# Credit Card Default Prediction

Sri Vishal Kotari

## INTRODUCTION & DOMAIN KNOWLEDGE

Predicting credit card defaults is a vital aspect of financial risk management. Lenders and financial institutions need reliable models to forecast whether a customer will miss their payments, as this helps mitigate potential losses and manage credit effectively. A **default** occurs when a borrower fails to meet their payment obligations on time. Accurate predictions allow institutions to make informed decisions, such as adjusting credit limits or offering payment plans, to reduce risks.

The dataset for this project includes historical data from 30,000 clients in Taiwan, covering demographics, bill amounts, previous payments, and payment history. The main challenge is class imbalance, as only a small percentage of clients default on payments, which can skew machine learning models toward the majority class.

In this project, we leverage **supervised machine learning techniques** to predict a customer's likelihood of default based on historical data. The dataset captures multiple dimensions of client behavior, including demographic details, credit limits, past bills, repayment patterns, and payment amounts over time. However, a key challenge lies in **class imbalance**—the majority of customers do not default, which can cause models to bias toward the non-default category.

To overcome this, we implement **Synthetic Minority Over-sampling Technique**, which generates synthetic data for the minority class to balance the dataset and enhance model performance. We build and compare two models: **Logistic Regression** and **Random Forest Classifier**. Logistic Regression offers a simple, interpretable solution, while Random Forest uses ensemble methods to handle complex patterns and reduce overfitting. This report covers the steps involved in analyzing the dataset, preprocessing the data, training models, and evaluating their performance. It concludes with observations and suggestions for further improvements.

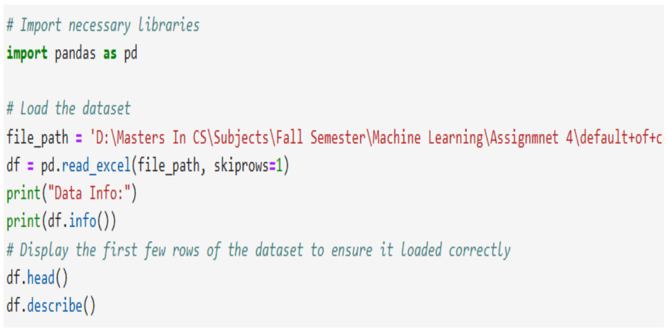
## I. DATASET ANALYSIS & UNDERSTANDING

### 1. Data Characteristics:

Understanding the structure and content of the dataset is the first step toward developing effective predictive models. The dataset used here, **"Default of Credit Card Clients,"** contains a variety of features that represent customer demographics, financial history, and payment behavior. Specifically:

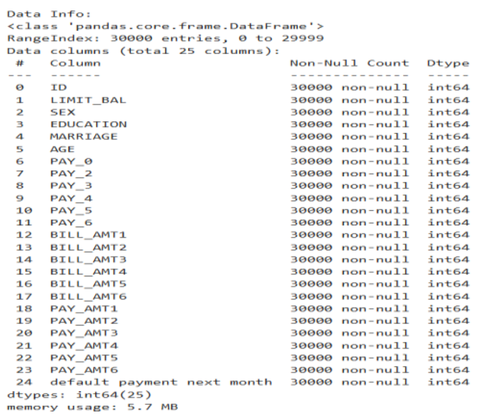
- **Rows:** Each row corresponds to a unique customer.
- **Columns:** Features include demographic details (e.g., age, gender), credit limits, bill amounts, repayment status, and payment history across six months.
- **Target Variable:** The column default payment next month is the target we aim to predict, where:
  - 0 = No Default (Customer makes timely payments)
  - 1 = Default (Customer fails to pay within the required time)

A quick look at the dataset with info() and describe() reveals the data types, ranges, and statistical summaries (mean, median, etc.). This helps us understand the scope of the problem and identify potential issues like missing values or outliers.



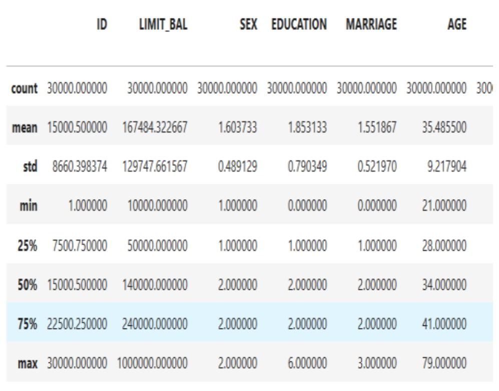
```python
# Import necessary libraries
import pandas as pd

# Load the dataset
file_path = 'D:\Masters In CS\Subjects\Fall Semester\Machine Learning\Assignmnet 4\default+of+c
df = pd.read_excel(file_path, skiprows=1)
print("Data Info:")
print(df.info())
# Display the first few rows of the dataset to ensure it loaded correctly
df.head()
df.describe()
```

