Credit Card Default Prediction

A Supervised Machine Learning Approach

Introduction

- The increasing prevalence of credit card use has led to the problem of credit card default, impacting both financial institutions and consumers.
- Machine learning techniques can be used to predict credit card default based on various factors, providing early warning signals to financial institutions.
- The aim of our study is to develop a robust predictive model to identify potential credit card defaulters.

Problem Statement

- **Problem:** Credit card default results in significant losses for financial institutions and can lead to financial instability for consumers. Identifying potential defaulters in advance could help mitigate these effects.
- **Objective:** To predict the probability of a customer defaulting on their credit card payments using supervised machine learning algorithms.
- **Approach:** Develop and compare the performance of four machine learning models: Logistic Regression, C-Support Vector Classification, Random Forest, and Gradient Boosting (XGBoost).
- **Evaluation:** Models will be evaluated based on testing accuracy, efficiency (training runtime) and Area Under the Receiver Operating Characteristic (AUROC).

Dataset

- Available on the UC Irvine Machine Learning Repository
- Name: default of credit card clients
- Creator: I-Cheng Yeh
- DOI: 10.24432/C55S3H
- Link: https://archive.ics.uci.edu/dataset/350/default+of+credit+card+clients

Feature Selection

	Features					Label	Feature
	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	default	PAY_LATE
0	20000	2	2	1	24	1	-2
1	120000	2	2	2	26	1	3
2	90000	2	2	2	34	0	0
3	50000	2	2	1	37	0	0
4	50000	1	2	1	57	0	-2
5	50000	1	1	2	37	0	0
6	500000	1	1	2	29	0	0
7	100000	2	2	2	23	0	-3
8	140000	2	3	1	28	0	2
9	20000	1	3	2	35	0	-10

default: 1 = yes; 0 = no.

Number of instances = 30000

LIMIT_BAL: Amount of the given credit.
SEX: 1 = male; 2 = female.
EDUCATION: 1 = graduate school; 2 = university; 3 = high school; 4 = others.
MARRIAGE: 1 = married; 2 = single; 3 = others.
AGE: in year.
PAY_LATE: Sum of past monthly payment penalty points. Measurement scale: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; . . .; 8 = payment delay for eight months; 9 = payment delay for nine months and above.

Data Splitting, Feature Scaling and Class Balancing

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Apply feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Apply SMOTE to oversample the minority class
smote = SMOTE(random_state=1)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scaled, y_train)
```

Model Training

```
# Create a Logistic Regression Model
LR = LogisticRegression(random_state=1)

# Start the timer
start_time = time.time()

# Train the SVM classifier
LR.fit(X_train_resampled, y_train_resampled)

# Stop the timer and calculate the runtime
runtime = time.time() - start_time
```

```
# Create an SVM classifier
svm_classifier = SVC(kernel='rbf', C=0.1, gamma='scale', random_state=1)

# Start the timer
start_time = time.time()

# Train the SVM classifier
svm_classifier.fit(X_train_resampled, y_train_resampled)

# Stop the timer and calculate the runtime
runtime = time.time() - start_time
```

Model Training

```
# Create the Random Forest classifier
rf_classifier = RandomForestClassifier(random_state=1, max_depth= None, min_samples_split= 5, n_estimators= 300)

# Start the timer
start_time = time.time()

# Train the classifier on the resampled training data
rf_classifier.fit(X_train_resampled, y_train_resampled)

# Stop the timer and calculate the runtime
runtime = time.time() - start_time
```

```
# Create an XGBoost classifier
xgb_classifier = xgb.XGBClassifier(random_state=1, learning_rate=0.1, max_depth=5, n_estimators=300)

# Start the timer
start_time = time.time()

# Train the XGBoost classifier
xgb_classifier.fit(X_train_resampled, y_train_resampled)

# Stop the timer and calculate the runtime
runtime = time.time() - start_time
```

Model Tuning using GridSearchCV

```
# Define the parameter grid for GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 4, 5],
    'learning_rate': [0.1, 0.01, 0.001]
}

# Create the GridSearchCV object
grid_search = GridSearchCV(xgb_classifier, param_grid, scoring='accuracy', cv=5)

# Start the timer
start_time = time.time()

# Fit the GridSearchCV object to the resampled training data
grid_search.fit(X_train_resampled, y_train_resampled)

# Stop the timer and calculate the runtime
runtime = time.time() - start_time

# Get the best parameters and best score from the grid search
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

Results

Logistic Regression

Accuracy: 0.6356666666666667
Classification Report:

	precision	recall	f1-score	support
0	0.86	0.63	0.73	4663
1	0.33	0.64	0.44	1337
accuracy			0.64	6000
macro avg	0.60	0.64	0.58	6000
weighted avg	0.74	0.64	0.67	6000

Runtime: 0.04 seconds

Random Forest

Accuracy: 0.6773
Classification Report:

	precision	recall	f1-score	support
0	0.77	0.83	0.80	4663
1	0.20	0.15	0.18	1337
accuracy			0.68	6000
macro avg	0.49	0.49	0.49	6000
weighted avg	0.65	0.68	0.66	6000

