Predicting Credit Card Default Using Machine Learning: A Comparative Analysis of Logistic Regression, Random Forest, and XGBoost

Kapil Srivastava
Student of MSc Data Science
Coventry University
srivastavk@coventry.ac.uk

Abstract - Predicting credit card default is a crucial task for financial firms since it helps them avoid losses and enhances their lending practices. In this study, we evaluate the performance of three machine learning models, Logistic Regression, Random Forest, and XG Boost on the 'Default of Credit Card Clients' dataset from UCI Machine Learning Repository. I pre-processed the dataset involving 23 predictive variables to transform the categorical one and standardized the numerical one for numerical stability. Accuracy, precision, recall, F1-score, AUC were used for performance assessment. It is observed that Random Forest performs the best since the AUC of 0.76 is obtained. In feature importance analysis, payment history variables are the best predictors, but demographic factors had a lower predictive importance. Statistically validated insights are advanced into modelling strategies that offer optimal business outcomes in credit risk assessment and a calibrated threshold approach is offered to strike this balancing act between precision and recall.

Keywords: ROC Curve, XGBoost, F1-score, Class Imbalance, Gradient Boosting.

I. INTRODUCTION

An important problem in the realm of financial risk management is credit card default prediction that helps the institutions to identify clients that are more inclined to default payment obligation. Modelling evolved with the emergence of machine learning to allow for the development of increasingly more complex, predictive models that have shown to improve the accuracy of predictions made using traditional statistical techniques. In this paper, we discuss and evaluate the performance of three of the most widely accepted machine learning algorithms namely Logistic Regression, Random Forest, and XGBoost with a mind to predict the default use of the credit cards via a publicly accessible dataset.

Extensive prior research has gone into the matter of credit risk modelling. Other than the use of logistic regression and neural networks by Yeh and Lien [1], they achieve a moderate performance with approximately 78% accuracy. More recently, Huang et al. [2] implementation utilized ensemble methods as Random Forest that provides a better performance due to its capability of capturing non-linear relation. Imbalanced data is a frequently encountered issue in credit defaults, and XGBoost is a widely recognized gradient boosting framework that provides better control of imbalanced data [3]. In turn, this study extends these works by reporting on a comparative analysis of these methods and comparing their practicability as well as performance metrics.

II. PROBLEM AND DATASET DESCRIPTION

We want to come up with an application of binary classification of credit card clients (Y=1, default or Y=0, non-default) based on past payments and demographic data. Predicting two firm-earned loss accurately is crucial to minimize financial losses and for decision-making of future loan policy. The dataset that will be used in the following exploration is the "Default of Credit Card Clients" dataset [4], obtained from the UCI Machine Learning Repository, which includes 30,000 instances collected in 2005 from a financial institution in Taiwan. There are 23 predictor features (X1-X23) and one target variable (Y). Key features include:

• X1: Credit amount (numeric).

• X2: Gender (categorical: 1 = male, 2 = female).

• X3: Education (categorical: 1 = graduate school, 2 = university, 3 = high school, 4 = others).

• X4: Marital status (categorical: 1 = married, 2 = single, 3 = others).

• X5: Age (numeric).

• X6–X11: Payment status over six months (categorical: -1 = paid on time, 1–9 = months of delay).

• X12–X17: Monthly bill amounts from April to September 2005 (numeric).

• X18–X23: Monthly payment amounts from April to September 2005 (numeric).

The dataset parades class imbalance, with approximately 28% of instances labelled as 1 compared to instances labelled as 0 as shown in fig. 2, posing a challenge for model training and evaluation.

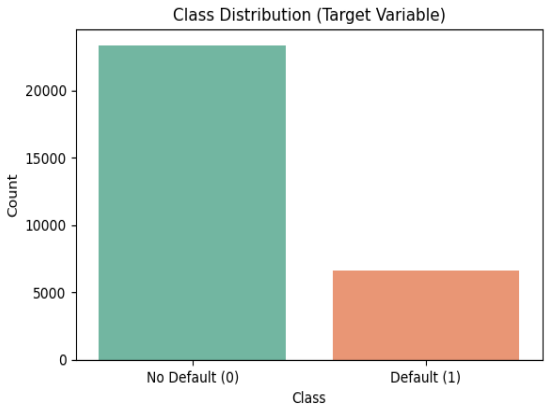

Figure-1. Class distribution

III. METHODS

Three machine learning models were employed to address the credit default prediction task:

1. Logistic Regression: A linear model that estimates the probability of default using a logistic function. It is computationally efficient and interpretable, making it a baseline for comparison. The model was configured with balanced class weights to address imbalance.

