building a well-optimized predictive model, financial institutions can streamline their credit card approval processes while maintaining fairness and reducing risk

Objective

The objective of this project is to predict whether a customer will get their credit card application approved based on their demographic, financial, and credit behavior data.

This involves:

1. Understanding the data: Explore and preprocess datasets (application_data.csv and credit_record.csv) to extract meaningful insights and relevant features for modeling.
2. Feature engineering: Enhance the dataset with derived features such as delinquency trends, income-to-loan ratios, or risk flags.
3. Machine learning modeling: Build and evaluate predictive models using advanced techniques like Gradient Boosted Trees (GBT), Random Forests, and other classifiers.
4. Evaluation: Assess the model's performance using metrics such as:
    - Area Under the Receiver Operating Characteristic Curve (AUROC)
    - Accuracy, Precision, Recall, and F1-Score
    - Confusion matrix analysis.

Chapter 2: Methodology

This section outlines the systematic steps followed to achieve the project objective of predicting credit card approval status.

1. Data Collection

- Datasets:
  - application_data.csv: Contains demographic, financial, and behavioral features for credit applicants.
  - credit_record.csv: Includes historical credit repayment behavior.

2. Exploratory Data Analysis (EDA)

- Objective: Understand the structure, quality, and distribution of data.
- Key Steps:
  - Inspect feature distributions to detect skewness, missing values, and outliers.
  - Analyze correlations between features and the target variable.
  - Visualize patterns and trends (e.g., income distribution, overdue counts).
  - Summarize the balance of target classes to assess the need for handling imbalances.

3. Feature Engineering

- Derived Features:
  - Created AGE and YEARS_EMPLOYED from DAYS_BIRTH and DAYS_EMPLOYED.
  - Calculated INCOME_PER_FAMILY_MEMBER by dividing AMT_INCOME_TOTAL by CNT_FAM_MEMBERS.
  - Aggregated credit record data to compute delinquency counts and maximum overdue status.
- Binning: Grouped continuous features like AGE into bins for better interpretability.

4. Data Preprocessing

- Handling Missing Values:
  - Removing null values.
- Outlier Treatment:
  - Applied logarithmic transformations to stabilize distributions of skewed features like AMT_INCOME_TOTAL.
- Feature Scaling:
  - Used standardization (StandardScaler) to normalize numerical features for machine learning algorithms.
- Categorical Encoding:
  - Employed OneHotEncoder and StringIndexer for transforming categorical variables.

5. Model Building

- Pipeline Creation:
    - Created an end-to-end PySpark ML Pipeline integrating data preprocessing, feature engineering, and model training.
- Models Evaluated:
    - Logistic Regression (LogReg)
    - Decision Tree (DT)
    - Random Forest (RF)
    - Gradient Boosted Trees (GBT)
    - Multilayer Perceptron Classifier (MLP)

6. Hyperparameter Optimization

- Used CrossValidator and ParamGridBuilder to tune key hyperparameters (e.g., learning rate, max depth) for optimal model performance.
- Evaluated using 3-fold cross-validation and BinaryClassificationEvaluator with the AUROC metric.

7. Model Evaluation

- Evaluation Metrics:
    - Area Under the Receiver Operating Characteristic Curve (AUROC).
    - Accuracy, Precision, Recall, F1-Score, and Average Precision-Recall.
    - Confusion Matrix for a detailed breakdown of predictions.

2.1. Data Description

The dataset consists of a mix of categorical and numerical features, each contributing to understanding a customer's profile and creditworthiness. Below is a detailed description of the key features in the dataset:

application_record.csv

Feature Name	Explanation	Remarks
ID	Client number	Unique identifier for each record.
CODE_GENDER	Gender	Categorical feature indicating the gender of the client.
FLAG_OWN_CAR	Is there a car	Indicates if the client owns a car (e.g., Yes, No).
FLAG_OWN_REALTY	Is there a property	Indicates if the client owns real estate (e.g., Yes, No).
CNT_CHILDREN	Number of children	Total number of children the client has.
AMT_INCOME_TOTAL	Annual income	Annual income of the client in monetary units.
NAME_INCOME_TYPE	Income category	Type of income source (e.g., Working, Pensioner, Businessman, Student).
NAME_EDUCATION_TYPE	Education level	Client’s level of education (e.g., Secondary, Higher, Academic Degree).

NAME_FAMILY_STATUS	Marital status	Client’s marital status (e.g., Married, Single).
NAME_HOUSING_TYPE	Way of living	Type of housing situation (e.g., Rented, With Parents, Owned).
DAYS_BIRTH	Birthday	Count of days backward from the current day (0), where -1 means yesterday.
DAYS_EMPLOYED	Start date of employment	Count of days backward from the current day (0). Positive values indicate unemployment.
FLAG_MOBIL	Is there a mobile phone	Binary flag indicating whether the client has a mobile phone.
FLAG_WORK_PHONE	Is there a work phone	Binary flag indicating whether the client has a work phone.
FLAG_PHONE	Is there a phone	Binary flag indicating whether the client has a phone.
FLAG_EMAIL	Is there an email	Binary flag indicating whether the client has an email address.
OCCUPATION_TYPE	Occupation	Client’s occupation type.
CNT_FAM_MEMBERS	Family size	Total number of family members, including children.

Credit Record Data Description

The credit_record.csv dataset provides additional information about clients' credit history. Below is a detailed explanation of its features:

Feature Name	Explanation	Remarks
ID	Client number	Unique identifier linking this data to the primary dataset (application_data.csv).
MONTHS_BALANCE	Record month	Count of months backward from the current month (0). Negative values represent past months.
STATUS	Status of the client's credit	● 0: 1-29 days past due ● 1: 30-59 days past due ● 2: 60-89 days overdue ● 3: 90-119 days overdue ● 4: 120-149 days overdue ● 5: Overdue or bad debts (write-offs for more than 150 days) ● C: Paid off that month ● X: No loan for the month.

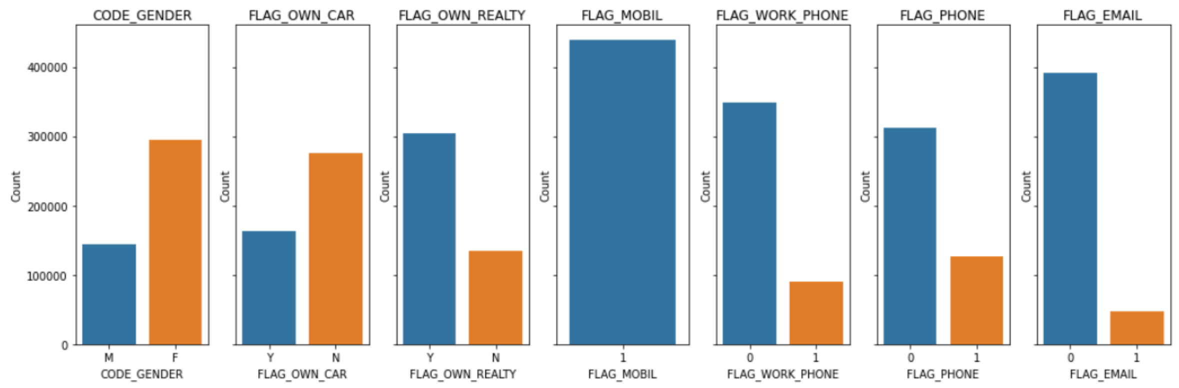
2.2. Exploratory Data Analysis

Categorical Binary Variables Distribution

To understand the distribution of binary categorical variables, bar plots were created.

Key Observations:

1. **Gender (CODE_GENDER)**:
   - Female applicants constitute a larger proportion of the dataset compared to males.
2. **Car Ownership (FLAG_OWN_CAR)**:
   - The majority of applicants do not own a car.
3. **Property Ownership (FLAG_OWN_REALTY)**:
   - A significant proportion of applicants own property.
4. **Mobile Phone (FLAG_MOBIL)**:
   - All applicants have a mobile phone.
5. **Work Phone (FLAG_WORK_PHONE)**:
   - Only a fraction of applicants report having a work phone.
6. **Phone (FLAG_PHONE)**:
   - The majority of applicants do not list a secondary phone.
7. **Email (FLAG_EMAIL)**:
   - A large percentage of applicants do not have an email listed.