Runtime: 13.23 seconds

C-Support Vector Classification (SVI

Accuracy: 0.7725
Classification Report:

	precision	recall	f1-score	support
0	0.85	0.86	0.85	4663
1	0.49	0.48	0.49	1337
accuracy			0.77	6000
macro avg	0.67	0.67	0.67	6000
weighted avg	0.77	0.77	0.77	6000

Runtime: 54.46 seconds

XGBoost (Gradient Boosting)

Accuracy: 0.8013
Classification Report:

	precision	recall	f1-score	support
0	0.83	0.93	0.88	4663
1	0.59	0.35	0.44	1337
accuracy			0.80	6000
macro avg	0.71	0.64	0.66	6000
weighted avg	0.78	0.80	0.78	6000

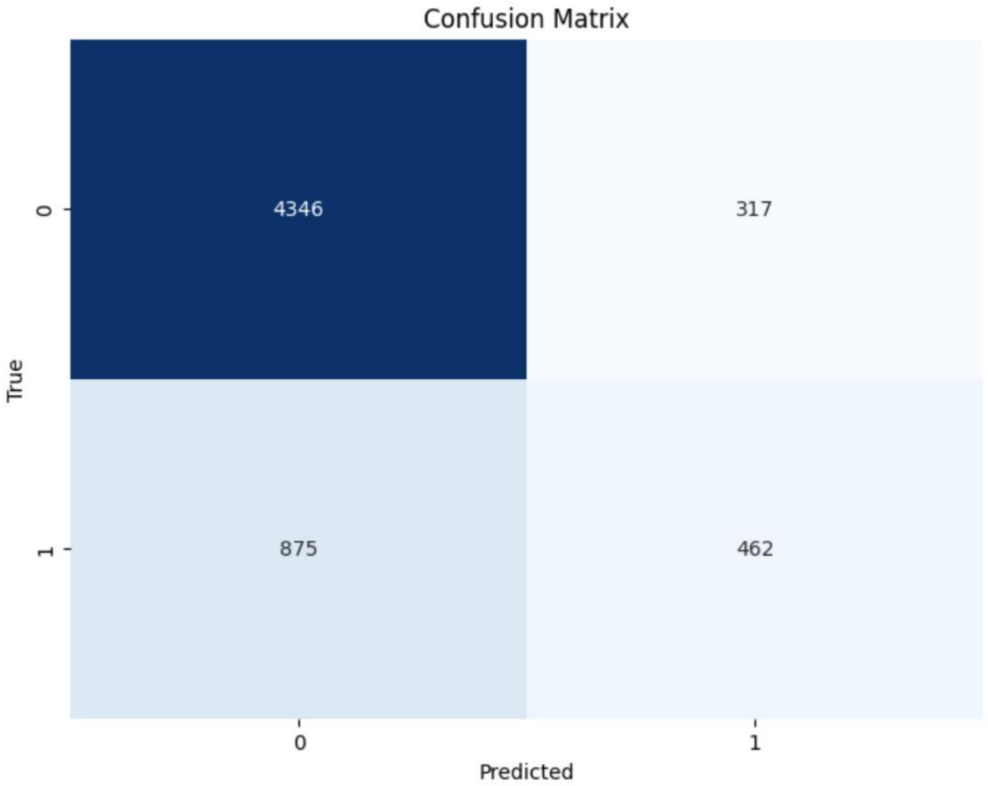
Runtime: 4.21 seconds

Model Selection

Model	Accuracy	Efficiency (Runtime)	
Logistic Regression	0.6357	0.04 s	
C-Support Vector Classification	0.7725	54.46 s	
Random Forest	0.6773	13.23 s	
XGBoost (Gradient Boosting)	0.8013	4.21 s	→ Selected

➔ Clearly, the gradient boosting method with XGBoost performed the best in term of both **accuracy** (0.8013) and **efficiency** (4.21 s).

