Credit Loan Default Detection

Team 6A

Deep Patel, Raymond Ruan, Vincent Lee, Wenchi Tseng

Data Description

Dataset Description

- "Credit_Risk_Analysis", from Kaggle
- 12 Columns x 32,582 Rows

Our Goal

- Predict the likelihood of the applicant's default
- Help lenders make informed lending decisions

- **ID**: Unique identifier for each loan applicant.
- **Age**: Age of the loan applicant.
- **Income**: Income of the loan applicant.
- **Home**: Home ownership status (Own, Mortgage, Rent).
- **Emp_Length**: Employment length in years.
- **Intent**: Purpose of the loan (e.g., education, home improvement).
- **Amount**: Loan amount applied for.
- **Rate**: Interest rate on the loan.
- **Status**: Loan approval status (Fully Paid, Charged Off, Current).
- **Percent_Income**: Loan amount as a percentage of income.
- **Default**: Whether the applicant has defaulted on a loan previously (Yes, No).
- **Cred_Length**: Length of the applicant's credit history.

Id	# Age	# Income	Home	# Emp_length	Intent	# Amount	# Rate	# Status	# Percent_income
Unique identifier for each person	Person's age	Person's income	Home ownership status	Employment length	Loan intent	Loan amount	Loan interest rate	Loan status	Loan percent of income
0 – 32.8k	20 – 144	4000 – 6.00m	RENT 50% MORTGAGE 41% Other (2691) 8%	0 – 123	EDUCATION 20% MEDICAL 19% Other (20057) 62%	500 – 35.0k	5.42 – 23.2	0 – 1	0 – 0.83
0	22	59000	RENT	123	PERSONAL	35000	16.02	1	0.59
1	21	9600	OWN	5	EDUCATION	1000	11.14	0	0.1
2	25	9600	MORTGAGE	1	MEDICAL	5500	12.87	1	0.57

Data Source: https://www.kaggle.com/datasets/nanditapore/credit-risk-analysis/data

Essential Data Preparation and Training Data

Essential Data Preparation

- Replace missing value with mean

```
#We use SimpleImputer to fill in the missing values with mean
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df_encoded), columns=df_encoded.columns)
```

- Transform category into numerical values

```
# We choose to use OneHotEncoder to transform our variables
from sklearn.preprocessing import OneHotEncoder

df_encoded = pd.get_dummies(df, columns=["Home", "Intent"], drop_first=True)
df_encoded['Default'] = [1 if i == "Y" else 0 for i in df['Default']]
```

Split and Clean Data

- Test size = 20%; Training size = 80%
- Random_state = 42

```
#Split data into x (features) and y (target)
X = df_cleaned.drop(columns=['Default'])
y = df_cleaned['Default']
```

```
# Create training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_st
```

- Drop irrelevant values

```
# Id column does not provide any useful information so we drop it
# df_cleaned will be our initial model dataset from this point
df_cleaned = df_imputed.drop(["Id"], axis=1)
```

Multinomial Naive Bayes - Benchmark

Why Naive Bayes?

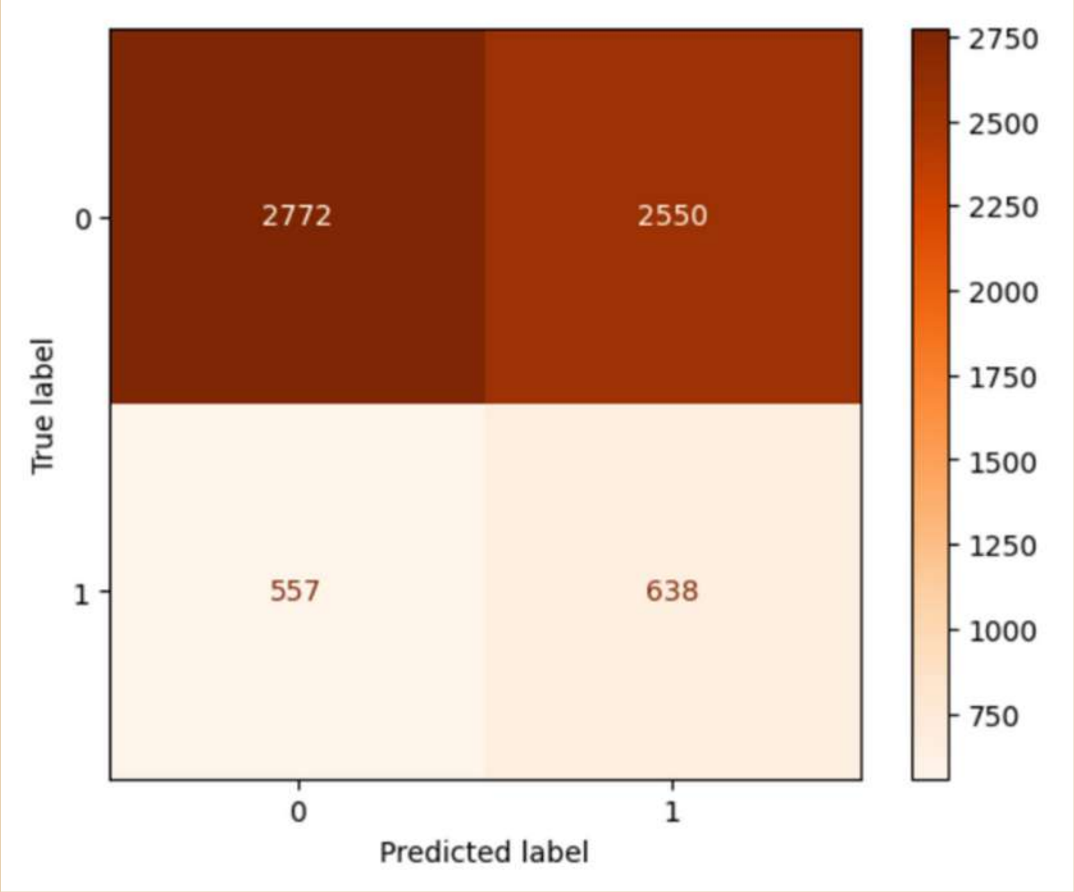
1. Straightforward interpretation
2. Computational efficiency in training data
3. Categorical features in our dataset

