

**UNIVERSITI TUNKU ABDUL RAHMAN**

**ACADEMIC YEAR 2024/2025**

**UECS3483**

**DATA MINING**

**GROUP ASSIGNMENT**

**BY**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Student ID** | **Name** | **Programme** | **Department** | **Tutorial Group** |
| 2202541 | Brandon Vun Kui Yung | SE | LKC FES | P3 |
| 2103898 | Yap Su Hong | SE | LKC FES | P3 |

**TABLE OF CONTENTS**

**1.0 INTRODUCTION3**

**2.0 DATA PREPARATION3**

**3.0 MODEL CREATION9**

*3.1* Logistic Regression*9*

*3.2* Logistic Regression with RandomizedSearchCV*10*

*3.3* Random Forest*12*

*3.4* Random Forest with RandomizedSearchCV*14*

*3.5* Random Forest with GridSearchCV*16*

*3.6* Decision Tree with RandomizedSearchCV*18*

*3.7* Decision Tree using AdaBoost Classifier*20*

*3.8* Support Vector Machine (SVM)*22*

*3.9* Gradient Boosting with RandomizedSearchCV*24*

*3.10* Naive Bayes with RandomizedSearchCV*26*

*3.11* Neural Network with RandomizedSearchCV*28*

*3.12* Gradient Boosting using Cat Boost Classifier*30*

**4.0 ANALYSIS32**

**5.0 CONCLUSION32**

**6.0 REFERENCES 33**

**INTRODUCTION**

In this group assignment task given, we were tasked to process 20 new loan applications in a NewApplicants.csv file given. Alongside it, there are 255, 327 observations in the BankLoanApproval.csv file given to be used as training data for our usage. In this report, we will determine if the loans applications for each of be 20 new applicants should be approved or should be rejected by using the best possible model.

**DATA PREPARATION**

First, we had used one hot encoding in order to turn the categorical variables into numerical values.

Python code:

import pandas as pd

from sklearn.preprocessing import OneHotEncoder

# Load data

df = pd.read\_csv('((GAssign) BankLoanApproval.csv')

# Define categorical columns

categorical\_columns = ['Education', 'EmploymentType', 'MaritalStatus', 'LoanPurpose']

# Initialize OneHotEncoder

encoder = OneHotEncoder(sparse=False)

# Encode categorical data

encoded\_data = encoder.fit\_transform(df[categorical\_columns])

# Create DataFrame with encoded data

encoded\_df = pd.DataFrame(encoded\_data, columns=encoder.get\_feature\_names\_out(categorical\_columns))

# Replace 'Yes' and 'No' with 1 and 0 for specified columns

columns\_to\_replace = ['HasMortgage', 'HasDependents', 'HasCoSigner']

df[columns\_to\_replace] = df[columns\_to\_replace].replace({'Yes': 1, 'No': 0})

# Drop original categorical columns and 'LoanID'

columns\_to\_drop = categorical\_columns + ['LoanID']

df.drop(columns\_to\_drop, axis=1, inplace=True)