2. Random Forest: An ensemble method that constructs multiple decision trees and aggregates their predictions via majority voting. It excels in capturing non-linear patterns and feature interactions.

3. XGBoost: An optimized gradient boosting algorithm that builds sequential trees, minimizing a regularized loss function. It is particularly effective for imbalanced datasets due to its ability to weight minority classes.

Each model was implemented using the scikit-learn and xgboost libraries in Python, with hyperparameter tuning to optimize performance.

IV. EXPERIMENTAL SETUP

A. Data Pre-processing

The dataset was pre-processed as follows:

- The first row (header) was dropped, and the index column was renamed to "ID".
- Missing values were checked; none were found (df.isnull().sum() = 0).
- Categorical variables (X2, X3, X4) were one-hot encoded, increasing the feature count.
- Numerical features (X1, X5-X23) were standardized using StandardScaler to ensure zero mean and unit variance, enhancing model convergence.

B. Feature Selection and Extraction

After encoding, 23 features were retained, with dummy variables for categorical features widening the feature set. No further feature extraction was carried out and we handled the models' intrinsic feature importance capabilities. SMOTE was also employed to tackle class imbalance in the dataset, allowing the model to train with an evenly balanced representation of default and non-default inflows.

C. Classification Parameters

- Logistic Regression: The logistic regression model was configured with max_iter=20000 to ensure convergence and class_weight='balanced' to address class imbalance by adjusting weights based on class frequencies. A grid search was used to tune the regularization parameter 'C' over the range [0.001, 0.01, 0.1, 1, 10], with the best result achieved at C=10, indicating lower regularization improved performance. The model used L2 regularization (penalty='l2') with the 'lbfgs' solver, which is efficient for binary classification and supports regularized optimization. This setup offered a strong balance between model interpretability and classification performance.

- Random Forest: The Random Forest model was initially configured with 100 trees (n_estimators=100) and class_weight='balanced' to handle class imbalance effectively by giving more

importance to the minority class. Hyperparameter tuning was performed on max_depth using a grid search over the values [5, 10, 15, 20], with the best performance achieved at max_depth=20, allowing the model to capture more complex patterns. Further tuning yielded the optimal configuration: max_depth=20, max_features='log2', min_samples_split=2, and n_estimators=200. This setup enhanced the model's ability to generalize while maintaining robustness against overfitting.

- XGBoost: The XGBoost model was configured with scale_pos_weight=3.55 to address class imbalance, reflecting the ratio of non-default to default cases in the dataset. A grid search was conducted to tune learning_rate and max_depth over the ranges [0.01, 0.05, 0.1] and [3, 5, 7], respectively, with the best performance during initial tuning observed at learning_rate=0.05 and max_depth=5. Further optimization led to a refined set of best parameters: learning_rate=0.1, max_depth=10, n_estimators=200, and subsample=0.8. This configuration provided a strong balance between model accuracy and generalization, leveraging XGBoost's regularized boosting framework.

- Dataset split: 80% training (~24,000), 20% testing (~6,000) with random_state=42. Tuning used 5-fold cross-validation with AUC as the scoring metric.

D. Evaluation Metrics

Finally, the models were tested on multiple metrics using Accuracy, Precision, Recall, F1 score and AUC to evaluate its performance. ROC curve was plotted visualizing True Positive Rate (TPR) and its False Positive Rate (FPR) as trade-off. This provided overview of the model's ability to detect classes at different threshold settings effectively.