Our dataset :

Id	Age	Income	Home	Emp_length	Intent	Amount	Rate	Status	Percent_income	Default
0	22	59000	RENT	123.0	PERSONAL	35000	16.02	1	0.59	Y
1	21	9600	OWN	5.0	EDUCATION	1000	11.14	0	0.10	N
2	25	9600	MORTGAGE	1.0	MEDICAL	5500	12.87	1	0.57	N
3	23	65500	RENT	4.0	MEDICAL	35000	15.23	1	0.53	N
4	24	54400	RENT	8.0	MEDICAL	35000	14.27	1	0.55	Y

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.83      0.52      0.64      5322
         1.0       0.20      0.53      0.29      1195

    accuracy                           0.52      6517
   macro avg       0.52      0.53      0.47      6517
weighted avg       0.72      0.52      0.58      6517

Accuracy: 0.5232468927420593
```

Accuracy of Benchmark Model ≈ 0.523



Not Default = 5322; Default = 1195

Multinomial Naive Bayes After Pre-Processing

Cross-Validation & Parameters Optimization

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV

# Define the parameter grid
param_grid = {'alpha': [0.02, 0.1, 0.5, 1.0, 2.0, 5.0] }

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=MultinomialNB(), param_grid=param_grid, cv=10, n_jobs=-1, verbose=2)

# Fit GridSearchCV
grid_search.fit(X_train, y_train)

# Best parameters
print("Best Parameters:", grid_search.best_params_)
```

```
Fitting 10 folds for each of 6 candidates, totalling 60 fits
Best Parameters: {'alpha': 0.02}
```

Accuracy = 0.52

(no change)

Feature Selection- Wrapper Method

```
# Selected Features
from sklearn.feature_selection import SequentialFeatureSelector
fsSvm = SequentialFeatureSelector(nb_model, scoring='recall', n_features_to_select=4)
fsSvm.fit(X_train, y_train)

selectedFeatureIndices = fsSvm.get_support(indices=True)
selectedFeatureColNames = X_train.columns[selectedFeatureIndices]
print("Selected Features(Forward selection):")
list(selectedFeatureColNames)
```

```
Selected Features(Forward selection):
['Age', 'Income', 'Amount', 'Rate']
```

Accuracy = 0.5237

(increase by 0.0005)

Scaling and SMOTE

Step1.
Use SMOTE to balance class distribution

```
# Introduce new resampled training data
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

Step2.
Scale the data to ensure equal weight

```
# Standardization
scaler = StandardScaler()
X_resampled_scaled = scaler.fit_transform(X_resampled)
X_test_scaled = scaler.transform(X_test)
```

Classification Report :

	precision	recall	f1-score	support
0.0	0.86	0.80	0.83	5322
1.0	0.33	0.44	0.38	1195
accuracy			0.73	6517
macro avg	0.60	0.62	0.60	6517
weighted avg	0.77	0.73	0.75	6517

Accuracy: 0.7334663188583704

Increase by 0.2 !

Logistic Regression - Benchmark

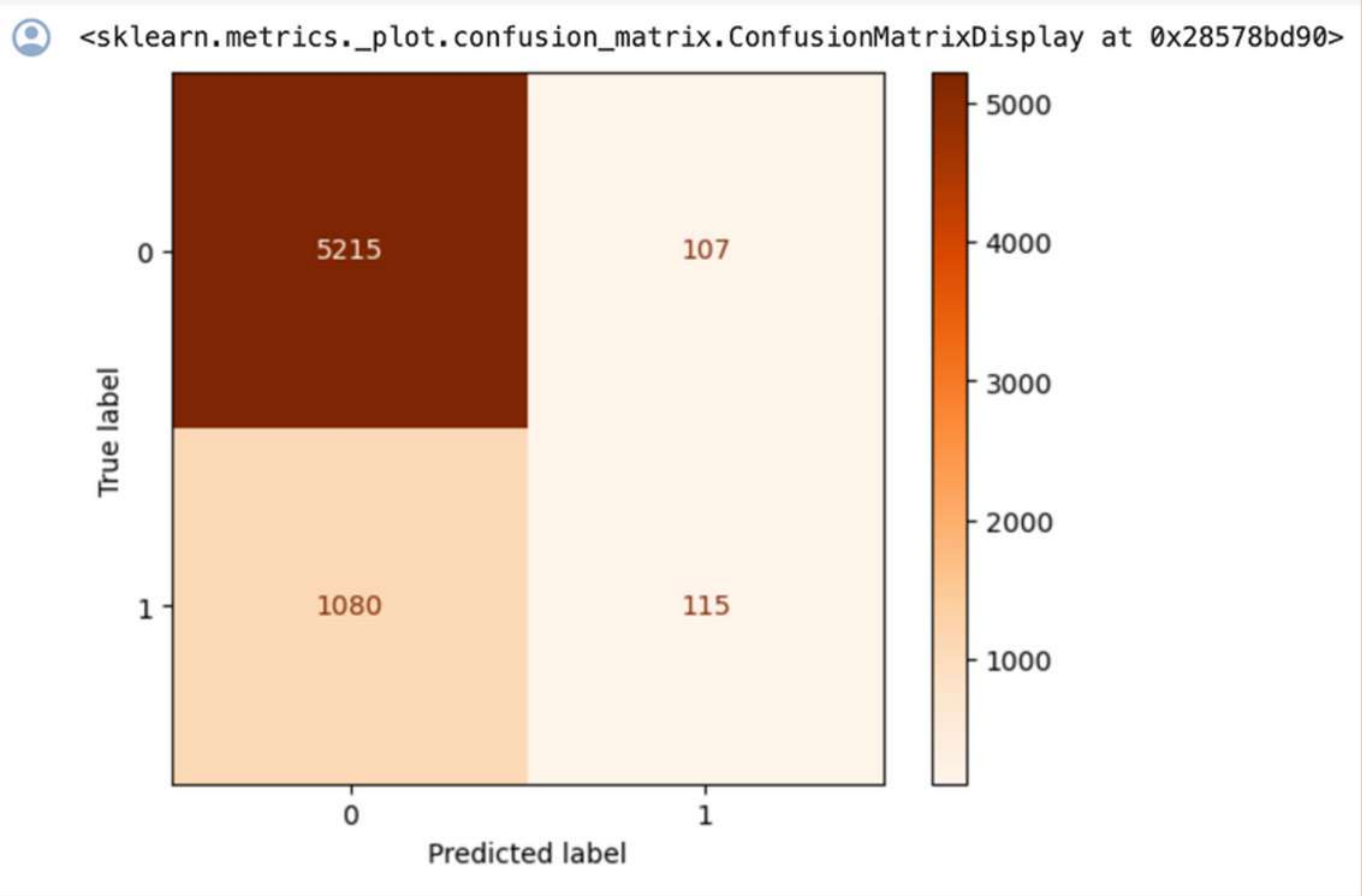
Classification Report