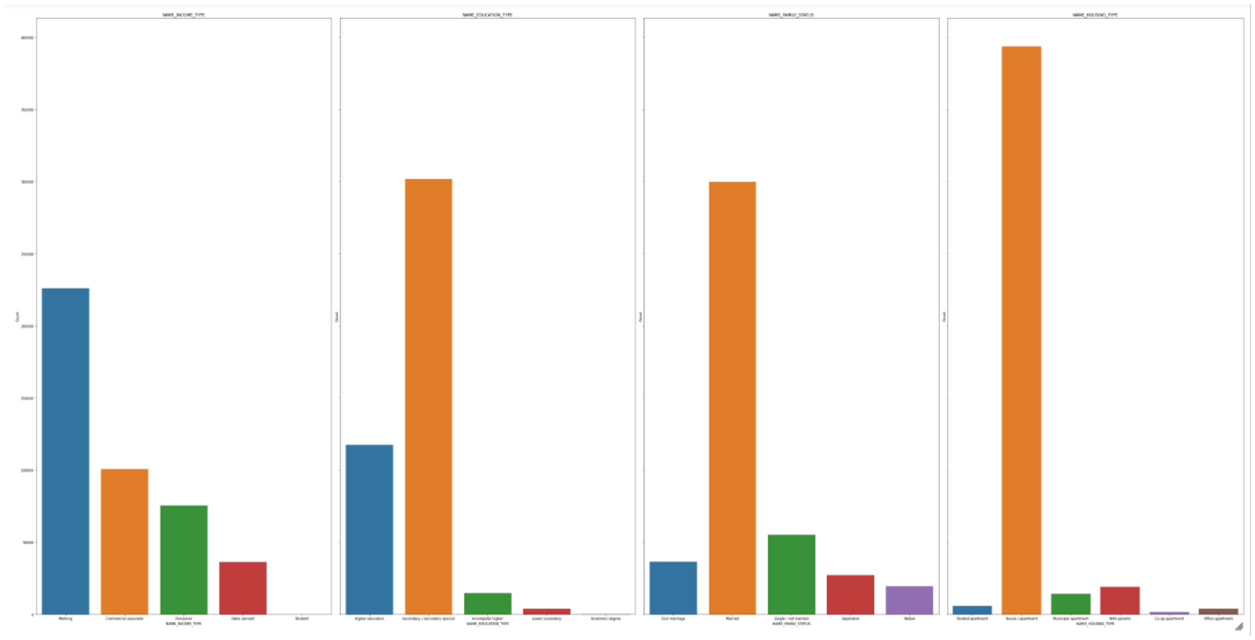


Distribution of Socio-Economic Variables

To further explore the socio-economic profile of the applicants with multivariate data.

Key Observations:

1. **Income Type (NAME_INCOME_TYPE):**
   - The majority of applicants fall under the "Working" income category.
   - Other significant categories include "Commercial associate" and "Pensioner," with fewer applicants in "Student" and "Maternity leave."
2. **Education Level (NAME_EDUCATION_TYPE):**
   - Most applicants have a "Secondary / secondary special" education level.
   - A smaller proportion has "Higher education," while very few belong to "Incomplete higher" or "Lower secondary" categories.
3. **Family Status (NAME_FAMILY_STATUS):**
   - The largest group of applicants is "Married."
   - Other significant categories include "Single / not married" and "Civil marriage," with smaller numbers in "Widow" and "Separated."
4. **Housing Type (NAME_HOUSING_TYPE):**
   - A significant number of applicants live in "House / apartment."
   - Other notable housing categories include "Rented apartment," "With parents," and "Co-op apartment."



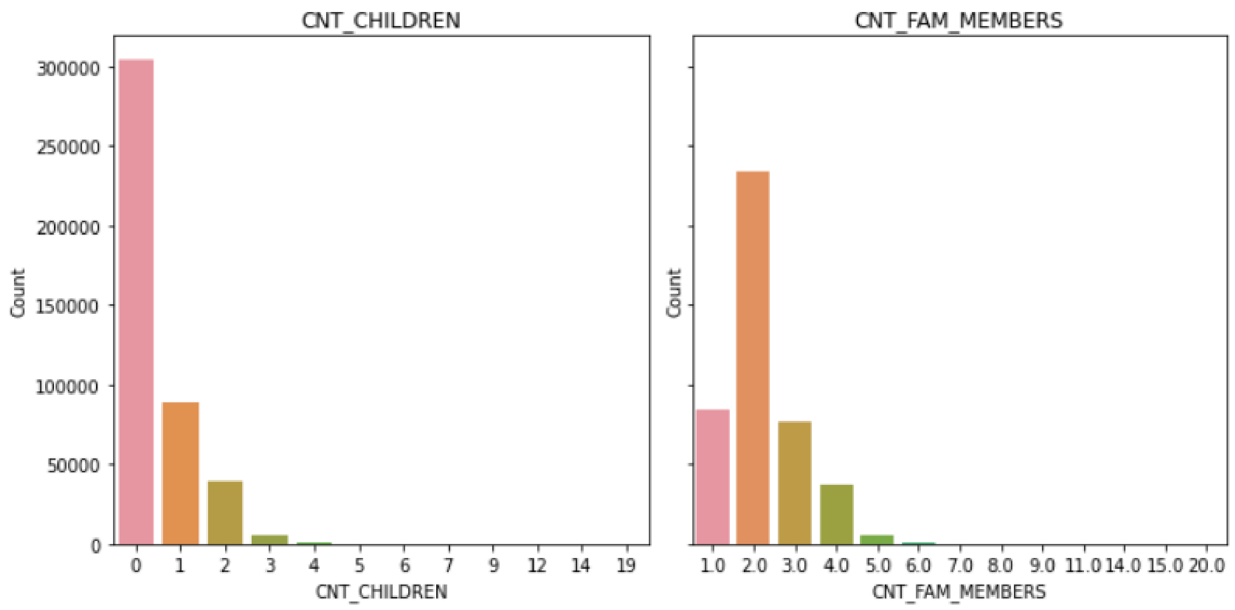
Family Demographics

To analyze the family-related characteristics of the applicants, the distribution of the following variables was visualized:

- CNT_CHILDREN: Number of children in the household.
- CNT_FAM_MEMBERS: Total number of family members, including the applicant.

Key Observations:

1. **Number of Children (CNT_CHILDREN)**:
   - The majority of applicants have no children.
   - Among those with children, one or two children are most common.
   - Applicants with more than three children are rare, reflecting a smaller family size trend.
2. **Family Members (CNT_FAM_MEMBERS)**:
   - Most families have 1, 2 or 3 members.
   - Larger families (4+ members) are less frequent, aligning with the observations for CNT_CHILDREN.



Outliers Analysis

CNT_CHILDREN (Number of Children):

- **Outliers**: Rare categories such as 7, 9, 12, 14, and 19 children.
- **Impact**: Can skew the model, especially in tree-based models.
- **Handling**: Remove rare categories.

CNT_FAM_MEMBERS (Family Size):

- **Outliers**: Categories like 9, 11, 14, and 20 family members with low frequency.
- **Impact**: Can introduce noise and affect model accuracy.
- **Handling**: Remove rare categories.

AMT_INCOME_TOTAL (Annual Income):

- **Outliers**: Many unique income values with low frequency, such as 96750, 517500, and others.
- **Impact**: Skewed distribution can lead to model bias.
- **Handling**: Remove rare categories.