Further Validation using XGBoost



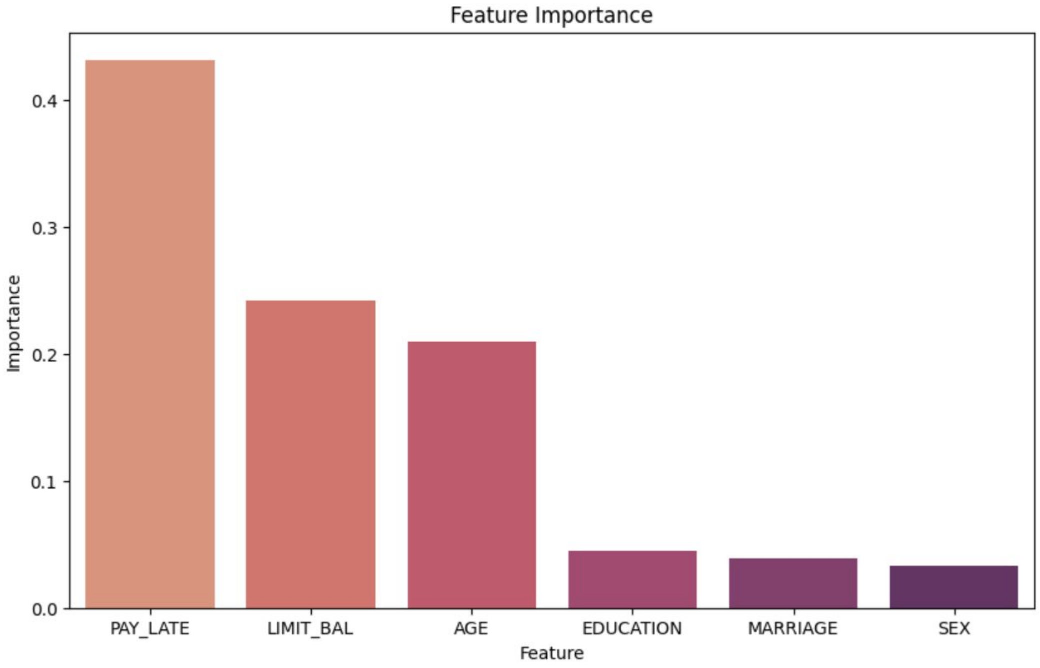
Accuracy: 0.8013

Classification Report:

	precision	recall	f1-score	support
0	0.83	0.93	0.88	4663
1	0.59	0.35	0.44	1337
accuracy			0.80	6000
macro avg	0.71	0.64	0.66	6000
weighted avg	0.78	0.80	0.78	6000

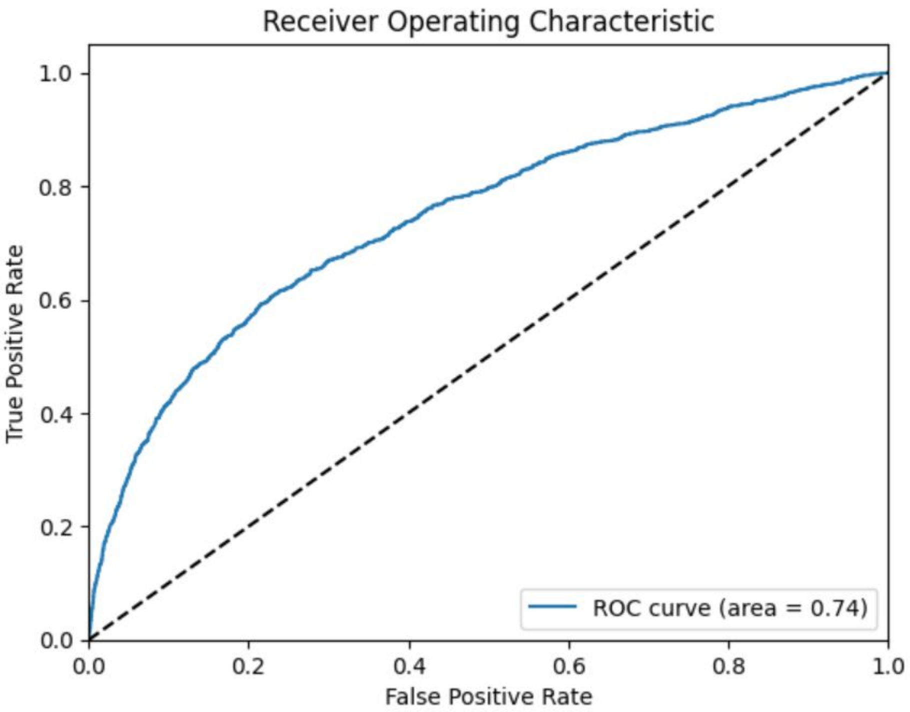
Runtime: 4.21 seconds

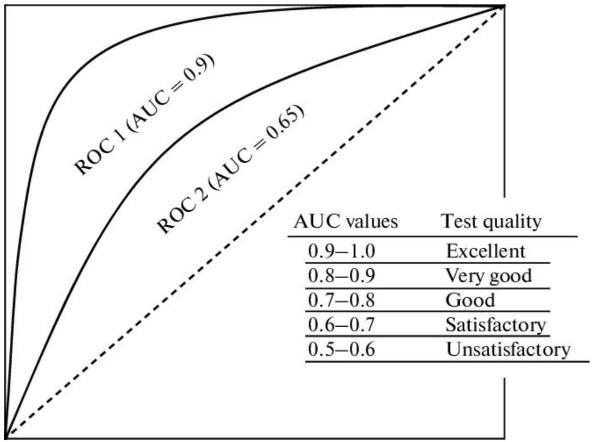
Further Validation using XGBoost




```
Feature: PAY_LATE, Importance: 0.43145278096199036
Feature: LIMIT_BAL, Importance: 0.24200183153152466
Feature: AGE, Importance: 0.2093764990568161
Feature: EDUCATION, Importance: 0.04513275995850563
Feature: MARRIAGE, Importance: 0.038681838661432266
Feature: SEX, Importance: 0.033354319632053375
```

Further Validation using XGBoost





AUC values	Test quality
0.9–1.0	Excellent
0.8–0.9	Very good
0.7–0.8	Good
0.6–0.7	Satisfactory
0.5–0.6	Unsatisfactory