*Fig 1: Basic Statistics describing the data and quick look of dataset and their statistics*



```
Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   ID                          30000 non-null  int64
 1   LIMIT_BAL                   30000 non-null  int64
 2   SEX                         30000 non-null  int64
 3   EDUCATION                   30000 non-null  int64
 4   MARRIAGE                    30000 non-null  int64
 5   AGE                         30000 non-null  int64
 6   PAY_0                       30000 non-null  int64
 7   PAY_2                       30000 non-null  int64
 8   PAY_3                       30000 non-null  int64
 9   PAY_4                       30000 non-null  int64
 10  PAY_5                       30000 non-null  int64
 11  PAY_6                       30000 non-null  int64
 12  BILL_AMT1                   30000 non-null  int64
 13  BILL_AMT2                   30000 non-null  int64
 14  BILL_AMT3                   30000 non-null  int64
 15  BILL_AMT4                   30000 non-null  int64
 16  BILL_AMT5                   30000 non-null  int64
 17  BILL_AMT6                   30000 non-null  int64
 18  PAY_AMT1                    30000 non-null  int64
 19  PAY_AMT2                    30000 non-null  int64
 20  PAY_AMT3                    30000 non-null  int64
 21  PAY_AMT4                    30000 non-null  int64
 22  PAY_AMT5                    30000 non-null  int64
 23  PAY_AMT6                    30000 non-null  int64
 24  default payment next month  30000 non-null  int64
dtypes: int64(25)
memory usage: 5.7 MB
```

*Fig 2: Basic Info Output for above code*



|  | ID | LIMIT_BAL | SEX | EDUCATION | MARRIAGE | AGE |  |
|---|---|---|---|---|---|---|---|
| **count** | 30000.000000 | 30000.000000 | 30000.000000 | 30000.000000 | 30000.000000 | 30000.000000 | 300 |
| **mean** | 15000.500000 | 167484.322667 | 1.603733 | 1.853133 | 1.551867 | 35.485500 |  |
| **std** | 8660.398374 | 129747.661567 | 0.489129 | 0.790349 | 0.521970 | 9.217904 |  |
| **min** | 1.000000 | 10000.000000 | 1.000000 | 0.000000 | 0.000000 | 21.000000 |  |
| **25%** | 7500.750000 | 50000.000000 | 1.000000 | 1.000000 | 1.000000 | 28.000000 |  |
| **50%** | 15000.500000 | 140000.000000 | 2.000000 | 2.000000 | 2.000000 | 34.000000 |  |
| **75%** | 22500.250000 | 240000.000000 | 2.000000 | 2.000000 | 2.000000 | 41.000000 |  |
| **max** | 30000.000000 | 1000000.000000 | 2.000000 | 6.000000 | 3.000000 | 79.000000 |  |

*Fig 3: Basic statistics of dataset Output for above code*

## 2. Feature Analysis & Selection

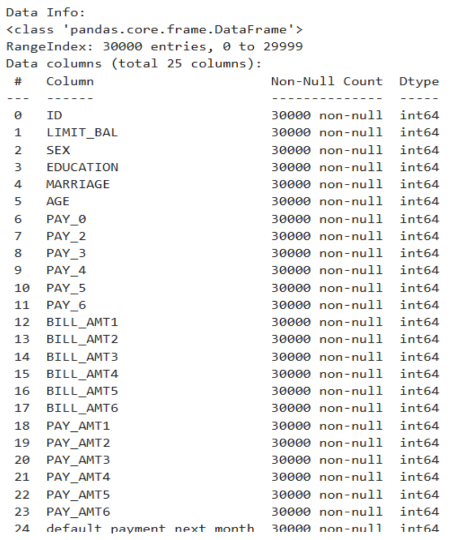
Analyzing each feature helps determine its relevance to the prediction task. Here, features can be categorized into several types:

- **Demographic Features:**
  - **SEX:** Gender (1 = Male, 2 = Female)
  - **AGE:** Customer's age
- **Credit and Payment Features:**
  - **LIMIT_BAL:** Total credit limit assigned to the client.
  - **BILL_AMT1 – BILL_AMT6:** Bill amounts from the previous six months.
  - **PAY_AMT1 –PAY_AMT6:** Payment amounts for each month.
  - **PAY_0 – PAY_6:** Payment status for the previous six months, where -1 indicates early payment, 0 indicates on-time payment, and positive values represent delayed payments.

We also assess the **correlation between features** using a heatmap. Highly correlated features (e.g., bill amounts across months) might carry redundant information and may need to be managed (e.g., through dimensionality reduction).

Feature selection is performed to:

1. Identify important features influencing default behavior.
2. Drop irrelevant or redundant features (e.g., **ID** column).



```
Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   ID                          30000 non-null  int64
 1   LIMIT_BAL                   30000 non-null  int64
 2   SEX                         30000 non-null  int64
 3   EDUCATION                   30000 non-null  int64
 4   MARRIAGE                    30000 non-null  int64
 5   AGE                         30000 non-null  int64
 6   PAY_0                       30000 non-null  int64
 7   PAY_2                       30000 non-null  int64
 8   PAY_3                       30000 non-null  int64
 9   PAY_4                       30000 non-null  int64
 10  PAY_5                       30000 non-null  int64
 11  PAY_6                       30000 non-null  int64
 12  BILL_AMT1                   30000 non-null  int64
 13  BILL_AMT2                   30000 non-null  int64
 14  BILL_AMT3                   30000 non-null  int64
 15  BILL_AMT4                   30000 non-null  int64
 16  BILL_AMT5                   30000 non-null  int64
 17  BILL_AMT6                   30000 non-null  int64
 18  PAY_AMT1                    30000 non-null  int64
 19  PAY_AMT2                    30000 non-null  int64
 20  PAY_AMT3                    30000 non-null  int64
 21  PAY_AMT4                    30000 non-null  int64
 22  PAY_AMT5                    30000 non-null  int64
 23  PAY_AMT6                    30000 non-null  int64
 24  default payment next month  30000 non-null  int64
```

*Figure 4: The features dropped from the original dataset, grouped by criteria.*

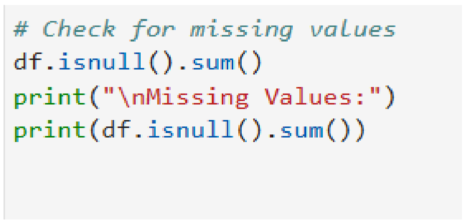
## 3. Data Cleaning / Preprocessing

Cleaning and preprocessing are essential steps to prepare the dataset for machine learning models:

- **Handling Missing Values:** Although this dataset does not have significant missing data, checking for null values ensures we don't overlook any inconsistencies (isnull().sum()).
- **Removing Duplicates:** Duplicate entries, if any, are removed to avoid data leakage.
- **Encoding Categorical Features:**
  - **Label Encoding:** Converts categorical values (e.g., gender) into

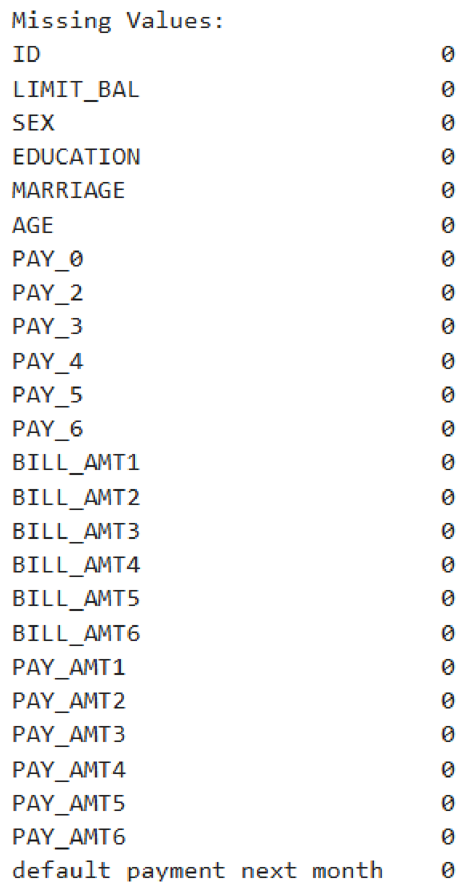
numeric forms for compatibility with machine learning models.

- **Handling Outliers:** Outliers in features like **LIMIT_BAL** or **BILL_AMT** may skew the model's predictions. We either cap or remove such extreme values, if necessary.



```
# Check for missing values
df.isnull().sum()
print("\nMissing Values:")
print(df.isnull().sum())
```

*Fig 5: Checking for Missing Values in dataset*



```
Missing Values:
ID                            0
LIMIT_BAL                     0
SEX                           0
EDUCATION                     0
MARRIAGE                      0
AGE                           0
PAY_0                         0
PAY_2                         0
PAY_3                         0
PAY_4                         0
PAY_5                         0
PAY_6                         0
BILL_AMT1                     0
BILL_AMT2                     0
BILL_AMT3                     0
BILL_AMT4                     0
BILL_AMT5                     0
BILL_AMT6                     0
PAY_AMT1                      0
PAY_AMT2                      0
PAY_AMT3                      0
PAY_AMT4                      0
PAY_AMT5                      0
PAY_AMT6                      0
default payment next month    0
```

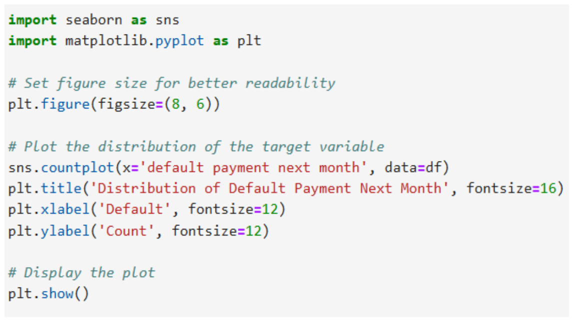
*Fig 6: Output for Missing Values in dataset*

These steps ensure the dataset is consistent, relevant, and ready for further transformation.

## 4. Data Visualization – Independent Features

Visualization allows us to explore trends and patterns within the data, helping to uncover insights and make informed decisions:

- **Target Distribution:** We use a **count plot** to visualize the distribution of the target variable (default payment next month). This step helps us understand the severity of class imbalance.
- **Histograms:** Plots of numerical features (e.g., **LIMIT_BAL**, **AGE**) give insight into their distribution (normal, skewed, etc.) and reveal outliers.
- **Boxplots:**
  Boxplots visualize the spread of bill and payment amounts, helping detect potential outliers.
- **Correlation Heatmap:** We generate a **heatmap** to show correlations between numeric features (e.g., **BILL_AMT** and **PAY_AMT** values over time). This helps in identifying multicollinearity—if two or more features are highly correlated, we might remove or combine them to avoid redundancy.



```
import seaborn as sns
import matplotlib.pyplot as plt

# Set figure size for better readability
plt.figure(figsize=(8, 6))

# Plot the distribution of the target variable
sns.countplot(x='default payment next month', data=df)
plt.title('Distribution of Default Payment Next Month', fontsize=16)
plt.xlabel('Default', fontsize=12)
plt.ylabel('Count', fontsize=12)

# Display the plot
plt.show()
```

*Fig 7: Distribution of Default Payment Next Month*

- This code generates a bar plot to show the count of clients who did and did not default in the default payment next month column of df, allowing visualization of class distribution in the target variable.