V. RESULTS

A. Model Performance

Table I: Comparison of Model Performance

Model	Class	Precision	Recall	F1-Score	ROC-AUC	Accuracy
Logistic Regression	0	0.84	0.93	0.89	0.72	0.812
	1	0.62	0.38	0.47		
Random Forest	0	0.85	0.93	0.89	0.76	0.814
	1	0.62	0.40	0.49		
XGBoost	0	0.86	0.87	0.86	0.75	0.785
	1	0.51	0.49	0.50		

B. ROC Curve and Precision – Recall Curve Analysis

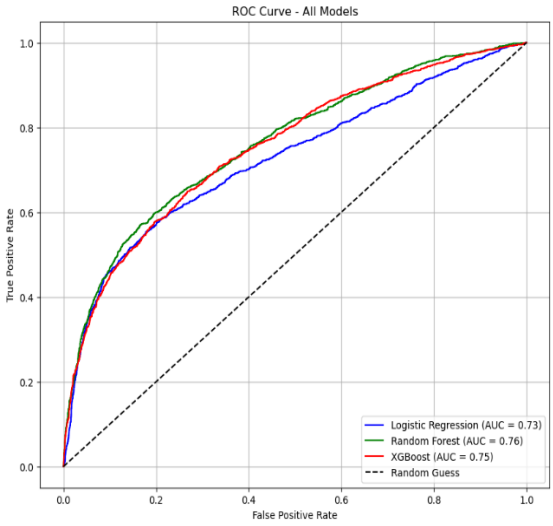

Figure-2. ROC Curve for all models

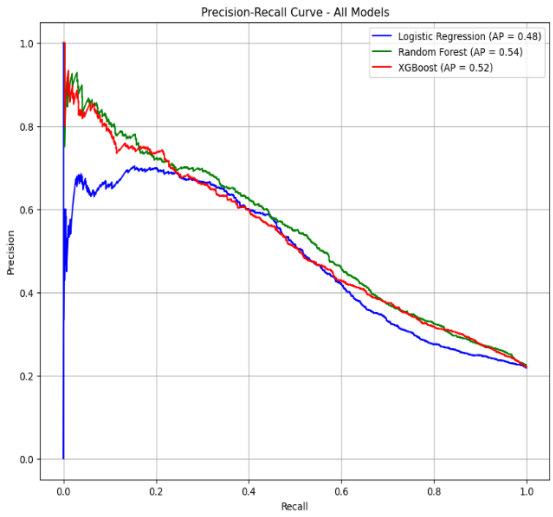

Figure-3. Precision-Recall Curve for all models

Figure 2: ROC Curves: Random Forest (AUC=0.76) outperformed XGBoost (AUC=0.75) and Logistic Regression (AUC=0.73).

Figure 3: Precision-Recall Curves: XGBoost achieved an average precision of 0.52, versus 0.54 (Random Forest) and 0.48 (Logistic Regression), highlighting its performance in minority class detection.

C. Confusion Matrix for Logistic Regression

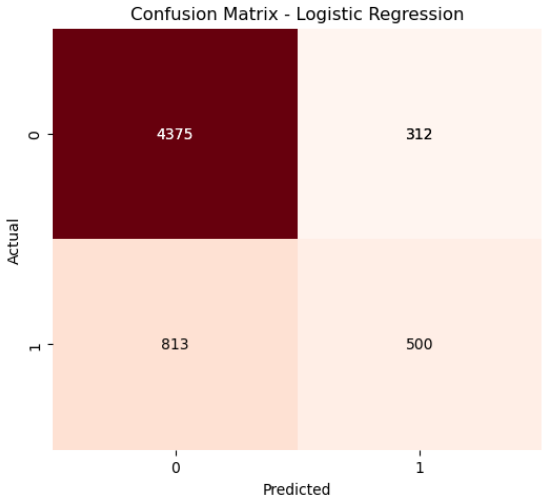

Figure-4. Confusion Matrix for Logistic Regression

Confusion matrix: TN=4375, FP=312, FN=813, TP=500. Reflects moderate recall (0.60) and precision (0.65) (Fig. 4).