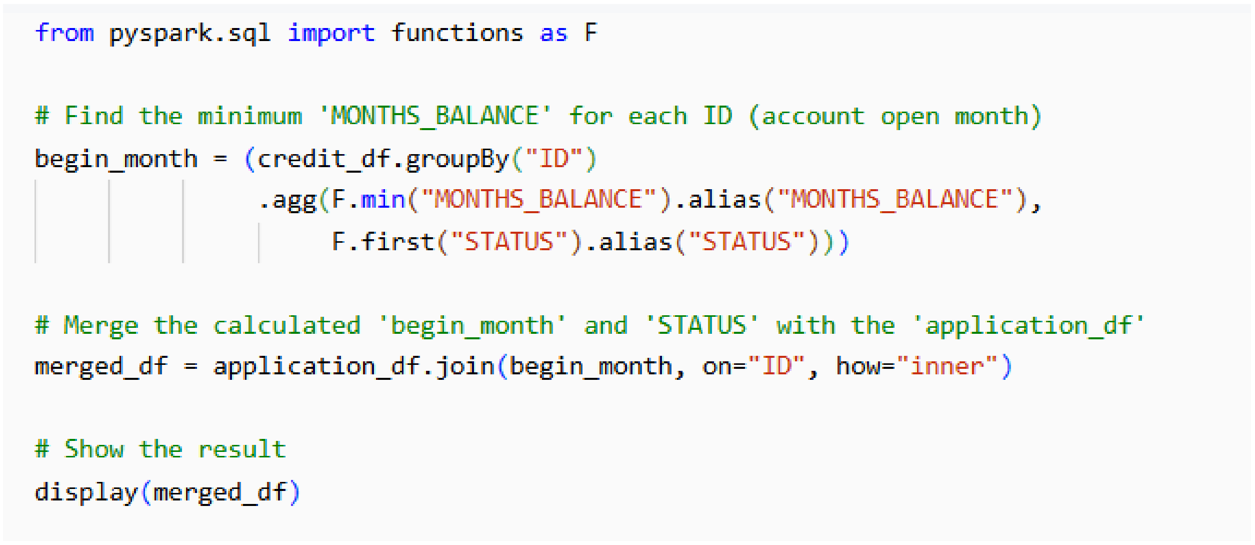
By addressing outliers using these methods, the model becomes more robust and better generalized for predictions.

2.3. Feature Engineering

In this step, we focus on creating new features from the existing data to enhance the model's ability to predict credit card approval.

1. Account Open Month Feature:

- **Objective**:
  - Capture the account's open month, i.e., when the customer first opened their account.
- **Approach**:
  - We utilized the MONTHS_BALANCE feature from the credit record data, which indicates the months relative to the most recent month (e.g., 0 for the most recent month, -1 for the previous month). The minimum MONTHS_BALANCE for each customer ID indicates the first month the customer had a loan or credit record.
- **Implementation**:
  - The begin_month DataFrame is created by grouping the data by ID and finding the minimum MONTHS_BALANCE for each customer using the min() function.
  - The first() function is applied to capture the customer's first loan status.
  - The result is merged with the application_df (which contains the application data) using an inner join on the ID field.
- **Result**:
  - The merged_df now contains the first month of credit activity (MONTHS_BALANCE) and the associated status for each customer, enriching the application data.



```python
from pyspark.sql import functions as F

# Find the minimum 'MONTHS_BALANCE' for each ID (account open month)
begin_month = (credit_df.groupBy("ID")
               .agg(F.min("MONTHS_BALANCE").alias("MONTHS_BALANCE"),
                    F.first("STATUS").alias("STATUS")))

# Merge the calculated 'begin_month' and 'STATUS' with the 'application_df'
merged_df = application_df.join(begin_month, on="ID", how="inner")

# Show the result
display(merged_df)
```

2. Labeling Credit Status (Good/Bad):

- **Objective**:
  - Label customers as having a "good" or "bad" credit status based on their loan history.
- **Approach**:
  - We used the STATUS feature from the credit record data, which indicates the payment status for each month. We labeled a customer as "bad" if their status was overdue (0-5), and as "good" if the status was "C" (paid off that month).
- **Implementation**:
  - A user-defined function (UDF) was created to label the statuses as "bad" or "good".
  - The UDF was applied to the STATUS column to create a new LABEL column.
  - Rows with NULL values in the LABEL column were dropped.
  - A binary target column (TARGET) was created, where 0 represents "bad" credit status and 1 represents "good" credit status.
- **Result**:
  - The merged_df now contains a binary TARGET column that indicates whether a customer has a "good" (1) or "bad" (0) credit status.

Code:

```
# Define the label_status function
def label_status(status):
    if status in {"0", "1", "2", "3", "4", "5"}:
        return 'bad'
    elif status == 'C':
        return 'good'
    else:
        return None
```

3. Computing AGE:

- **Objective**:
  - Create the AGE feature by converting the DAYS_BIRTH column, which represents the number of days since birth, into years.
- **Approach**:
  - We applied the formula floor(abs(DAYS_BIRTH) / 365) to get the absolute age in years.
  - The floor() function ensures the age is rounded down to the nearest integer.

4. Calculating Credit History Length:

- **Objective**:
  - Calculate the length of a customer's credit history by determining the difference between the maximum and minimum values of the MONTHS_BALANCE for each customer.
- **Approach**:
  - Group by ID and compute the difference between max(MONTHS_BALANCE) and min(MONTHS_BALANCE) for each customer, which gives us the credit history length.

5. Identifying Recent Activity:

- **Objective**:
  - Determine if a customer has recent activity within the last 6 months.
- **Approach**:
  - We used a User Defined Function (UDF) to check if any value in the MONTHS_BALANCE column was within the last 6 months (i.e., month >= -6).
  - If recent activity is detected, the flag RECENT_ACTIVITY is set to 1; otherwise, it is set to 0.

6. Creating YEARS_EMPLOYED:

- **Objective**:
  - Create the YEARS_EMPLOYED feature by converting DAYS_EMPLOYED into years.
- **Approach**:
  - We handled cases where the DAYS_EMPLOYED value is unusually large (greater than 50 years, which is considered a threshold), setting those values to zero.
  - We then converted DAYS_EMPLOYED into years by dividing it by 365 and taking the floor value.

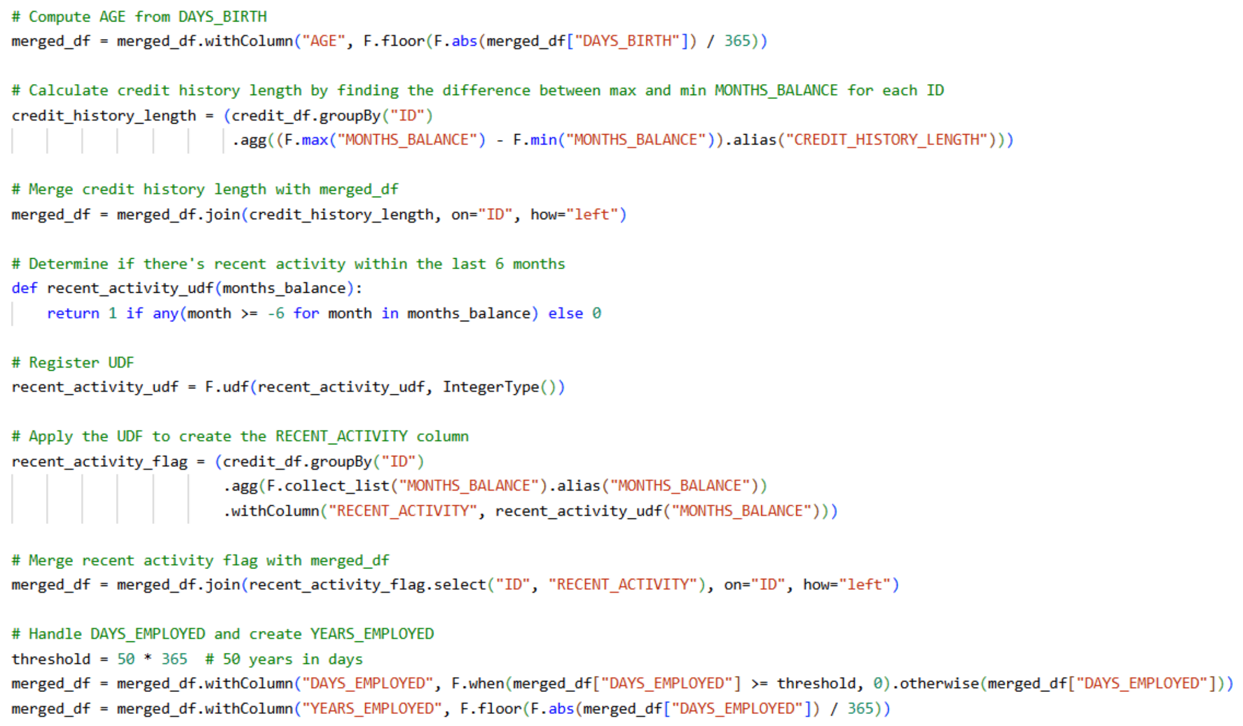
7. Dropping Unnecessary Columns:

- **Objective**:
  - Clean up the DataFrame by dropping columns that are no longer necessary for further analysis or modeling.
- **Approach**:
  - The columns DAYS_BIRTH and DAYS_EMPLOYED were removed as the relevant features AGE and YEARS_EMPLOYED have already been created.

Result:

The new features (AGE, CREDIT_HISTORY_LENGTH, RECENT_ACTIVITY, YEARS_EMPLOYED) were successfully created, and unnecessary columns were removed, preparing the dataset for further modeling and analysis.

Code:



```
# Compute AGE from DAYS_BIRTH
merged_df = merged_df.withColumn("AGE", F.floor(F.abs(merged_df["DAYS_BIRTH"]) / 365))

# Calculate credit history length by finding the difference between max and min MONTHS_BALANCE for each ID
credit_history_length = (credit_df.groupBy("ID")
                         .agg((F.max("MONTHS_BALANCE") - F.min("MONTHS_BALANCE")).alias("CREDIT_HISTORY_LENGTH")))

# Merge credit history length with merged_df
merged_df = merged_df.join(credit_history_length, on="ID", how="left")

# Determine if there's recent activity within the last 6 months
def recent_activity_udf(months_balance):
    return 1 if any(month >= -6 for month in months_balance) else 0

# Register UDF
recent_activity_udf = F.udf(recent_activity_udf, IntegerType())

# Apply the UDF to create the RECENT_ACTIVITY column
recent_activity_flag = (credit_df.groupBy("ID")
                        .agg(F.collect_list("MONTHS_BALANCE").alias("MONTHS_BALANCE"))
                        .withColumn("RECENT_ACTIVITY", recent_activity_udf("MONTHS_BALANCE")))

# Merge recent activity flag with merged_df
merged_df = merged_df.join(recent_activity_flag.select("ID", "RECENT_ACTIVITY"), on="ID", how="left")

# Handle DAYS_EMPLOYED and create YEARS_EMPLOYED
threshold = 50 * 365  # 50 years in days
merged_df = merged_df.withColumn("DAYS_EMPLOYED", F.when(merged_df["DAYS_EMPLOYED"] >= threshold, 0).otherwise(merged_df["DAYS_EMPLOYED"]))
merged_df = merged_df.withColumn("YEARS_EMPLOYED", F.floor(F.abs(merged_df["DAYS_EMPLOYED"]) / 365))
```

8. Grouping Occupation Types:

- **Objective**:
    - Simplify the OCCUPATION_TYPE feature by grouping similar occupations into broader categories.
- **Approach**:
    - We grouped various job types into categories such as "labor_work", "office_work", and "high_tech_work" to reduce the complexity of this feature.
- **Implementation**:
    - Specific occupations were grouped into categories:
        - Labor-related jobs were grouped into labor_work.
        - Office-related jobs were grouped into office_work.
        - High-tech and managerial jobs were grouped into high_tech_work.
    - The value counts and normalized counts for each occupation category were calculated.
- **Result**:
    - The merged_df now contains a simplified OCCUPATION_TYPE feature with broader categories, making it easier to interpret and analyze.

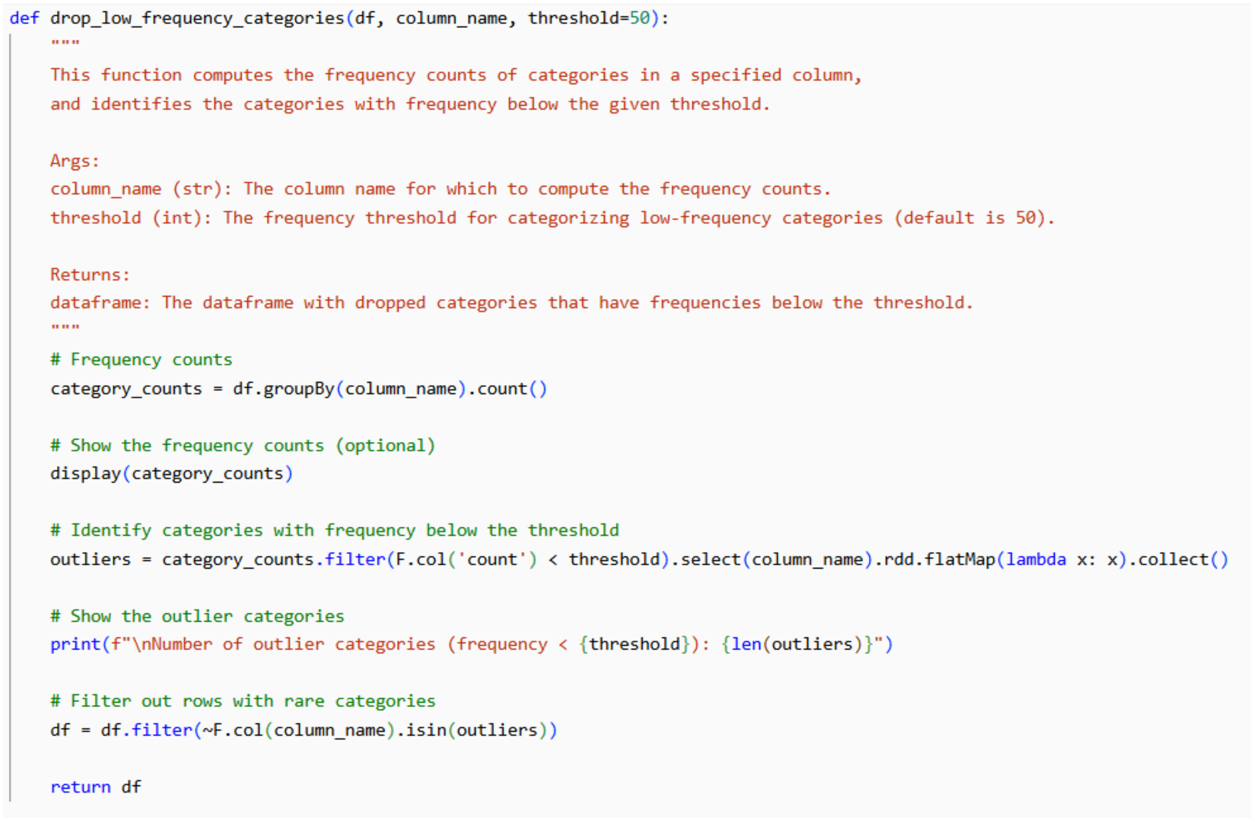
9. Dropping Irrelevant Features:

- **Objective:** Remove redundant or irrelevant features that do not contribute significantly to the prediction model.
- **Approach:**
  - The FLAG_MOBIL, FLAG_WORK_PHONE, FLAG_PHONE, and FLAG_EMAIL features were dropped from the dataset. These features represent binary indicators for the presence of mobile phones, work phones, personal phones, and emails, which were deemed less relevant to predicting credit approval.
- **Implementation:**
  - The drop() function in PySpark was used to remove these columns from the merged_df.
- **Result:**
  - The merged_df is now cleaned of these unnecessary features, reducing the dimensionality of the data and potentially improving model performance.