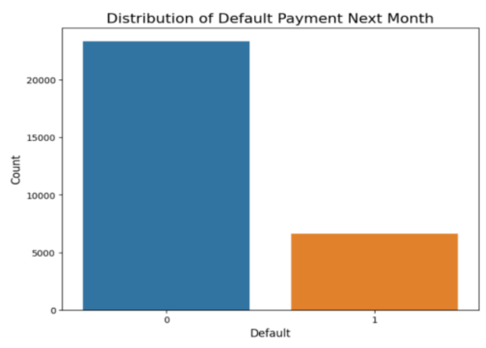
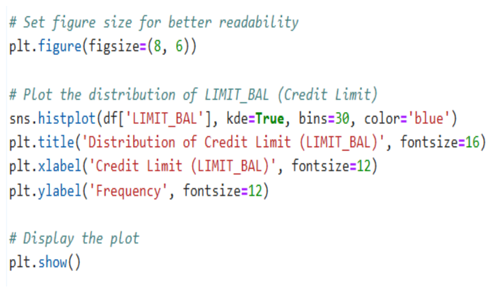


*Fig 8: Output for Distribution of Default Payment Next Month.*



```
# Set figure size for better readability
plt.figure(figsize=(8, 6))

# Plot the distribution of LIMIT_BAL (Credit Limit)
sns.histplot(df['LIMIT_BAL'], kde=True, bins=30, color='blue')
plt.title('Distribution of Credit Limit (LIMIT_BAL)', fontsize=16)
plt.xlabel('Credit Limit (LIMIT_BAL)', fontsize=12)
plt.ylabel('Frequency', fontsize=12)

# Display the plot
plt.show()
```

*Fig 9: Distribution of Credit Limit (LIMIT _BAL)*

- This code creates a histogram with a density curve to show the distribution of credit limits (LIMIT_BAL) in df, helping visualize the spread and frequency of different credit limit values and the below figure is the output.

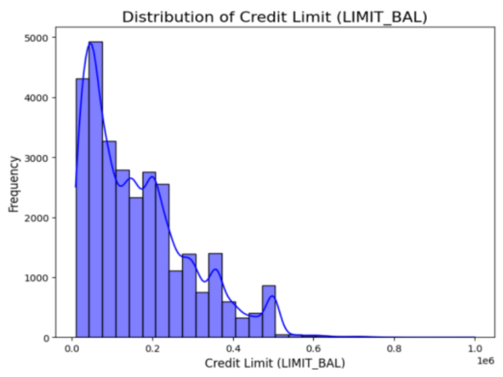
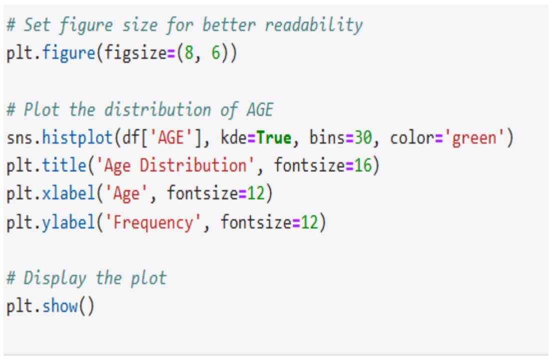


*Fig 10: Output for Distribution of Credit Limit (LIMIT _BAL)*



```
# Set figure size for better readability
plt.figure(figsize=(8, 6))

# Plot the distribution of AGE
sns.histplot(df['AGE'], kde=True, bins=30, color='green')
plt.title('Age Distribution', fontsize=16)
plt.xlabel('Age', fontsize=12)
plt.ylabel('Frequency', fontsize=12)

# Display the plot
plt.show()
```

*Figure 11: Age Distribution*

- This code creates a histogram with a density curve to show the distribution of AGE in df, highlighting the frequency and spread of ages in the dataset and below figure is the output.

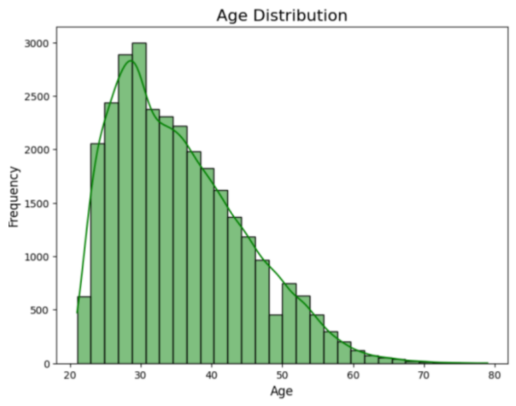
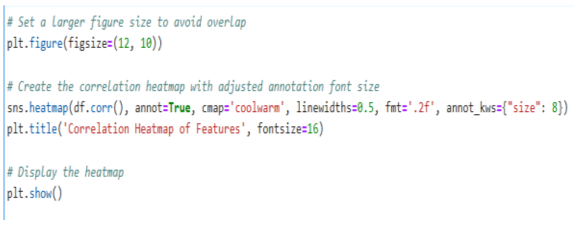


*Fig 12: Output for Age Distribution*



```
# Set a larger figure size to avoid overlap
plt.figure(figsize=(12, 10))

# Create the correlation heatmap with adjusted annotation font size
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', linewidths=0.5, fmt='.2f', annot_kws={"size": 8})
plt.title('Correlation Heatmap of Features', fontsize=16)

# Display the heatmap
plt.show()
```

*Fig 13: Correlation Heatmap of Features*

- This code generates a heatmap to display correlations between features in df, with annotations, colors indicating correlation strength, and a larger figure size for clarity.