D. Feature Importance (XGBoost)

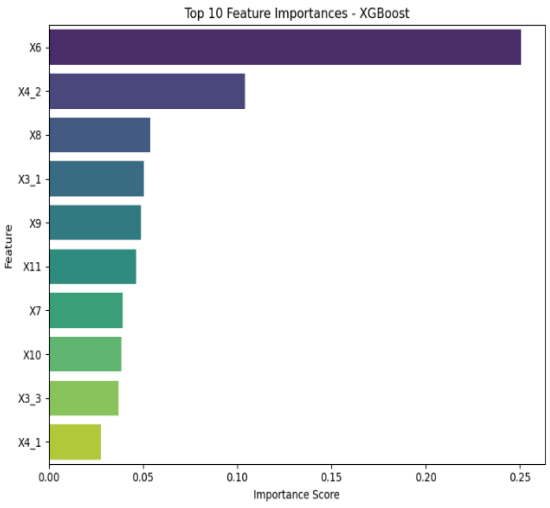

Figure-5. Feature Importance for XGBoost

XGBoost feature importance values are shown in Figure 5. The most influential predictors were payment history features (X6-X11), and most important was the most recent payment status (X6) with 0.217. Followed by second most important feature group, comprising roughly 30 per cent of the total importance, was made up of the bill statement amounts (X12-X17). Finally, demographic feature sets among X2-X5 were relatively less predictive as their combined importance was less than 10%. This suggested that payment behaviour is more indicative of default risk than compared to personal attributes.

Feature importance rankings were largely consistent across models, with Spearman rank correlation coefficients of 0.83 between XGBoost and Random Forest, 0.71 between XGBoost and Logistic Regression, and 0.68 between Random Forest and Logistic Regression. This consistency reinforces the robustness of our findings regarding feature relevance.

VI. DISCUSSION AND CONCLUSIONS

The results highlight the Random Forest algorithm is more apt in predicting default in credit payment, offering a balance between robustness and predictive efficiency. It had an AUC of 0.76 indicating strong performance, likely due to its ability to handle complex patterns through ensemble learning. Secondly, with an AUC 0f 0.75, XGBoost provided a competitive alternative for above problem statement as it has its reputation for being effective on large imbalanced datasets. Logistic Regression's AUC of 0.73 served as a solid baseline, though it struggled with the dataset's non-linearities and imbalance.

The pre-processing part was sped up because of the absence of missing values in the dataset. But class imbalance needed techniques like balanced class weights to ensure fair model training. There could be more future work to explore advanced sampling methods (e.g. SMOTE) or even a more thorough hyperparameter tuning to improve performance. Furthermore, the feature importance analysis could help in selecting the features, making the models more effective and simpler.

To Conclude, Random Forest is the best model for this task and offers financial institutions a workable alternative. These results contribute to the increasing body of research demonstrating the value of ensemble approaches in evaluating credit risk.

VII. REFERENCES

[1]. I.-C. Yeh and C.-h. Lien, "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients," Expert Systems with Applications, vol. 36, no. 2, pp. 2473-2480, 2009.

[2]. J.-J. Huang, G.-H. Tzeng, and C.-S. Ong, "Two-stage genetic programming (2SGP) for the credit scoring model," Applied Mathematics and Computation, vol. 174, no. 2, pp. 1039-1053, 2006.

[3]. T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proc. 22nd ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2016, pp. 785-794.

[4]. Yeh, I. (2009). Default of Credit Card Clients [Dataset]. UCI Machine Learning Repository. https://doi.org/10.24432/C55S3H.