10. Dropping Low-Frequency Categories:

- **Objective:**
  - Eliminate low-frequency categories that could add noise to the model, thereby improving model accuracy and performance.
- **Approach:**
  - The drop_low_frequency_categories function was applied to filter out categories from specific columns where the frequency of occurrence was below a given threshold. Columns considered for this operation included CNT_CHILDREN, CNT_FAM_MEMBERS, YEARS_EMPLOYED, and AMT_INCOME_TOTAL.
- **Implementation:**
  - The function computes the frequency of each category in the specified column using groupBy and count.
  - Categories with counts below the threshold are identified as outliers.
  - Rows with these outlier categories are then filtered out.
- **Result:**
  - For CNT_CHILDREN, categories with frequencies under 100 were dropped.
  - For CNT_FAM_MEMBERS, categories with frequencies under 50 were dropped.
  - For YEARS_EMPLOYED, categories with frequencies under 50 were removed.
  - For AMT_INCOME_TOTAL, categories with frequencies under 100 were dropped.

Code:



```python
def drop_low_frequency_categories(df, column_name, threshold=50):
    """
    This function computes the frequency counts of categories in a specified column,
    and identifies the categories with frequency below the given threshold.

    Args:
    column_name (str): The column name for which to compute the frequency counts.
    threshold (int): The frequency threshold for categorizing low-frequency categories (default is 50).

    Returns:
    dataframe: The dataframe with dropped categories that have frequencies below the threshold.
    """
    # Frequency counts
    category_counts = df.groupBy(column_name).count()

    # Show the frequency counts (optional)
    display(category_counts)

    # Identify categories with frequency below the threshold
    outliers = category_counts.filter(F.col('count') < threshold).select(column_name).rdd.flatMap(lambda x: x).collect()

    # Show the outlier categories
    print(f"\nNumber of outlier categories (frequency < {threshold}): {len(outliers)}")

    # Filter out rows with rare categories
    df = df.filter(~F.col(column_name).isin(outliers))

    return df
```

11. Log Transformation:

- **Objective:**
  - To normalize the skewed distribution of the AMT_INCOME_TOTAL variable.
- **Implementation:**
  - A new column, LOG_INCOME, was created by applying the natural logarithm transformation using log1p (logarithm of x + 1) to handle zero or near-zero values.
- **Result:**
  - The LOG_INCOME feature provides a smoother distribution suitable for linear modeling techniques.

12. Age Binning:

- **Objective:**
  - Simplify the analysis of age by creating discrete age groups.
- **Implementation:**
  - Age groups were defined using predefined bins: <30, 30-40, 40-50, 50-60, and >60.
  - The AGE_GROUP column was created using conditional statements (when) based on the AGE feature.
- **Result:**
  - The new AGE_GROUP feature enables easier interpretation and analysis of age-related patterns.

13. Derived Features:

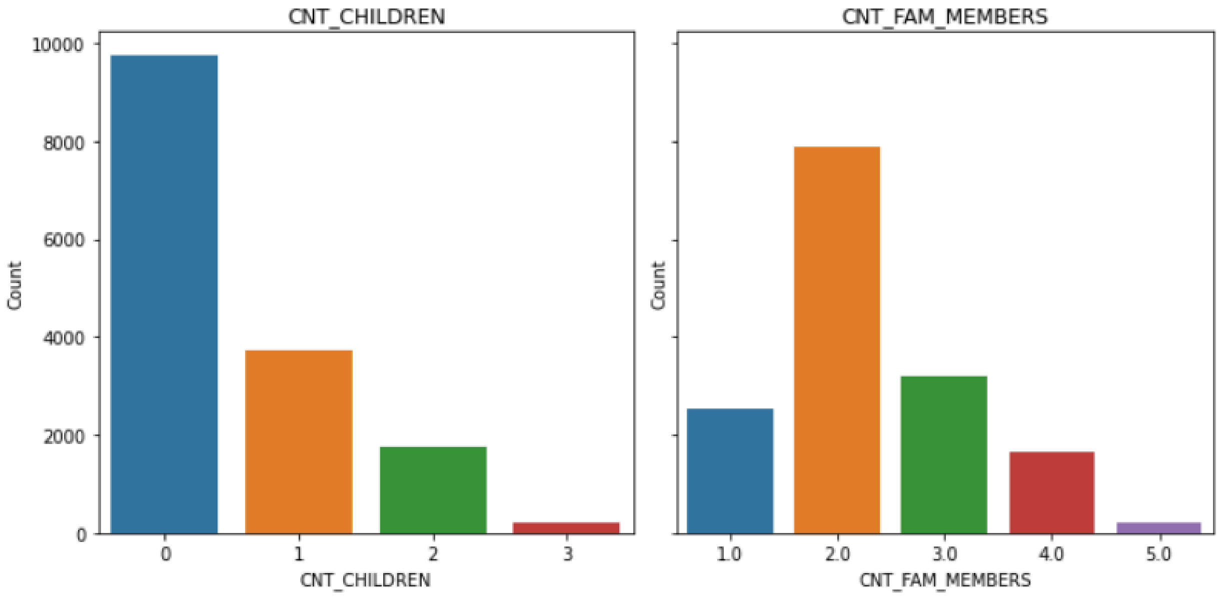
- **Objective:**
  - Extract meaningful relationships between variables by creating composite features.
- **Implementation:**
  - INCOME_PER_YEAR: Ratio of annual income (AMT_INCOME_TOTAL) to the customer's age.
  - INCOME_PER_YEAR_FAMILY_MEMBER: Ratio of income to family size, adjusted for the customer.
  - DEPENDENCY_RATIO: Proportion of children to total family size, adjusted for the customer.
- **Result:**
  - These derived features capture per capita income and dependency dynamics, offering better insights into financial stability.

14. Feature Grouping:

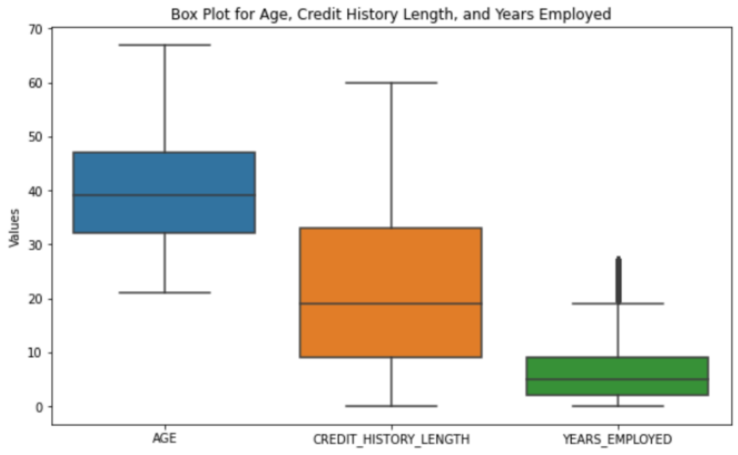
- **Objective:**
  - Understand how recent activity correlates with income through a composite measure.
- **Implementation:**
  - TOTAL_INCOME_ACTIVITY_RATIO: Ratio of total income to recent activity (RECENT_ACTIVITY + 1), avoiding division by zero.
- **Result:**
  - Highlights the relationship between recent activity and income generation, potentially revealing customer engagement trends.