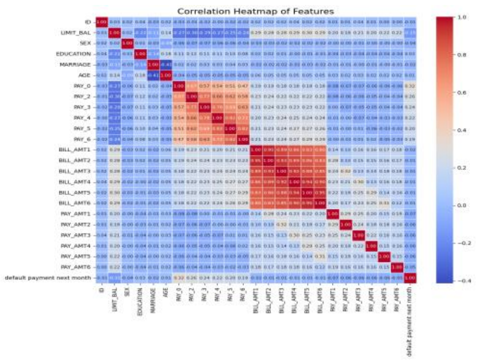


*Fig 14: Output for Correlation Heatmap of Features*

## II. Data Transformation & Models Used

The two models discussed in this section are the logistic regression classifier and random forest ensemble classifier. In order to train the models to ensure consistent, meaningful, and accurate predictions, several preliminary measures were taken in general and specifically for each model in order to prepare and transform into a format which could be easily used for training and prediction.

### 1. Feature Scaling

Feature scaling ensures that all features contribute equally to the model, especially those with different units or magnitudes.

- **StandardScaler** from sklearn.preprocessing was applied to transform the dataset.
    - This scaling process converts each feature to have a **mean of 0** and a **standard deviation of 1**, ensuring that all numeric features are on the same scale.
    - Scaling is particularly important for **Logistic Regression**, which assumes that all features have similar scales. Without scaling, the model could give more weight to features with larger values, leading to skewed predictions.

Example:

- **LIMIT_BAL** (credit limit) values range from thousands to hundreds of thousands.
- **AGE** ranges only from 20 to 79 years.
  Without scaling, the **credit limit** feature could dominate the **age** feature simply because of its larger magnitude. StandardScaler addresses this issue by normalizing the data.

### 2. One-hot Encoding for Logistic Regression

Logistic Regression requires all inputs to be numeric, and it cannot directly interpret categorical data. To convert categorical variables like **SEX**, **MARRIAGE**, and **EDUCATION** into numerical form, we applied **one-hot encoding**.

- **One-hot encoding** creates a new binary feature for each category. For example:
    - **SEX**: Male becomes [1, 0], and Female becomes [0, 1].
    - **EDUCATION**: Values are split into multiple binary columns (e.g., Graduate School, University, High School).

This technique ensures that the model treats categories as independent, avoiding implicit ordinal relationships (e.g., encoding male as 1 and female as 2 might imply a rank).
While **one-hot encoding** increases the dimensionality of the data, it works well with models like Logistic Regression that benefit from linear relationships.

### 3. Integer Label Encoding for Random Forest Classifier

While **one-hot encoding** works well for linear models like Logistic Regression, it can increase the complexity of the dataset for tree-based models. Instead, for **Random Forest**, we used **label encoding** to convert categorical features into integer values.

- **Label encoding** assigns a unique integer to each category. For example:
    - **SEX**: Male = 1, Female = 2.
    - **MARRIAGE**: Single = 1, Married = 2, Others = 3.

This encoding is suitable for Random Forest because decision trees can naturally split data based on feature values without assuming any inherent relationship between encoded values. The model can handle categorical splits efficiently by learning the best thresholds during training.

### 4. Logistic Regression Classifier

**Logistic Regression** is a simple, yet effective linear model used for binary classification tasks. In this project, we use Logistic Regression to predict whether a customer will **default** (1) or **not default** (0) on their credit card payment.

- Logistic Regression models the relationship between input features and the log-odds of the target class. The **Sigmoid function** transforms the output into a probability value between 0 and 1, with a threshold (typically 0.5) used to assign the final class label.

**Why use Logistic Regression?**

- **Interpretability:** The coefficients of the model indicate the importance and impact of each feature.
- **Simplicity:** It is easy to implement and works well when the relationship between features and the target is linear.

- **Regularization:** The model can include **L1 or L2 regularization** to prevent overfitting.

**Evaluation using Cross-Validation:**

We used **5-fold cross-validation** to evaluate the performance of the Logistic Regression model. Cross-validation splits the dataset into 5 parts and iteratively trains the model on 4 parts while testing on the remaining 1 part.
This process ensures a more robust evaluation by mitigating the effect of overfitting or underfitting to a specific data split.

**5. Random Forest Ensemble Classifier**

**Random Forest** is an ensemble model that builds multiple decision trees and aggregates their predictions to produce a final output. It is well-suited for both **linear and non-linear data** and can capture complex interactions between features.

- Each tree in the forest is trained on a random subset of the data, and at each split, only a subset of features is considered. This **randomization** helps reduce over-fitting and ensures that no single feature dominates the predictions.

**Why use Random Forest?**

- **Handling Non-linear Relationships:** Random Forests can capture complex, non-linear interactions between features.
- **Robustness:** They are less prone to over-fitting compared to individual decision trees.
- **Feature Importance:** Random Forest provides an estimate of feature importance, helping us understand which features are most relevant for predicting defaults.

**Evaluation using Cross-Validation:**

Similar to Logistic Regression, we evaluated the Random Forest using **5-fold cross-validation** to assess its performance on different subsets of the data.

**6. Handling the Data Imbalance with SMOTE**

In our dataset, the **class imbalance** problem arises because the number of **non-default cases** far exceeds the number of **default cases**. This imbalance can bias the model towards predicting the majority class (non-default), resulting in poor generalization for minority class instances.

To address this issue, we used **SMOTE** (Synthetic Minority Oversampling Technique):

- **SMOTE** generates synthetic samples for the minority class by interpolating between existing instances. It selects two or more neighboring points in the feature space and creates new synthetic points along the line connecting them.

**Why use SMOTE?**

- **Balanced Data:** It ensures the model is exposed to enough instances of both classes (default and non-default), helping it learn better.
- **Improved Generalization:** The model becomes more reliable at predicting rare events, such as defaults.
- **Alternative to Under-sampling:** SMOTE avoids the loss of information associated with under-sampling the majority class.