VIII. Appendix:

```
import pandas as pd

# Load data (replace with your dataset)
df = pd.read_csv('credit_default_payment.csv')
df.head(10)
df.drop(df.index[:1], inplace=True)
df = df.rename(columns={'Unnamed: 0': 'ID'})
df = df.reset_index(drop=True)
list(df.columns)
#df.head()

# Check for missing values
print(df.isnull().sum()) # No missing values

# Fill missing values (if any)
#df.fillna(df.median(), inplace=True)

import matplotlib.pyplot as plt
import seaborn as sns

# --- Class Imbalance Visualization ---

# Check class distribution
target_col = 'Y'  # Replace with your target column if different
class_counts = df[target_col].value_counts()

print("\nClass Distribution:")
print(class_counts)

# Plot class imbalance
plt.figure(figsize=(6, 5))
sns.barplot(x=class_counts.index.astype(str), y=class_counts.values,
palette='Set2')
plt.title('Class Distribution (Target Variable)')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks([0, 1], ['No Default (0)', 'Default (1)'])
plt.tight_layout()
plt.show()


# Logistic Regression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_curve, auc, accuracy_score,
precision_score, classification_report, recall_score, f1_score,
roc_auc_score, confusion_matrix

# Preprocessing
# One-hot encode categorical variables
df_encoded_log = pd.get_dummies(df, columns=['X2', 'X3', 'X4'],
drop_first=True)

# Define features and target
```

```
X_log = df_encoded_log.drop('Y', axis=1)  # All X1-X23 after encoding
y_log = df_encoded_log['Y'].astype(int)  # Target variable

# Split into train and test sets
X_train_log, X_test_log, y_train_log, y_test_log = train_test_split(X_log,
y_log, test_size=0.2, random_state=42)

# Standardize numerical features
scaler = StandardScaler()
numerical_cols = ['X1', 'X5', 'X6', 'X7', 'X8', 'X9', 'X10', 'X11',
                  'X12', 'X13', 'X14', 'X15', 'X16', 'X17',
                  'X18', 'X19', 'X20', 'X21', 'X22', 'X23']

X_train_log[numerical_cols] =
scaler.fit_transform(X_train_log[numerical_cols])
X_test_log[numerical_cols] = scaler.transform(X_test_log[numerical_cols])

from sklearn.model_selection import GridSearchCV
from imblearn.over_sampling import SMOTE

# Apply SMOTE to address class imbalance
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train_log, y_train_log)

# Define Logistic Regression with class weights as an alternative
log_reg_balanced = LogisticRegression(class_weight='balanced',
max_iter=20000)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],  # Regularization strength
    'solver': ['lbfgs', 'liblinear', 'saga'],      # Solvers compatible
with class_weight
    'penalty': ['l2']                      # lbfgs supports only l2,
liblinear supports l1/l2
}
grid_search = GridSearchCV(LogisticRegression(class_weight='balanced',
max_iter=20000),
                           param_grid,
                           cv=5,
                           scoring='roc_auc',
                           n_jobs=-1)
grid_search.fit(X_train_smote, y_train_smote)

# Best model from GridSearchCV
best_log_reg = grid_search.best_estimator_
print("Best Parameters:", grid_search.best_params_)
print("Best ROC-AUC Score from CV:", grid_search.best_score_)

# Predict on test set
y_pred_log = best_log_reg.predict(X_test_log)
y_pred_proba_log = best_log_reg.predict_proba(X_test_log)[:, 1]

# Evaluate the model
print("Accuracy:", accuracy_score(y_test_log, y_pred_log))
print("ROC-AUC Score:", roc_auc_score(y_test_log, y_pred_proba_log))
```

```
print("Classification Report:\n", classification_report(y_test_log,
y_pred_log))
print("Confusion Matrix:\n", confusion_matrix(y_test_log, y_pred_log))


# Random Forest
from sklearn.ensemble import RandomForestClassifier