# Concatenate original DataFrame with encoded DataFrame

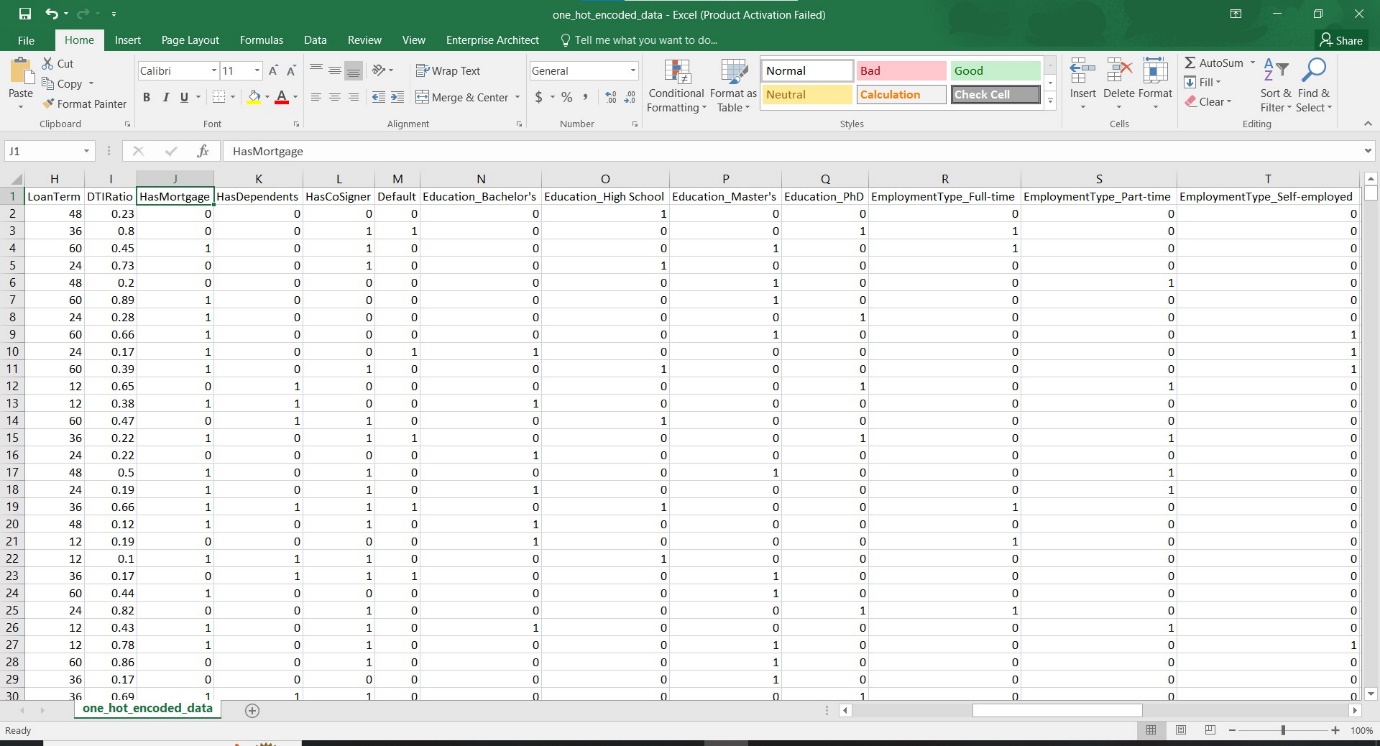
final\_data = pd.concat([df, encoded\_df], axis=1)

# Save the final data to a CSV file

final\_data.to\_csv('one\_hot\_encoded\_data.csv',index=False)

This code first loads a dataset from a CSV file ('BankLoanApproval.csv') into a pandas data frame. It then identifies the categorical columns in the data frame. It then encodes these categorical columns using one-hot encoding. After that, it replaces 'Yes' and 'No' values with 1s and 0s in the specified columns. The code then drops the original categorical columns and the column named 'LoanID'. Then, the code links the original data frame with the encoded data frame. Finally, it saves the processed data to a new CSV file named 'one\_hot\_encoded\_data.csv'.

result:

****

*Screenshot of one\_hot\_encoded\_data.csv*

Afterwards, we normalized the values of Age, Income, LoanAmount, CreditScore, MonthsEmployed, InterestRate, LoanTerm and DTIRatio.

Python code:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import MinMaxScaler

df = pd.read\_csv('C:\Users\Brandon VKY\Desktop\Data Mining Assignment\one\_hot\_encoded\_data.csv')

df.columns

result:

Index(['Age', 'Income', 'LoanAmount', 'CreditScore', 'MonthsEmployed',

'NumCreditLines', 'InterestRate', 'LoanTerm', 'DTIRatio', 'HasMortgage',

'HasDependents', 'HasCoSigner', 'Default', 'Education\_Bachelor's',

'Education\_High School', 'Education\_Master's', 'Education\_PhD',

'EmploymentType\_Full-time', 'EmploymentType\_Part-time',

'EmploymentType\_Self-employed', 'EmploymentType\_Unemployed',

'MaritalStatus\_Divorced', 'MaritalStatus\_Married',

'MaritalStatus\_Single', 'LoanPurpose\_Auto', 'LoanPurpose\_Business',

'LoanPurpose\_Education', 'LoanPurpose\_Home', 'LoanPurpose\_Other'],

dtype='object')