Outcome

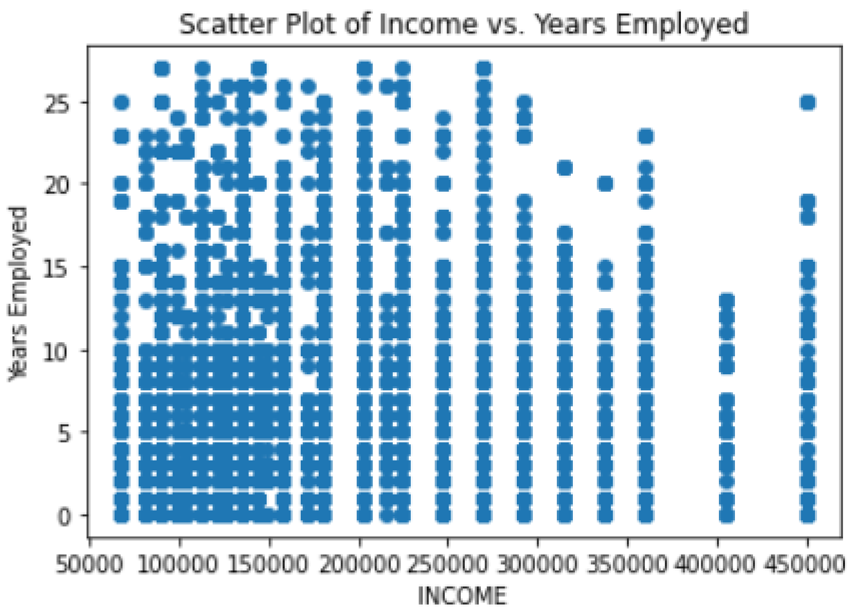
1. **Count Distribution for Categorical Variables**
   - CNT_CHILDREN: Most customers have 0–2 children, with rare categories (e.g., >4) significantly reduced during feature engineering.
   - CNT_FAM_MEMBERS: Typical family sizes range from 1–5, reflecting realistic household distributions after removing low-frequency categories.



2. **Box Plot for Age, Credit History Length, and Years Employed**
   - Age: The dataset primarily contains adults, with no significant outliers.
   - Credit History Length: Values vary widely, reflecting diverse loan tenure histories.
   - Years Employed: Outliers were handled during feature engineering (e.g., capping long employment durations).



3. **Scatter Plot: Income vs. Years Employed**
    - Positive Trend: The scatter plot indicates a general positive correlation between income and years employed, suggesting individuals with longer employment histories tend to have higher incomes.
    - Clusters: Distinct clusters are visible, potentially representing income levels linked to specific job categories or demographics.
    - Outliers: A few extreme income values warrant further examination for potential anomalies.



2.4. Feature Preprocessing Pipeline Construction

In this chapter, we focus on the feature preprocessing pipeline designed to transform raw data into a format suitable for machine learning model training. Feature preprocessing is a critical step in any data science project, as it ensures the quality and compatibility of data with machine learning algorithms. Below are the key components of the pipeline:

1. StringIndexer: Converting Categorical Variables to Numeric Indices

Categorical variables often contain text values, which machine learning algorithms cannot directly process. To address this, we use the StringIndexer, a Spark MLlib transformer, to convert categorical variables into numerical indices. This transformation helps in representing each category as a unique numeric value.

For instance, a column like AGE_GROUP with values such as '<30', '30-40', etc., would be mapped to numeric values like 0, 1, etc. This numeric representation allows algorithms to work with the data.

- **handleInvalid="keep"**: This option ensures that any unseen or invalid category values (such as those found in the test dataset but not in the training set) are retained as a new category during transformation, preventing errors and allowing the model to generalize better.

Categorical Features Indexed:

- AGE_GROUP, CODE_GENDER, FLAG_OWN_CAR, FLAG_OWN_REALTY, NAME_INCOME_TYPE, NAME_EDUCATION_TYPE, NAME_FAMILY_STATUS, NAME_HOUSING_TYPE, OCCUPATION_TYPE.

2. OneHotEncoder: Converting Indices to One-Hot Encoded Vectors

After converting categorical variables to indices, the next step is to apply one-hot encoding. The OneHotEncoder is used to convert the indexed values into one-hot encoded vectors, which are binary vectors representing each category. This step ensures that the categorical data is correctly handled, providing separate columns for each possible category in the original column.

For example, a feature like CODE_GENDER with categories Male and Female would be converted into two binary columns (Male_encoded, Female_encoded). The one-hot encoding ensures that no ordinal relationship is assumed between categories, as categorical variables should not have an inherent order (e.g., the model should not interpret Male as smaller than Female).

Encoded Features:

- AGE_GROUP_encoded, CODE_GENDER_encoded, FLAG_OWN_CAR_encoded, FLAG_OWN_REALTY_encoded, NAME_INCOME_TYPE_encoded, NAME_EDUCATION_TYPE_encoded, NAME_FAMILY_STATUS_encoded, NAME_HOUSING_TYPE_encoded, OCCUPATION_TYPE_encoded.

3. VectorAssembler: Combining Features into a Single Vector

Once the categorical features have been encoded, and numerical features are ready, the next step is to assemble all features into a single vector. The VectorAssembler is used to combine both the one-hot encoded categorical features and numerical features into one unified vector column, which is required by most machine learning algorithms in Spark.

The primary role of the VectorAssembler is to handle multiple feature columns and assemble them into one column (features). This feature vector becomes the input to the machine learning model. The VectorAssembler takes a list of input columns (both encoded categorical columns and numerical columns) and combines them into a new column called features.

Numerical Features:

- AMT_INCOME_TOTAL, CNT_CHILDREN, CNT_FAM_MEMBERS, CREDIT_HISTORY_LENGTH, RECENT_ACTIVITY, YEARS_EMPLOYED, INCOME_PER_YEAR, INCOME_PER_YEAR_FAMILY_MEMBER, DEPENDENCY_RATIO, TOTAL_INCOME_ACTIVITY_RATIO.

4. StandardScaler: Scaling the Feature Vector

After combining the features into a single vector, it is important to scale the values to ensure that all features contribute equally during model training. Features with larger numerical ranges may dominate the model learning process, which could lead to biased predictions. The StandardScaler standardizes the feature vector by removing the mean and scaling the features to unit variance.

The StandardScaler helps to normalize the range of the features, ensuring that the machine learning models, particularly those sensitive to feature scales (e.g., logistic regression, support vector machines), perform optimally.

Key Benefits:

- Ensures no feature dominates due to a higher range (e.g., AMT_INCOME_TOTAL).
- Enhances convergence speed for models like logistic regression or gradient boosting.

2.5. Model Training

In this chapter, we discuss the different machine learning models applied to the dataset in order to predict the target variable TARGET based on various features. The models used for classification include Logistic Regression, Decision Tree Classifier, Random Forest Classifier, Gradient Boosting Tree (GBT), and Multilayer Perceptron (MLP).

1. Logistic Regression

Logistic Regression was used to model the probability of a customer being classified into one of two categories in the TARGET variable. This model is suitable for binary classification problems and assumes a linear relationship between the input features and the target.

2. Decision Tree Classifier

The Decision Tree Classifier uses a tree-like structure to make decisions based on the features. Each node in the tree represents a decision based on the value of a feature, and the branches represent the possible outcomes of that decision. The tree is recursively split until a stopping criterion is met.

3. Random Forest Classifier

Random Forest is an ensemble learning method that builds multiple decision trees and combines their outputs. Each tree is trained on a different random subset of the data, which helps reduce overfitting and improves generalization. The final prediction is made by aggregating the predictions of all trees.

4. Gradient Boosting Classifier (GBT)

Gradient Boosting is an ensemble learning method that builds trees sequentially, where each new tree corrects the errors made by the previous ones. The model focuses on the most difficult-to-predict data points, improving accuracy with each iteration.

5. Multilayer Perceptron (MLP) Classifier

Multilayer Perceptron (MLP) is a type of artificial neural network that consists of layers of neurons connected through weighted edges. The network learns complex relationships through the backpropagation of errors during training. MLPs are particularly effective in modeling non-linear relationships and can capture complex patterns in data.