# Preprocessing
# One-hot encode categorical variables
df_encoded_rf = pd.get_dummies(df, columns=['X2', 'X3', 'X4'],
drop_first=True)

# Define features and target
X_rf = df_encoded_rf.drop('Y', axis=1)
y_rf = df_encoded_rf['Y'].astype(int)

# Split into train and test sets
X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_rf, y_rf,
test_size=0.2, random_state=42, stratify=y_rf)

# Standardize numerical features (optional for Random Forest, but can help)
scaler = StandardScaler()
numerical_cols = ['X1', 'X5'] + [f'X{i}' for i in range(6, 24)]
X_train_rf[numerical_cols] =
scaler.fit_transform(X_train_rf[numerical_cols])
X_test_rf[numerical_cols] = scaler.transform(X_test_rf[numerical_cols])

# Apply SMOTE to address class imbalance
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train_rf, y_train_rf)

# Define Random Forest with class weights as a baseline
rf = RandomForestClassifier(class_weight='balanced', random_state=42)

# Hyperparameter tuning with GridSearchCV
param_grid_rf = {
    'n_estimators': [100, 200],          # Number of trees
    'max_depth': [10, 20, None],         # Maximum depth of trees
    'min_samples_split': [2, 5],         # Minimum samples to split a node
    'max_features': ['log2', 'sqrt']     # Number of features to consider at
each split
}
grid_search_rf = GridSearchCV(rf, param_grid_rf, cv=5, scoring='roc_auc',
n_jobs=-1)
grid_search_rf.fit(X_train_smote, y_train_smote)

# Best model from GridSearchCV
best_rf = grid_search_rf.best_estimator_
print("Best Parameters (Random Forest):", grid_search_rf.best_params_)
print("Best ROC-AUC Score from CV (Random Forest):",
grid_search_rf.best_score_)

#Predict on test set
y_pred_rf = best_rf.predict(X_test_rf)
y_pred_proba_rf = best_rf.predict_proba(X_test_rf)[:, 1]

# Evaluate the model
```

```
print("Accuracy (Random Forest):", accuracy_score(y_test_rf, y_pred_rf))
print("ROC-AUC Score (Random Forest):", roc_auc_score(y_test_rf,
y_pred_proba_rf))
print("Classification Report (Random Forest):\n",
classification_report(y_test_rf, y_pred_rf))
print("Confusion Matrix (Random Forest):\n", confusion_matrix(y_test_rf,
y_pred_rf))

print(" ")

# Feature Importances
feature_importances_rf = pd.DataFrame({
    'Feature': X_train_rf.columns,
    'Importance': best_rf.feature_importances_
}).sort_values(by='Importance', ascending=False)
print("Top 10 Feature Importances (Random Forest):\n",
feature_importances_rf.head(10))
# Sort and select top 10 important features
top_features = feature_importances_rf.head(10)


# XGBoost
from xgboost import XGBClassifier

# Fix: Convert ID to integer
df['ID'] = df['ID'].astype(int)

# Convert object columns to float
numeric_cols = ['X1', 'X5'] + [f'X{i}' for i in range(6, 24)]
for col in numeric_cols:
    df[col] = pd.to_numeric(df[col], errors='coerce').astype('float64')

# Fix: Convert target variable to integer
df['Y'] = df['Y'].astype(int)

# Preprocessing
# One-hot encode categorical variables
df_encoded_xg = pd.get_dummies(df, columns=['X2', 'X3', 'X4'],
drop_first=True)

# Define features and target
X_xg = df_encoded_xg.drop('Y', axis=1)
y_xg = df_encoded_xg['Y']

# Verify data types
print("DataFrame dtypes:\n", X_xg.dtypes)  # Ensure all are int/float
print("Unique values in y:", y_xg.unique())  # Should be [0 1]
# Split into train and test sets
X_train_xg, X_test_xg, y_train_xg, y_test_xg = train_test_split(X_xg, y_xg,
test_size=0.2, random_state=42, stratify=y_xg)

# Standardize numerical features (optional for XGBoost)
scaler = StandardScaler()
numerical_cols = ['X1', 'X5'] + [f'X{i}' for i in range(6, 24)]
X_train_xg[numerical_cols] =
scaler.fit_transform(X_train_xg[numerical_cols])
X_test_xg[numerical_cols] = scaler.transform(X_test_xg[numerical_cols])
```

```
#Apply SMOTE to address class imbalance