```
# Classification report
print(f"Classification Report:\n{classification_report(y_test, logreg_pred)}")
print(f"Accuracy: {accuracy_score(y_test, logreg_pred)}")
```

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.83      0.98      0.90      5322
         1.0       0.52      0.10      0.16      1195

    accuracy                           0.82      6517
   macro avg       0.67      0.54      0.53      6517
weighted avg       0.77      0.82      0.76      6517

Accuracy: 0.8178609789780574
```

Accuracy of Benchmark Model ≈ 0.817

Model Initialization: The logistic regression model is initialized with the solver='liblinear' and random_state= 42



Confusion Matrix

These metrics serve as vital indicators of the logistic regression model's efficacy in our binary classification task, facilitating a comprehensive evaluation of its performance.

Logistic Regression After Pre-Processing

```
[ ] # Utilize lasso (l1) regularization and the C parameters
    new_logreg_model = LogisticRegression(penalty='l1', C=1, solver='liblinear', random_state=42)
```

- Pre-processing:

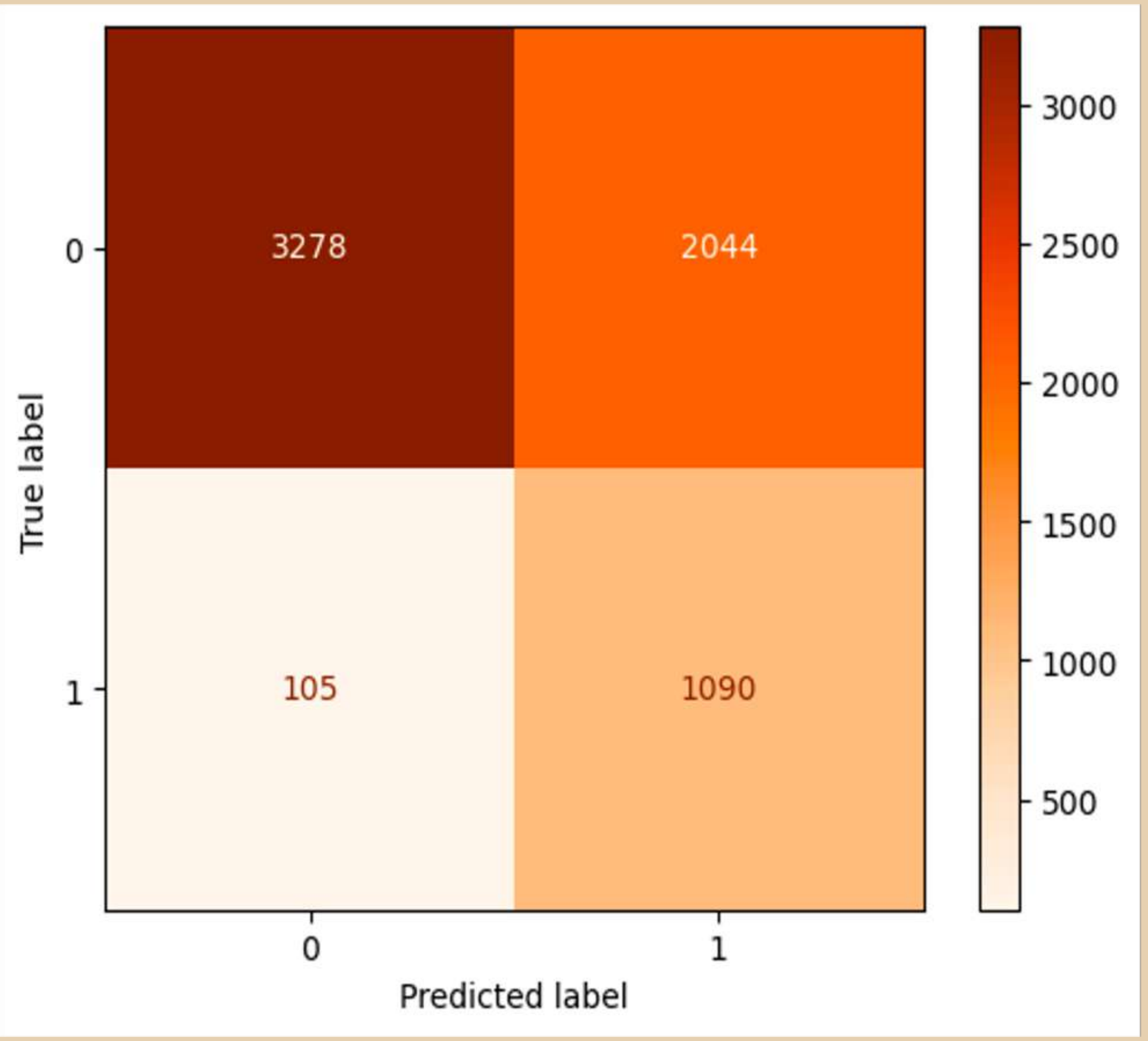
  Scaling, LASSO, and C Parameter

- Benchmark Comparison:

  Decreased our accuracy by 0.04