By using **SMOTE** on the training data, we ensure that both **Logistic Regression** and **Random Forest** models receive balanced input, improving their ability to identify default cases.

This section highlights the **transformation techniques** and **modeling choices** made to build accurate and reliable credit default prediction models. Feature scaling, encoding strategies, and balancing the data were crucial steps to ensure the models could learn effectively from the available data. The combination of **Logistic Regression** and **Random Forest** allows us to compare the performance of a simple linear model against a more robust ensemble model.

## III. Experiments & Model Results

**1. Logistic Regression Tuning & Evaluation**

**Training Process**

- **Dataset:** The model was trained on the **resampled dataset's** generated using SMOTE to address the imbalance between default and non-default classes.
- **Evaluation Metric:** We used **5-fold cross-validation** to measure the model’s performance. This method divides the data into 5 subsets (folds).

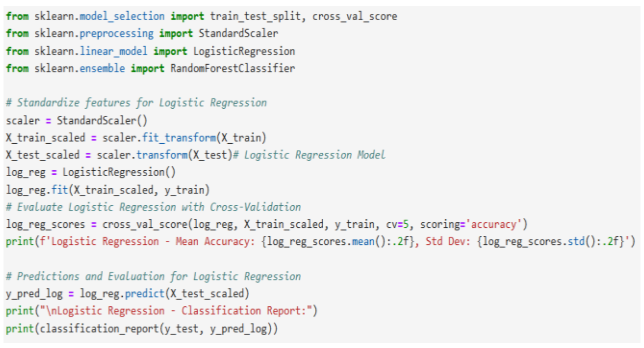
The model is trained on 4 folds and tested on the remaining one, iterating five times with a different test fold each time. The final result is the average performance across all iterations.

**Results and Interpretation**

- **Mean Accuracy:** 0.81
- **Standard Deviation of Accuracy:** 0
- **Interpretation of Accuracy:** Accuracy measures the percentage of correct predictions (both default and non-default). A higher accuracy value indicates that the model can correctly predict whether a client will default or not. However, since the original dataset's is imbalanced (more non-defaults than defaults), accuracy alone may not be the best measure.
- **Bias Toward Majority Class:** Despite resampling with SMOTE, the Logistic Regression model showed **some bias toward the majority class** (non-default). This is because Logistic Regression is a **linear model**, and financial behavior may exhibit **non-linear relationships** between features (such as credit limit and repayment history). This limitation affects the model's ability to distinguish edge cases, such as clients on the verge of default.

**Insights for Tuning Logistic Regression:**

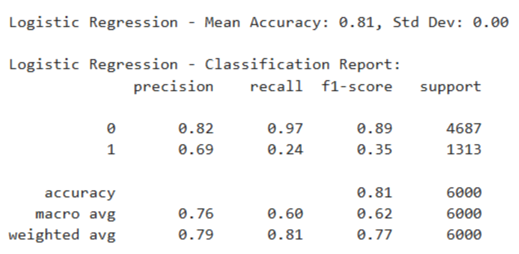
- **Hyper-parameter Tuning:**
  - We could adjust the **regularization strength (C)** to control over-fitting. A higher value of C reduces regularization, fitting the training data more closely.
  - Experiment with **class weights** to assign more importance to the minority class, improving the recall for default cases.
- **Alternative Metrics:**
  - **Precision and Recall:** Recall for the default class is critical since it measures how well the model identifies defaulters.
  - **F1-score:** Balances precision and recall, providing a better measure for imbalanced datasets.



```python
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

# Standardize features for Logistic Regression
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)# Logistic Regression Model
log_reg = LogisticRegression()
log_reg.fit(X_train_scaled, y_train)
# Evaluate Logistic Regression with Cross-Validation
log_reg_scores = cross_val_score(log_reg, X_train_scaled, y_train, cv=5, scoring='accuracy')
print(f'Logistic Regression - Mean Accuracy: {log_reg_scores.mean():.2f}, Std Dev: {log_reg_scores.std():.2f}')

# Predictions and Evaluation for Logistic Regression
y_pred_log = log_reg.predict(X_test_scaled)
print("\nLogistic Regression - Classification Report:")
print(classification_report(y_test, y_pred_log))
```

*Fig 15: Logistic Regression classification report*

- This code scales features, trains a Logistic Regression model on the scaled training data, evaluates it with 5-fold cross-validation for accuracy, and outputs mean accuracy, standard deviation, and a classification report for predictions on the test set.



```
Logistic Regression - Mean Accuracy: 0.81, Std Dev: 0.00

Logistic Regression - Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.97      0.89      4687
           1       0.69      0.24      0.35      1313

    accuracy                           0.81      6000
   macro avg       0.76      0.60      0.62      6000
weighted avg       0.79      0.81      0.77      6000
```

*Fig 16: Output for Logistic Regression classification report*

**2. Random Forest Tuning & Evaluation**

**Training Process**

- **Dataset:** The Random Forest model was trained on the same **SMOTE-resampled data** to balance the number of default and non-default cases.
- **Why Random Forest Works Better:** Random Forest combines multiple **decision trees** to form an ensemble model, handling **non-linear relationships** between features more effectively than Logistic Regression. Each decision tree sees a random subset of data and features, preventing any single tree from dominating the predictions.

**Results and Interpretation**

- **Mean Accuracy:** 0.81
- **Standard Deviation of Accuracy:** 0

- **Better Performance Than Logistic Regression:**
  Random Forest achieved slightly higher accuracy than Logistic Regression. The ensemble nature of the model allows it to generalize better and reduces the risk of over-fitting.
- **Handling Non-linear Data:**
  Random Forest can effectively capture complex interactions between features, such as how the **repayment history** and **credit limit** together influence the likelihood of default. This flexibility makes it a more powerful model for datasets with complicated patterns.