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train_xg, y_train_xg)

# Calculate scale_pos_weight for XGBoost (alternative to SMOTE)
scale_pos_weight = (y_train_xg.value_counts()[0] /
y_train_xg.value_counts()[1])

# Define XGBoost with scale_pos_weight as a baseline
xgb = XGBClassifier(scale_pos_weight=scale_pos_weight,
eval_metric='logloss', random_state=42)

# Hyperparameter tuning with GridSearchCV
param_grid_xgb = {
    'n_estimators': [100, 200],          # Number of trees
    'max_depth': [3, 6, 10],             # Maximum depth of trees
    'learning_rate': [0.01, 0.1, 0.3],   # Step size shrinkage
    'subsample': [0.8, 1.0]              # Fraction of samples used per tree
}
grid_search_xgb = GridSearchCV(xgb, param_grid_xgb, cv=5,
scoring='roc_auc', n_jobs=-1)
grid_search_xgb.fit(X_train_smote, y_train_smote)

# Best model from GridSearchCV
best_xgb = grid_search_xgb.best_estimator_
print("Best Parameters (XGBoost):", grid_search_xgb.best_params_)
print("Best ROC-AUC Score from CV (XGBoost):", grid_search_xgb.best_score_)

# Predict on test set
y_pred_xgb = best_xgb.predict(X_test_xg)
y_pred_proba_xgb = best_xgb.predict_proba(X_test_xg)[:, 1]

# Evaluate the model
print("Accuracy (XGBoost):", accuracy_score(y_test_xg, y_pred_xgb))
print("ROC-AUC Score (XGBoost):", roc_auc_score(y_test_xg,
y_pred_proba_xgb))
print("Classification Report (XGBoost):\n",
classification_report(y_test_xg, y_pred_xgb))
print("Confusion Matrix (XGBoost):\n", confusion_matrix(y_test_xg,
y_pred_xgb))

print(" ")

# Feature Importances
feature_importances_xgb = pd.DataFrame({
    'Feature': X_train_xg.columns,
    'Importance': best_xgb.feature_importances_
}).sort_values(by='Importance', ascending=False)
print("Top 10 Feature Importances (XGBoost):\n",
feature_importances_xgb.head(10))
# Sort and select top 10 important features
top_features = feature_importances_xgb.head(10)

# Code for combined ROC curve
import matplotlib.pyplot as plt
import numpy as np
```

```
from sklearn.metrics import roc_curve, auc

fpr_log, tpr_log, _ = roc_curve(y_test_log, y_pred_proba_log)
roc_auc_log = auc(fpr_log, tpr_log)
fpr_ran, tpr_ran, _ = roc_curve(y_test_rf, y_pred_proba_rf)
roc_auc_ran = auc(fpr_ran, tpr_ran)
fpr_xg, tpr_xg, _ = roc_curve(y_test_xg, y_pred_proba_xgb)
roc_auc_xg = auc(fpr_xg, tpr_xg)

plt.figure(figsize=(8, 6))
plt.plot(fpr_log, tpr_log, color='blue', label=f'Logistic Regression (AUC =
{roc_auc_log:.2f})')
plt.plot(fpr_ran, tpr_ran, color='green', label=f'Random Forest (AUC =
{roc_auc_ran:.2f})')
plt.plot(fpr_xg, tpr_xg, color='red', label=f'XGBoost (AUC =
{roc_auc_xg:.2f})')
plt.plot([0, 1], [0, 1], 'k--', label='Random Guess')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - All Models')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# Code for combined Precision-Recall curve
from sklearn.metrics import precision_recall_curve, average_precision_score

precision_log, recall_log, _ = precision_recall_curve(y_test_log,
y_pred_proba_log)
ap_log = average_precision_score(y_test_log, y_pred_proba_log)

precision_rf, recall_rf, _ = precision_recall_curve(y_test_rf,
y_pred_proba_rf)
ap_rf = average_precision_score(y_test_rf, y_pred_proba_rf)

precision_xgb, recall_xgb, _ = precision_recall_curve(y_test_xg,
y_pred_proba_xgb)
ap_xgb = average_precision_score(y_test_xg, y_pred_proba_xgb)

plt.figure(figsize=(8, 6))
plt.plot(recall_log, precision_log, label=f'Logistic Regression (AP =
{ap_log:.2f})', color='blue')
plt.plot(recall_rf, precision_rf, label=f'Random Forest (AP =
{ap_rf:.2f})', color='green')
plt.plot(recall_xgb, precision_xgb, label=f'XGBoost (AP = {ap_xgb:.2f})',
color='red')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve - All Models')
plt.legend(loc='upper right')
plt.grid(True)
plt.show()
```