Accuracy of Updated Model ≈ 0.784

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.94      0.78      0.86      5322
         1.0       0.45      0.79      0.57      1195

    accuracy                           0.78      6517
   macro avg       0.70      0.79      0.72      6517
weighted avg       0.85      0.78      0.80      6517

Accuracy: 0.7845634494399264
```



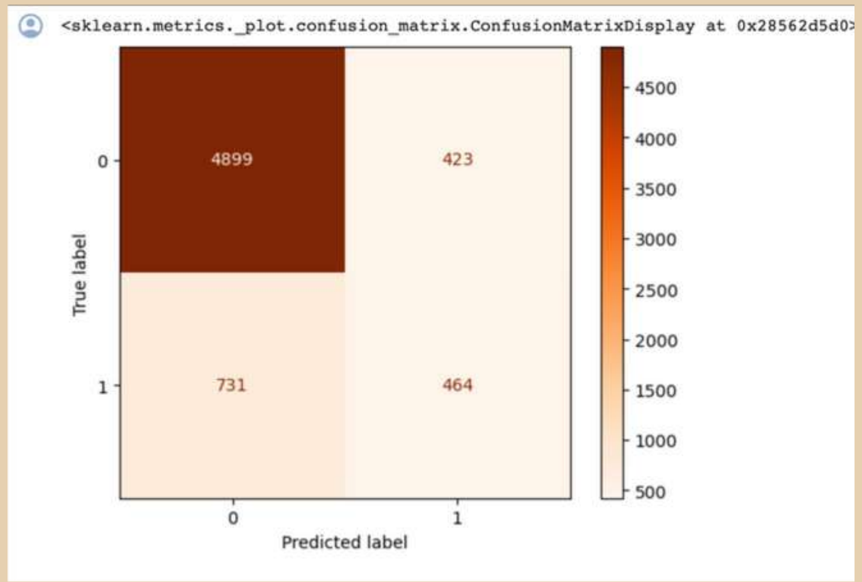
Random Forest - Benchmark

Building the model

```
[ ] # Fitting our Random Forest model on our training data
    rf_model = RandomForestClassifier(random_state=42)
    rf_model.fit(X_train, y_train)

    # Getting our prediction
    rf_pred = rf_model.predict(X_test)
```

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.87      0.92      0.89      5322
         1.0       0.52      0.39      0.45      1195

    accuracy                           0.82      6517
   macro avg       0.70      0.65      0.67      6517
weighted avg       0.81      0.82      0.81      6517

Accuracy: 0.8229246585852386
```

Accuracy = 0.8229



Why Random Forest

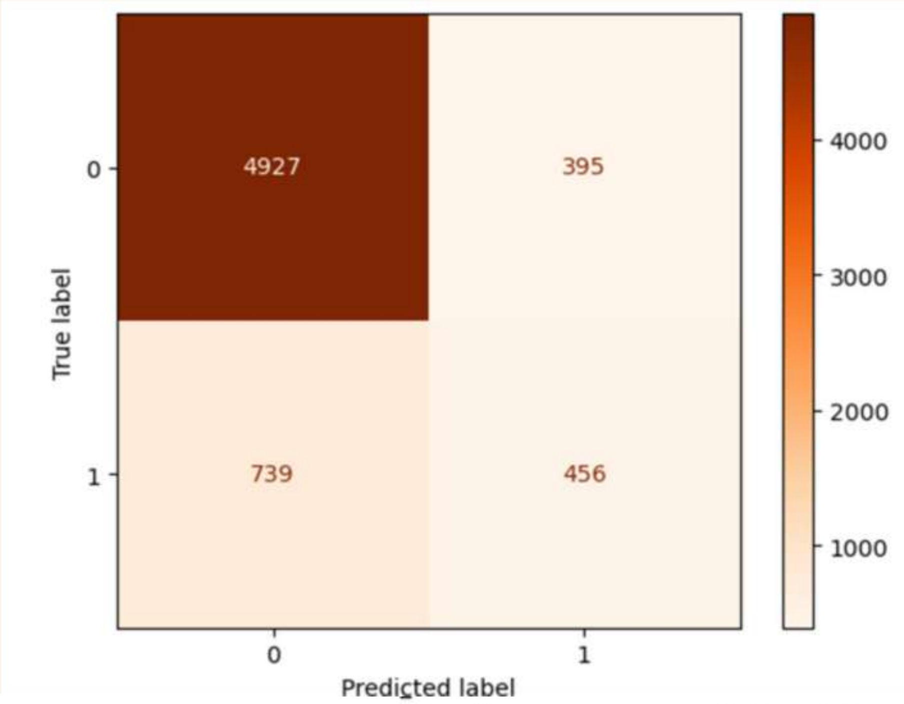
1. Creation of Decisions Tree
2. Robustness to Overfitting
3. Handling Loan Default Prediction
4. Improved Accuracy