**Insights for Tuning Random Forest:**

- **Hyperparameters for Tuning:**

  - **Number of Trees (n_estimators):** Increasing the number of trees can improve accuracy but at the cost of longer training times.
  - **Max Depth of Trees:** Controlling the depth prevents over-fitting by limiting how complex each decision tree can become.
  - **Minimum Samples per Leaf:** This parameter ensures that each leaf node (end of a tree) contains enough data points, improving generalization.

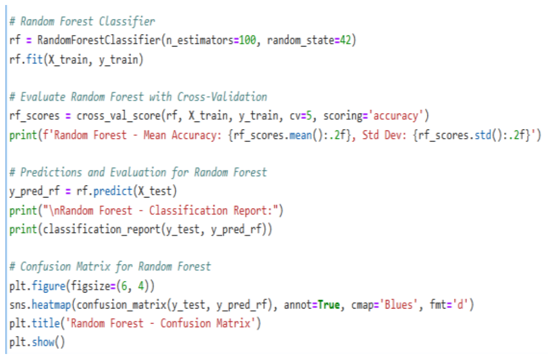
- **Feature Importance:**
  Random Forest provides **feature importance scores**, allowing us to identify which features contribute the most to predictions. This insight can be used to simplify the model by eliminating irrelevant features.

**Alternative Metrics for Evaluation:**

- **Precision and Recall for Default Cases:**

  - **Precision:** The proportion of true default predictions among all instances predicted as default. A high precision ensures fewer false alarms.
  - **Recall:** The proportion of actual defaults correctly predicted by the model. This is crucial in financial applications, where missing a default case can have severe consequences.

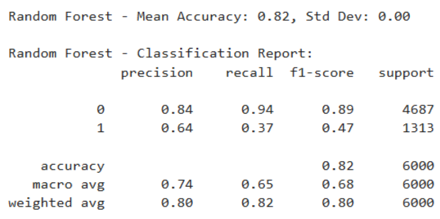
- **ROC-AUC (Receiver Operating Characteristic - Area Under Curve):**
  The AUC score measures how well the model separates the two classes (default vs. non-default) across different thresholds. A higher AUC indicates a better-performing model.



```python
# Random Forest Classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Evaluate Random Forest with Cross-Validation
rf_scores = cross_val_score(rf, X_train, y_train, cv=5, scoring='accuracy')
print(f'Random Forest - Mean Accuracy: {rf_scores.mean():.2f}, Std Dev: {rf_scores.std():.2f}')

# Predictions and Evaluation for Random Forest
y_pred_rf = rf.predict(X_test)
print("\nRandom Forest - Classification Report:")
print(classification_report(y_test, y_pred_rf))

# Confusion Matrix for Random Forest
plt.figure(figsize=(6, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, cmap='Blues', fmt='d')
plt.title('Random Forest - Confusion Matrix')
plt.show()
```

*Fig 17: Random Forest Parameter Tuning*

- This code trains a Random Forest model on the training data, evaluates it using 5-fold cross-validation, prints mean accuracy and standard deviation, generates a classification report for test predictions, and displays a confusion matrix heatmap.



```
Random Forest - Mean Accuracy: 0.82, Std Dev: 0.00

Random Forest - Classification Report:
              precision    recall  f1-score   support

           0       0.84      0.94      0.89      4687
           1       0.64      0.37      0.47      1313

    accuracy                           0.82      6000
   macro avg       0.74      0.65      0.68      6000
weighted avg       0.80      0.82      0.80      6000
```

*Fig 17: Random Forest Parameter Tuning*

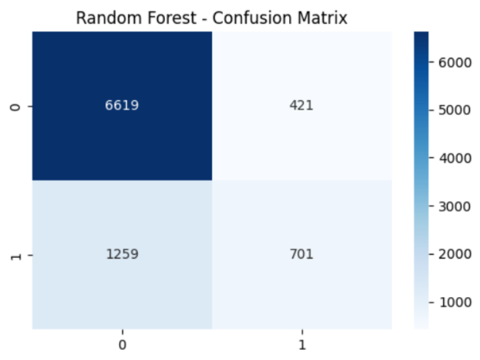


*Figure 10: Random Forest - Confusion Matrix*

**Comparison of Models:**

| Metric | Logistic Regression | Random Forest |
|---|---|---|
| **Mean Accuracy** | 0.81 | 0.82 |
| **Standard Deviation** | 0 | 0 |
| **Handling Non-linearity** | Poor | Good |
| **Interpretability** | High | Moderate |
| **Training Time** | Fast | Slower (with large trees) |
| **Risk of Overfitting** | Moderate | Low |

**Summary of Results:**

- **Logistic Regression** is fast, interpretable, and easy to implement, but it struggled to capture complex patterns in the data. It performed reasonably well, but it showed some bias toward the majority class (non-default).
- **Random Forest**, on the other hand, provided better accuracy and generalization. Its ability to handle non-linear relationships and complex interactions between features gave it an edge over Logistic Regression. However, it is less interpretable and takes longer to train, especially with large datasets.

**Conclusion from Experiments:**

- **Random Forest** is the preferred model for this task, given its superior performance and ability to handle complex data. However, **Logistic Regression** can still be useful when simplicity and interpret-ability are important.
- **Future Work:** Further improvements could involve hyperparameter tuning, feature selection, and testing additional models such as **XGBoost** or **Gradient Boosting** to see if they offer further performance gains. Using **ensemble techniques** (e.g., stacking) that combine the strengths of multiple models could also enhance prediction accuracy.