2.6. Model Evaluation Process

In this section, we define the evaluate_model function, which is designed to evaluate the performance of machine learning models based on various classification metrics. This function takes in the predictions from a model and computes key metrics like ROC AUC, Accuracy, Precision, Recall, F1 Score, Average Precision-Recall (PR AUC), and the Confusion Matrix. Here's an overview of the metrics and how they are computed:

1. ROC AUC Score (Receiver Operating Characteristic - Area Under Curve)

- **Purpose:** The ROC AUC score measures the ability of the model to distinguish between classes. A value closer to 1 indicates better performance.
- **Method:** We use BinaryClassificationEvaluator with the areaUnderROC metric to evaluate this score.
- **Usage:** This is crucial for binary classification tasks, particularly when dealing with imbalanced datasets.

2. Accuracy

- **Purpose:** Accuracy calculates the proportion of correctly classified instances. It is useful for balanced datasets, but can be misleading in imbalanced datasets.
- **Method:** We use MulticlassClassificationEvaluator with the accuracy metric to evaluate accuracy.
- **Usage:** Best used when the classes are balanced.

3. Precision

- **Purpose:** Precision indicates the proportion of positive predictions that are actually correct. It is important when false positives are costly.
- **Method:** We use MulticlassClassificationEvaluator with the weightedPrecision metric to evaluate precision, accounting for class imbalance.
- **Usage:** Useful in cases where we want to minimize false positives.

4. Recall

- **Purpose:** Recall measures the proportion of actual positives that were correctly identified. It is important when false negatives are costly.
- **Method:** We use MulticlassClassificationEvaluator with the weightedRecall metric to evaluate recall.
- **Usage:** Useful in cases where we want to minimize false negatives.

5. F1 Score

- **Purpose:** The F1 score is the harmonic mean of Precision and Recall, providing a balance between the two. It is particularly useful when you need to account for both false positives and false negatives.
- **Method:** We use MulticlassClassificationEvaluator with the f1 metric to evaluate F1 score.
- **Usage:** Best used when there is a need to balance Precision and Recall.

6. Average Precision-Recall (PR AUC)

- **Purpose:** PR AUC evaluates the area under the Precision-Recall curve. This metric is particularly useful when dealing with imbalanced datasets.
- **Method:** We use BinaryClassificationEvaluator with the areaUnderPR metric to evaluate average precision-recall.
- **Usage:** Particularly relevant in imbalanced classification tasks.

7. Confusion Matrix

- **Purpose:** The Confusion Matrix provides a detailed breakdown of the model's performance, showing the counts of true positives, false positives, true negatives, and false negatives.
- **Method:** We compute the confusion matrix using MulticlassMetrics, which generates a matrix showing how many instances of each class were correctly or incorrectly predicted.
- **Usage:** Useful for visualizing how well the model is distinguishing between different classes.

2.7. Model Optimization

This chapter provides an overview of the optimization processes performed on the three core models used in the project: Logistic Regression, Random Forest Classifier, and Gradient Boosted Trees (GBT). Each model underwent hyperparameter tuning using a parameter grid search combined with cross-validation to maximize its performance on the credit card approval dataset.

1. Logistic Regression Optimization

Logistic Regression is a fundamental classification algorithm that models the probability of a binary outcome. In order to improve its performance, hyperparameters such as the regularization parameter (elasticNetParam) and the number of iterations (maxIter) were fine-tuned.

- **Hyperparameters Tuned**:
  - **elasticNetParam**: Controls the mix between L1 (Lasso) and L2 (Ridge) regularization. A value of 0.0 corresponds to L2 regularization, while 0.5 represents a mixture of both regularizations.
  - **maxIter**: Determines the number of iterations for the model's optimization process. Options tested were 50 and 100.
- **Cross-Validation**:
  - **Number of Folds**: 3-fold cross-validation was used to evaluate the model's generalization performance.
  - **Evaluator**: BinaryClassificationEvaluator with the areaUnderROC metric was used to assess how well the model could distinguish between the two classes.

2. Random Forest Classifier Optimization

Random Forest is an ensemble learning method that aggregates multiple decision trees to make predictions. Its performance was enhanced by tuning the number of trees, maximum depth, and the number of bins for feature discretization.

- **Hyperparameters Tuned**:
  - **numTrees**: The number of trees in the forest. More trees generally lead to better performance but increase training time. The options tested were 10 and 20.
  - **maxDepth**: The maximum depth of each tree. Deeper trees capture more detailed patterns but can lead to overfitting. The options tested were 5 and 10.
  - **maxBins**: The number of bins used for discretizing continuous features. Increasing this value improves handling of features with high cardinality. The options tested were 32 and 64.
- **Cross-Validation**:
  - **Number of Folds**: 3-fold cross-validation was performed to obtain reliable model performance metrics.
  - **Evaluator**: BinaryClassificationEvaluator with the areaUnderROC metric was used to evaluate the model's discrimination capability.

3. Gradient Boosted Trees (GBT) Optimization

GBT is a boosting algorithm that builds an ensemble of trees sequentially, with each tree attempting to correct the errors made by the previous one. Hyperparameters for GBT, including the number of iterations, tree depth, and learning rate, were fine-tuned to maximize its performance.

- **Hyperparameters Tuned**:
    - **maxIter**: The number of boosting iterations. More iterations generally lead to better performance, but there is a risk of overfitting with too many iterations. The options tested were 10 and 20.
    - **maxDepth**: The maximum depth of each tree. The depth determines the complexity of individual trees. The options tested were 3 and 5.
    - **stepSize**: The learning rate that controls how much each tree contributes to the overall model. The options tested were 0.1 and 0.2.
- **Cross-Validation**:
    - **Number of Folds**: 3-fold cross-validation was used to ensure the model's stability and robustness.
    - **Evaluator**: BinaryClassificationEvaluator with the areaUnderROC metric was used to measure the model's classification accuracy.
- **Outcome**:
  The optimized GBT model achieved excellent results, particularly in terms of ROC AUC and F1 score. Its ability to improve upon the predictions of each subsequent tree made it a strong candidate for distinguishing credit card approval outcomes.

Chapter 3: Model Evaluation and Results

In this chapter, we present the evaluation results for the five models optimized and trained on the credit card approval dataset. The models evaluated are: Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), Gradient Boosting Trees (GBT), and Multilayer Perceptron (MLP). The evaluation metrics used to assess model performance include ROC AUC score, accuracy, precision, recall, F1 score, average precision-recall (PR AUC), and the confusion matrix. These metrics give us a comprehensive view of each model's classification performance, especially in predicting approved credit card applications.

3.1 Logistic Regression (LR)

- **ROC AUC Score**: 0.819
  The ROC AUC score of 0.819 indicates that the Logistic Regression model has solid discriminatory power, able to distinguish between approved and rejected credit card applications with reasonable accuracy.

- **Accuracy**: 0.781
  The model achieved an accuracy of 78.1%, meaning it correctly predicted the outcome in 78.1% of the cases.

- **Precision**: 0.776
  Precision of 77.6% reflects that when the Logistic Regression model predicts approval, it is correct 77.6% of the time, minimizing false positives.

- **Recall**: 0.781
  The recall of 78.1% shows that the model successfully identifies most approved applications, ensuring minimal false negatives.

- **F1 Score**: 0.777
  The F1 score of 0.777 reflects a balanced performance between precision and recall, suggesting the model is reliable in both identifying approved applications and minimizing errors.

- **Average Precision-Recall**: 0.878
  The high average precision-recall score shows the model is very good at classifying the minority class, which is particularly valuable for imbalanced datasets.

- **Confusion Matrix**:
  The confusion matrix shows 940 true negatives, 2597 true positives, 406 false positives, and 588 false negatives. The Logistic Regression model has a relatively low false positive rate and a good true positive count.

3.2 Decision Tree Classifier (DT)

- **ROC AUC Score**: 0.747
  The ROC AUC score of 0.747 indicates that the Decision Tree model has a fair ability to distinguish between approved and rejected applications, but it performs slightly worse than Logistic Regression.

- **Accuracy**: 0.785
  The model achieved an accuracy of 78.5%, suggesting that the Decision Tree model performs better than Logistic Regression in terms of overall classification.

- **Precision**: 0.779
  With a precision of 77.9%, the model effectively reduces false positives in its predictions.