Random Forest After Pre-Processing

```python
param_grid = {
    'n_estimators': [50, 100, 150, 200, 250, 300],
    'max_depth': [5, 10, 15, 20, 25, 30],
    'min_samples_split' : [2, 5, 0],
    'min_samples_leaf' : [1, 2, 4]
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=RandomForestClassifier(random_state=42), param_grid=param_grid, cv=5, n_jobs=-

# Fit GridSearchCV
grid_search.fit(X_train_scaled, y_train)
```

```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
Best Parameters: {'max_depth': 15, 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200}
```



	precision	recall	f1-score	support
0.0	0.87	0.93	0.90	5322
1.0	0.54	0.38	0.45	1195
accuracy			0.83	6517
macro avg	0.70	0.65	0.67	
weighted avg	0.81	0.83	0.81	

Accuracy: 0.8259935553168636

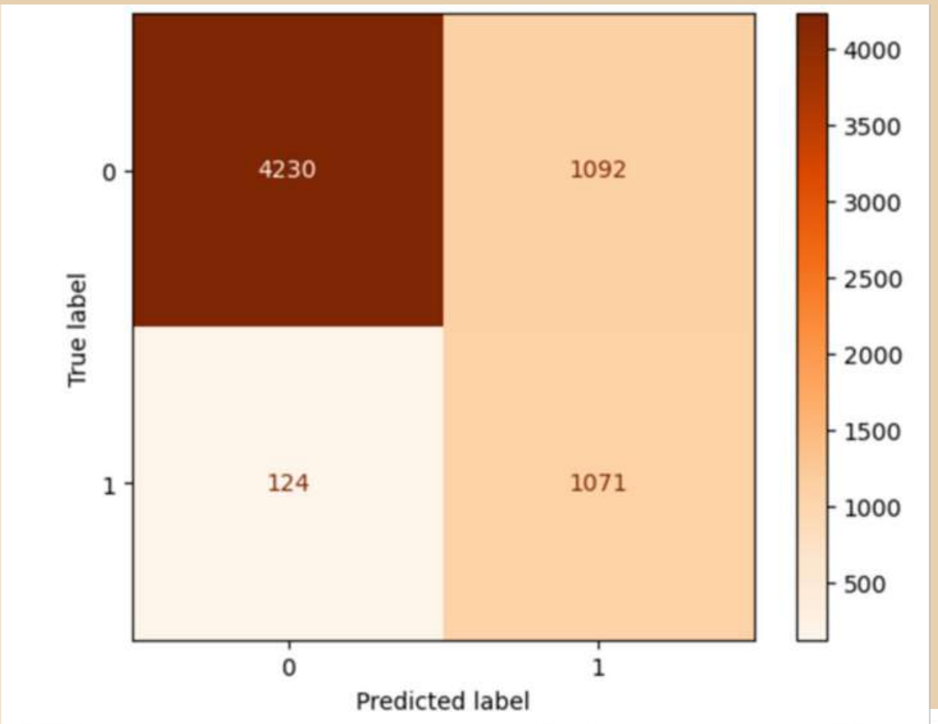
1. Scaling Data
2. Cross Validation
3. Optimizing Parameters

Accuracy = 0.82599

(increase by 0.003)

```python
# Import module
from imblearn.ensemble import BalancedRandomForestClassifier

# Fit our balanced rf model on training data
brfc = BalancedRandomForestClassifier(random_state=42)
brfc.fit(X_train, y_train)

# Our prediction
brfc_pred = brfc.predict(X_test)
```

1. Balanced Random Forest Classifier



	precision	recall	f1-score	support
0.0	0.97	0.79	0.87	5322
1.0	0.50	0.90	0.64	1195
accuracy			0.81	6517
macro avg	0.73	0.85	0.76	6517
weighted avg	0.88	0.81	0.83	6517

Accuracy: 0.8134110787172012

Accuracy = 0.81341

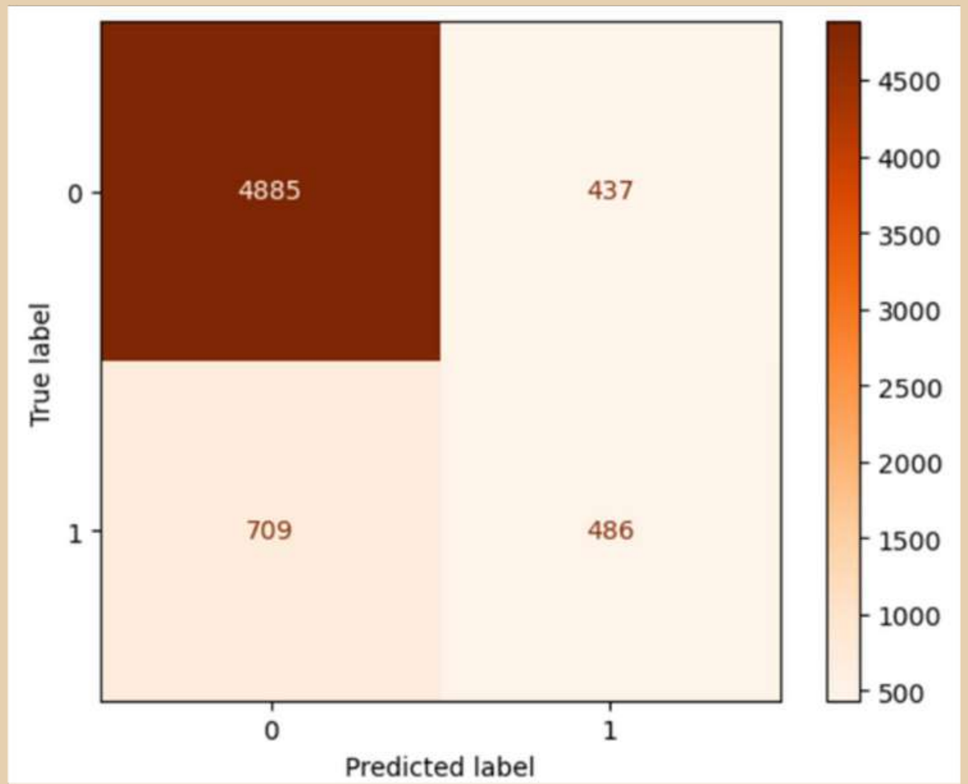
(Decrease by 0.009)