This section provides a thorough comparison of the models, the rationale for their performance, and actionable steps to improve future results.

## IV. Conclusion

**Lessons Learned**

1. **Class Imbalance Matters:** Models tend to favor the majority class, leading to poor recognition of defaults. SMOTE helped improve predictions by balancing the dataset with synthetic minority samples.
2. **Feature Scaling:** Logistic Regression benefited from scaling to standardize feature values, while Random Forest performed well without scaling due to its tree-based nature.
3. **Handling Correlation:** Highly correlated features, like bill amounts across months, can introduce redundancy and affect linear models. Removing multicollinear features or using PCA could improve future performance.

**Mistakes Made, Challenges, & Future Considerations**

1. **Data Scaling Errors:** Initially, **unscaled test data** caused poor model performance. Applying consistent scaling to both training and testing sets fixed the issue.
2. **Encoding Challenges:** Incorrect encoding strategies led to errors. **One-hot encoding** for Logistic Regression and **integer encoding** for Random Forest solved the problem.
3. **Future Work:**
    a) Explore advanced models like **XGBoost** and **Neural Networks** for better predictions.
    b) Use **hyperparameter tuning** and **ensemble techniques** to enhance accuracy.
    c) Balance interpretability with predictive power to ensure actionable insights for financial decisions.

This project underscores the importance of **effective preprocessing, model selection, and proper handling of imbalances** for reliable credit risk prediction.

**APPENDIX A**

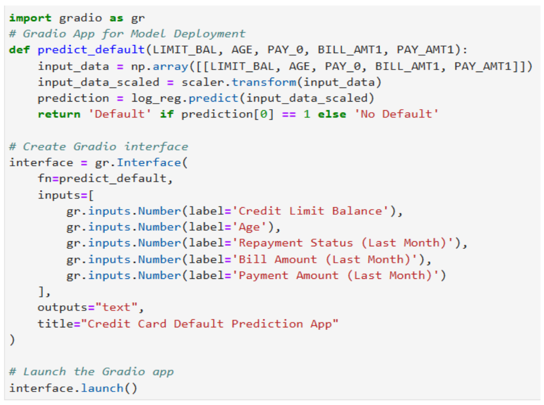
**Gradio App Usage**

We implemented a **Gradio web interface** to provide an interactive platform for credit card default predictions.

**Instructions:**

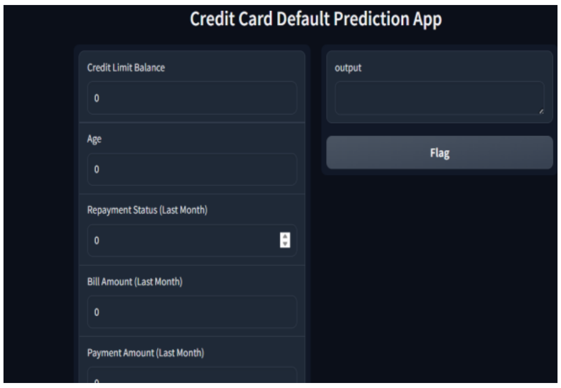
1. **Enter the Client Data:** Fill in all the required fields with relevant client information, such as credit limit, bill amounts, repayment status, and payment amounts.
2. **Submit the Data:** Click the **"Submit"** button to generate the prediction. The app will indicate whether the client is likely to **default** or **not default** on their payment.
3. **Error Handling:** If any field is missing or the input data is invalid (e.g., non-numeric values), an **error message** will be shown, prompting the user to correct the input.

This interface allows for easy testing of the prediction model, providing real-time results and enhancing usability for non-technical users.



```python
import gradio as gr
# Gradio App for Model Deployment
def predict_default(LIMIT_BAL, AGE, PAY_0, BILL_AMT1, PAY_AMT1):
    input_data = np.array([[LIMIT_BAL, AGE, PAY_0, BILL_AMT1, PAY_AMT1]])
    input_data_scaled = scaler.transform(input_data)
    prediction = log_reg.predict(input_data_scaled)
    return 'Default' if prediction[0] == 1 else 'No Default'

# Create Gradio interface
interface = gr.Interface(
    fn=predict_default,
    inputs=[
        gr.inputs.Number(label='Credit Limit Balance'),
        gr.inputs.Number(label='Age'),
        gr.inputs.Number(label='Repayment Status (Last Month)'),
        gr.inputs.Number(label='Bill Amount (Last Month)'),
        gr.inputs.Number(label='Payment Amount (Last Month)')
    ],
    outputs="text",
    title="Credit Card Default Prediction App"
)

# Launch the Gradio app
interface.launch()
```

*Fig 11: Code for Gradio app interface*



Credit Card Default Prediction App

Credit Limit Balance: 0

Age: 0

Repayment Status (Last Month): 0

Bill Amount (Last Month): 0

Payment Amount (Last Month)

output

*Fig 12: Output for Gradio app interface*

**GIT Repository Link:**
https://github.com/vishal9640/Credit-Card-Default-Prediction

**REFERENCES**

1. Yeh, I. C., & Lien, C. H. (2009). UCI Machine Learning Repository: Default of Credit Card Clients Dataset. Retrieved from UCI Repository.
2. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research, 12*, 2825-2830.
3. https://www.gradio.app/docs
4. Python Software Foundation. Python Language Reference, version 3.7. Available at http://www.python.org
5. https://scikit-learn.org/stable/index.html
6. https://pandas.pydata.org/