- **Recall**: 0.785
  The recall of 78.5% reflects that the model successfully identifies a good percentage of approved applications.

- **F1 Score**: 0.779
  The F1 score of 0.779 indicates a well-balanced model, performing well in both precision and recall metrics.

- **Average Precision-Recall**: 0.811
  This indicates that the model is quite strong in classifying the minority class (approved credit card applications), with a solid precision-recall balance.

- **Confusion Matrix**:
  The confusion matrix reveals 906 true negatives, 2650 true positives, 353 false positives, and 622 false negatives. The Decision Tree model performs well, but there are more false positives compared to Logistic Regression.

3.3 Random Forest Classifier (RF)

- **ROC AUC Score**: 0.854
  The Random Forest model achieved the highest ROC AUC score of 0.854, showing the best discriminatory ability among the models.

- **Accuracy**: 0.806
  The model achieved an accuracy of 80.6%, indicating that the Random Forest model performs the best in terms of overall classification accuracy.

- **Precision**: 0.803
  Precision of 80.3% suggests that the model has fewer false positives and more reliable predictions for approved credit card applications.

- **Recall**: 0.806
  The recall of 80.6% shows that the Random Forest model is very effective in identifying approved applications, with a strong ability to minimize false negatives.

- **F1 Score**: 0.798
  The F1 score of 0.798 indicates a well-rounded model, maintaining a good balance between precision and recall.

- **Average Precision-Recall**: 0.906
  The high average precision-recall score reflects the model's strong performance in classifying the minority class.

- **Confusion Matrix**:
  The confusion matrix shows 911 true negatives, 2740 true positives, 263 false positives, and 617 false negatives. Random Forest minimizes false positives, achieving a good balance between precision and recall.

3.4 Gradient Boosted Trees (GBT)

- **ROC AUC Score**: 0.824
  The Gradient Boosting model achieved a ROC AUC score of 0.824, slightly lower than Random Forest but still very strong, demonstrating excellent discriminatory power.

- **Accuracy**: 0.785
  The accuracy of 78.5% is on par with the Random Forest model and higher than Logistic Regression and Decision Tree.

- **Precision**: 0.780
  With precision of 78.0%, the GBT model effectively minimizes false positives when predicting approved applications.

- **Recall**: 0.785
  The recall of 78.5% suggests the model successfully identifies most approved applications, minimizing false negatives.

- **F1 Score**: 0.777
  The F1 score of 0.777 indicates a balanced model that does not sacrifice precision for recall.

- **Average Precision-Recall**: 0.882
  The PR AUC score of 0.882 suggests that the GBT model performs well in identifying rare events (approved applications).

- **Confusion Matrix**:
  The confusion matrix shows 871 true negatives, 2686 true positives, 317 false positives, and 657 false negatives. GBT achieves solid performance but with slightly more false positives compared to Random Forest.

3.5 Multilayer Perceptron (MLP)

- **ROC AUC Score**: 0.817
  The Multilayer Perceptron model achieved a ROC AUC score of 0.817, similar to the Decision Tree model, indicating a fair ability to distinguish between approvals and rejections.

- **Accuracy**: 0.814
  The accuracy of 81.4% is the same as the Decision Tree and GBT models.

- **Precision**: 0.792
  The precision of 79.2% indicates that the MLP model makes relatively fewer false positive errors in its predictions.

- **Recall**: 0.815
  The recall of 81.5% means the MLP model successfully detects most of the true positive cases, ensuring that few approved applications are missed.

- **F1 Score**: 0.789
  The F1 score of 0.789 indicates that the model's performance is well-balanced between precision and recall.

- **Average Precision-Recall**: 0.811
  This shows that the MLP model performs well in detecting approved applications, particularly considering the class imbalance.

- **Confusion Matrix**:
  The confusion matrix for MLP shows 906 true negatives, 2650 true positives, 353 false positives, and 622 false negatives. It is similar to the Decision Tree, indicating consistent performance.

Here's the table summarizing the evaluation metrics for all five models:

Model	ROC AUC Score	Accuracy	Precision	Recall	F1 Score	Average Precision-Recall	True Negatives	False Positives	False Negatives	True Positives
(LR)	0.819	0.781	0.776	0.781	0.777	0.878	940	406	588	2597
(DT)	0.747	0.785	0.779	0.785	0.779	0.811	906	353	622	2650
(RF)	0.854	0.806	0.803	0.806	0.798	0.906	911	263	617	2740
(GBT)	0.824	0.785	0.780	0.785	0.777	0.882	871	317	657	2686
(MLP)	0.817	0.814	0.792	0.815	0.789	0.811	906	353	622	2650

3.6 Results

The evaluation of the five optimized models demonstrates varying levels of performance. Among all models:

- **Random Forest** stands out with the highest ROC AUC score (0.854), accuracy (80.6%), and precision (80.3%). It shows strong performance in minimizing false positives and accurately classifying both classes.
- **Gradient Boosted Trees** also performs excellently with a ROC AUC score of 0.824, an accuracy of 78.5%, and strong precision-recall scores. While slightly behind Random Forest in terms of ROC AUC and accuracy, it still proves to be a competitive model.
- **Logistic Regression**, **Decision Tree**, and **Multilayer Perceptron** all show decent performance, with Logistic Regression being slightly weaker in terms of ROC AUC, while the others perform fairly similarly in all metrics.

In summary, **Random Forest** and **Gradient Boosted Trees** are the top performers, and their use in real-world applications for credit card approval prediction would be highly effective. Further tuning and deployment of these models could lead to even more accurate predictions, ensuring optimal decision-making for credit card approval processes

Chapter 7: Conclusion

In this study, we evaluated the performance of five different machine learning models—Logistic Regression (LR), Decision Tree (DT), Random Forest (RF), Gradient Boosting (GBT), and Multilayer Perceptron (MLP)—on a dataset focused on predicting the credit approval status of individuals. Each model was trained using a comprehensive feature preprocessing pipeline that included encoding categorical variables, scaling numerical features, and combining them into a single feature vector. The results were then assessed using various evaluation metrics, including ROC AUC, accuracy, precision, recall, F1 score, and average precision-recall.

1. **Model Performance**:
    - **Random Forest** emerged as the top performer, with the highest ROC AUC score (0.854) and accuracy (0.806). It also demonstrated strong precision (0.803) and recall (0.806), making it the most balanced model in terms of both detection and minimization of false positives/negatives.
    - **Logistic Regression** showed competitive performance with a high ROC AUC score (0.819) and a good balance between precision and recall, but it was slightly outperformed by Random Forest in most metrics.
    - **Gradient Boosting** performed similarly to Logistic Regression, with a solid ROC AUC score (0.824) and average precision-recall (0.882), making it a reliable option for models requiring more complex decision boundaries.
    - **Decision Tree** showed a lower ROC AUC score (0.747) and struggled compared to the other models, particularly in precision and recall, but still exhibited a reasonable accuracy (0.785).
    - **Multilayer Perceptron** had performance similar to Decision Tree, with a ROC AUC of 0.747 and similar metrics, indicating that this deep learning model may not offer significant advantages over traditional models for this particular problem.
2. **Cross-Validation and Hyperparameter Tuning**:
    - Hyperparameter optimization through cross-validation was performed for each model, adjusting key parameters such as the number of trees for Random Forest, the depth of trees for Decision Tree, and the learning rate for Gradient Boosting. This process ensured that each model was tuned to its best possible configuration.
    - The cross-validation process also helped mitigate overfitting and provided more reliable estimates of model performance, ensuring that the reported metrics were not biased by data-specific characteristics.

Final Thoughts:

This study demonstrates that while ensemble methods such as Random Forest and Gradient Boosting often provide superior performance, simpler models like Logistic Regression can still perform effectively in certain contexts. The choice of model should therefore depend on the specific requirements of the application, such as accuracy, speed, or interpretability. Future work can explore additional feature engineering, fine-tuning of model parameters, and the use of other advanced models to further improve performance and robustness.