XGBoost - Benchmark

Building the model

```
# Fitting our model on our initial training data
xgb_model = xgb.XGBClassifier(random_state=42)
xgb_model.fit(X_train, y_train)

# Getting our predication
xgb_pred = xgb_model.predict(X_test)
```

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.87      0.92      0.90      5322
         1.0       0.53      0.41      0.46      1195

    accuracy                           0.82      6517
   macro avg       0.70      0.66      0.68      6517
weighted avg       0.81      0.82      0.82      6517

Accuracy: 0.8241522172778886
```



Accuracy of Benchmark Model ≈ 0.824

Why XGBoost?

- Recent ensemble model developed in 2014 to improve the traditional Gradient Boosting model
- Known for its fast performance and consistent accuracy
- Uses decision trees as base learner

XGBoost After Pre-Processing

SMOTE, Standardizing, “Scale_Pos_Weight”

- Addresses class imbalance and outliers

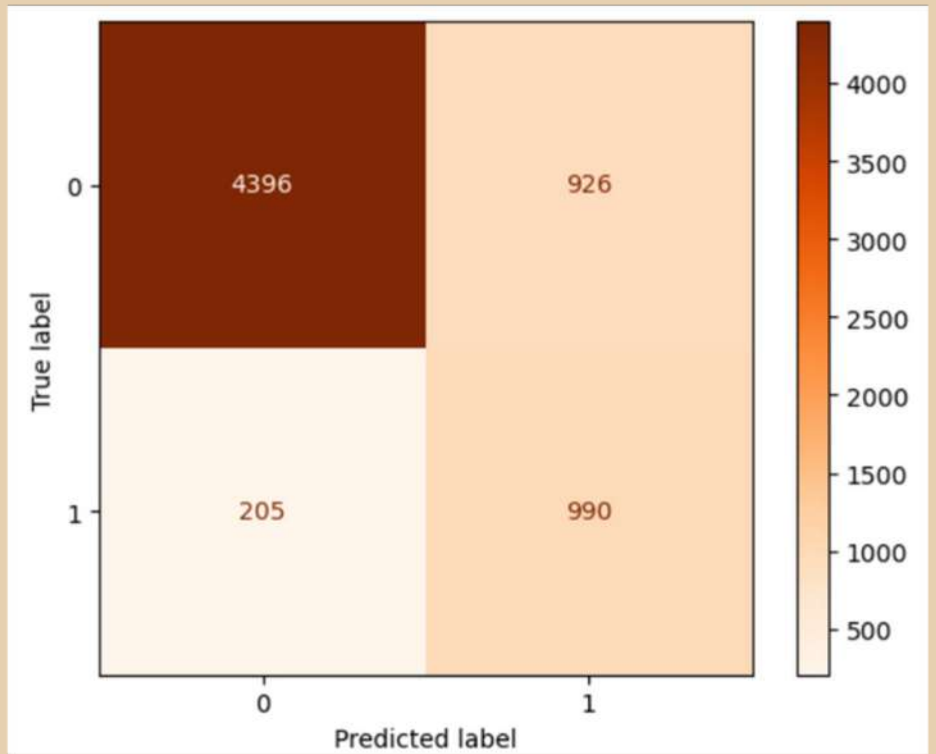
```python
# Introduce new resampled training data
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

# Scale the data, including test data after resampling it to see if it further improves the model
scaler = StandardScaler()
X_resampled_scaled = scaler.fit_transform(X_resampled)
X_test_scaled = scaler.transform(X_test)
```

```python
# Implementing our new model on the resampled and scaled data
new_xgb_model = xgb.XGBClassifier(scale_pos_weight=3, random_state=42)
new_xgb_model.fit(X_resampled_scaled, y_resampled)

# Getting our new prediction
new_xgb_pred = new_xgb_model.predict(X_test_scaled)
```

Classification Report:

	precision	recall	f1-score	support
0.0	0.96	0.83	0.89	5322
1.0	0.52	0.83	0.64	1195
accuracy			0.83	6517
macro avg	0.74	0.83	0.76	6517
weighted avg	0.87	0.83	0.84	6517

Accuracy: 0.8264538898266073



Accuracy ≈ 0.826 (improved by 0.002)

SMOTE, Standardizing, Predictability Threshold

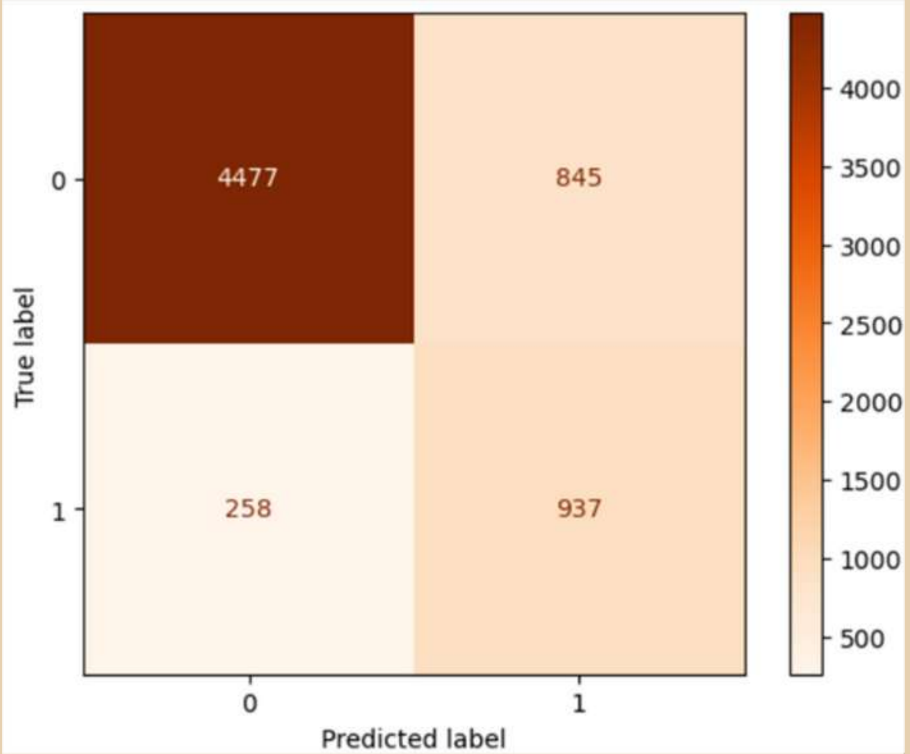
- y_probs controls balance of FN & FP predictions
  - Higher value = more conservative in positive predictions

```python
y_probs = new_xgb_model.predict_proba(X_test_scaled)[:, 1]

# Adjust the threshold
adj_xgb_pred = (y_probs > 0.57).astype(int)
```

Classification Report:

	precision	recall	f1-score	support
0.0	0.95	0.84	0.89	5322
1.0	0.53	0.78	0.63	1195
accuracy			0.83	6517
macro avg	0.74	0.81	0.76	6517
weighted avg	0.87	0.83	0.84	6517

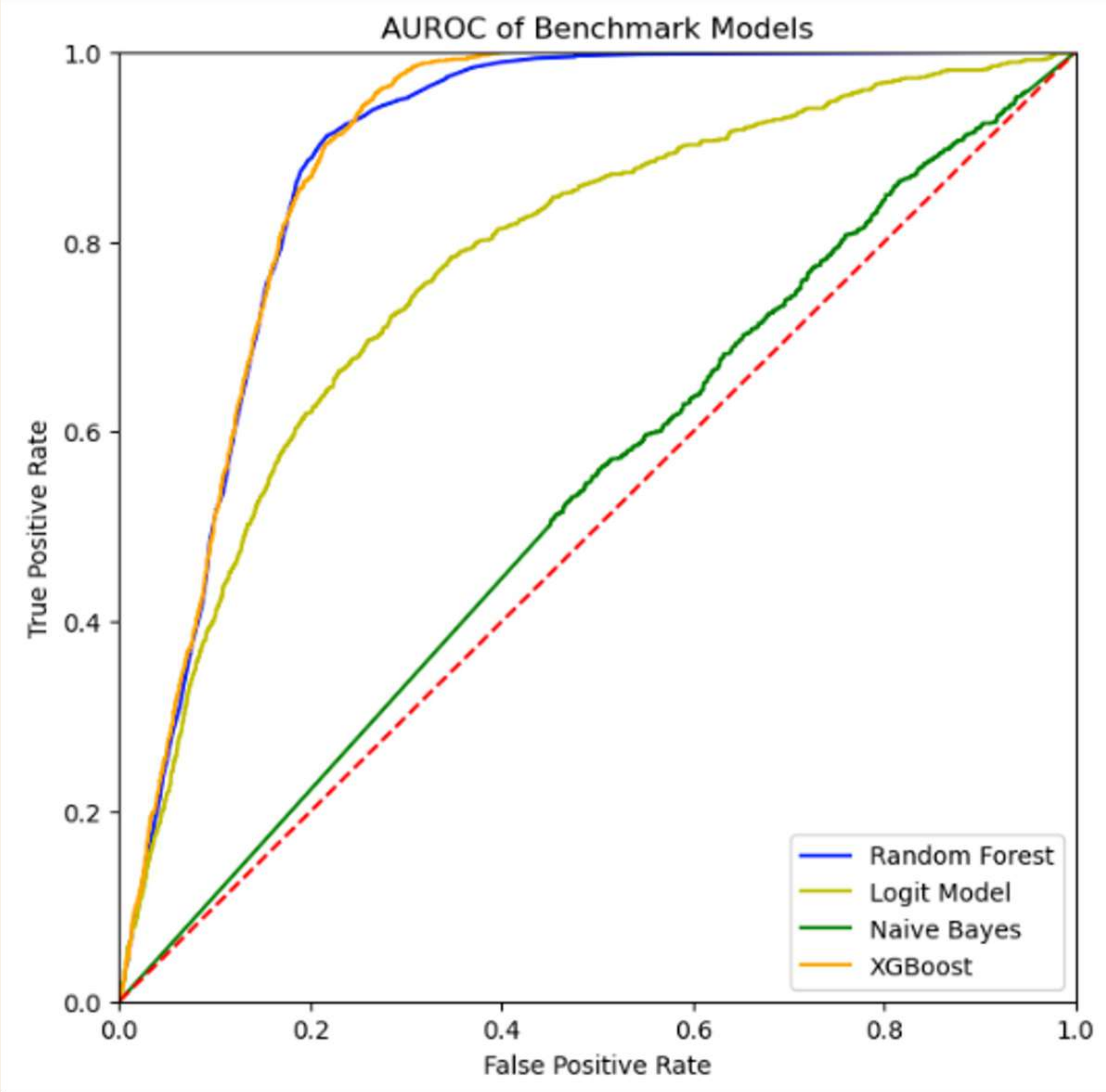
Accuracy: 0.8307503452508823



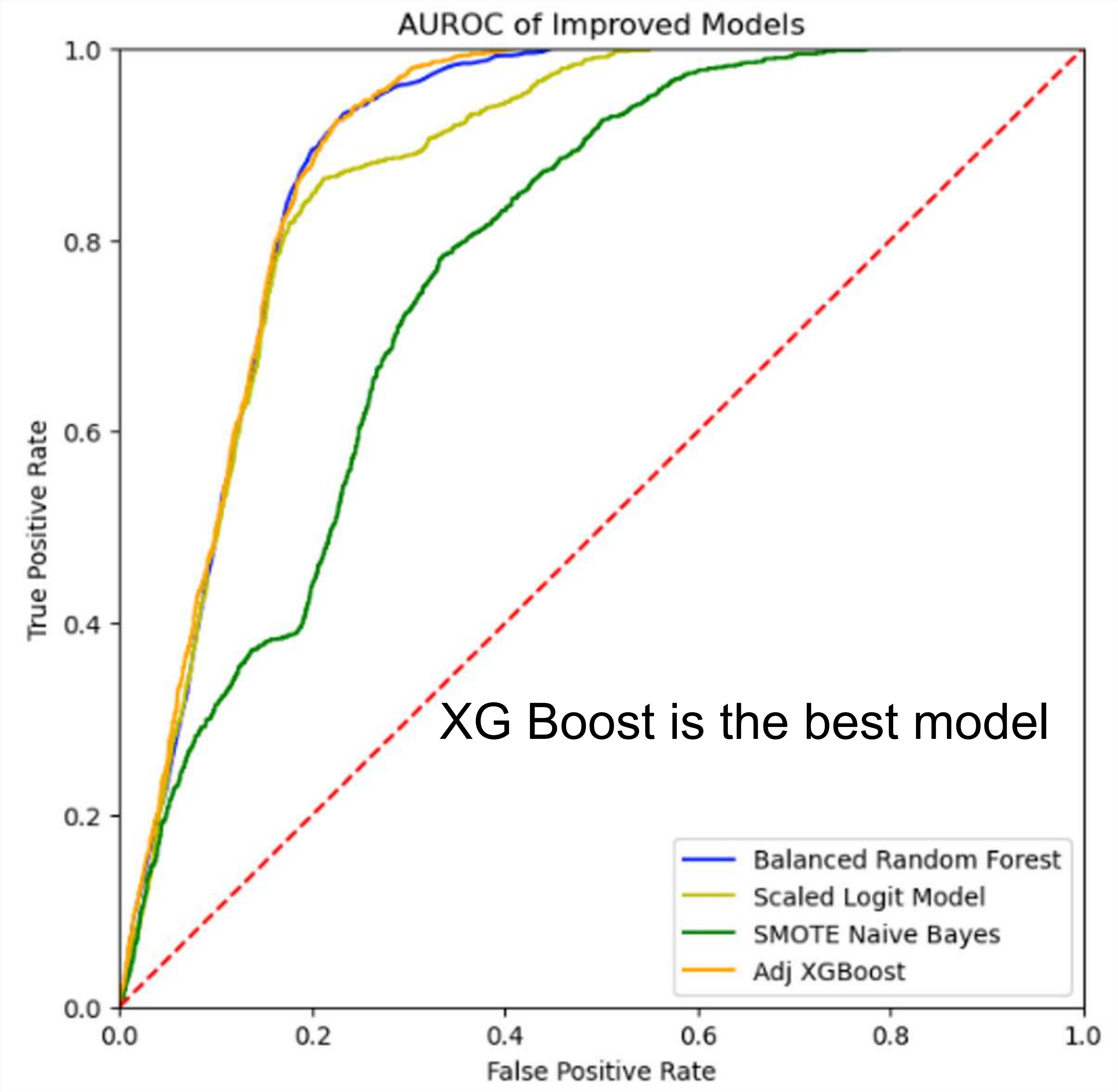
Accuracy ≈ 0.831 (improved by 0.007)

Visualization of Model Performances

Before Pre-Processing



After Pre-Processing



Takeaways

1. Importance of Data Preprocessing
2. Impact of Model Selection
3. Preprocessing Method Selection
4. Parameter Tuning Significance
5. Overall Accuracy ≠ Good Model

Classification Report:

	precision	recall	f1-score	support
0.0	0.82	1.00	0.90	5322
1.0	0.00	0.00	0.00	1195
accuracy			0.82	6517
macro avg	0.41	0.50	0.45	6517
weighted avg	0.67	0.82	0.73	6517

Classification Report:

	precision	recall	f1-score	support
0.0	0.87	0.92	0.90	5322
1.0	0.53	0.41	0.46	1195
accuracy			0.82	6517
macro avg	0.70	0.66	0.68	6517
weighted avg	0.81	0.82	0.82	6517

Both models have the same overall accuracy but consider the precision score for each class.

Thank You