# Specify columns to normalize

columns\_to\_normalize = ['Age', 'Income', 'LoanAmount', 'CreditScore', 'MonthsEmployed', 'InterestRate', 'LoanTerm', 'DTIRatio']

# Initialize MinMaxScaler

scaler = MinMaxScaler()

# Create a new DataFrame to store the normalized data

normalized\_df = df.copy()

# Normalize specified columns

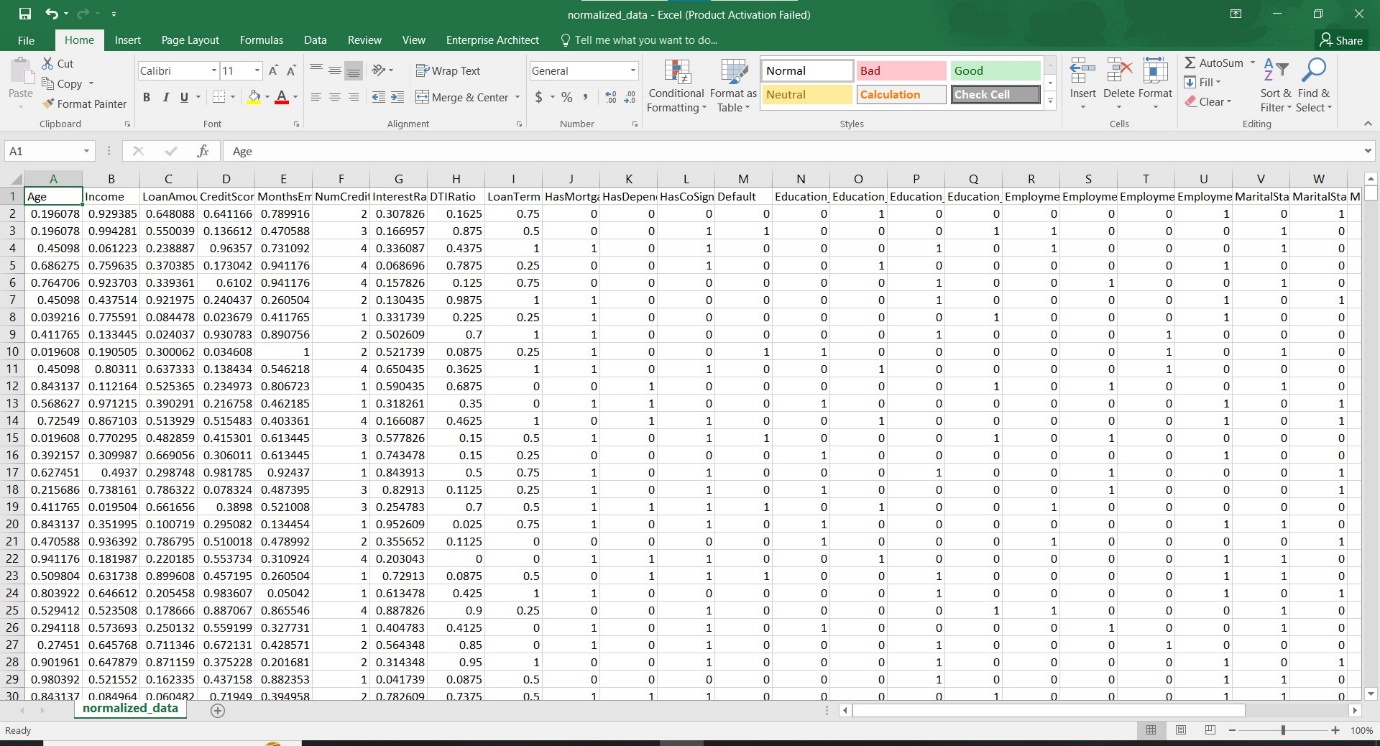
normalized\_df[columns\_to\_normalize] = scaler.fit\_transform(normalized\_df[columns\_to\_normalize])

# Save the normalized DataFrame to a new CSV file

normalized\_df.to\_csv('normalized\_data.csv', index=False)

This code first loads a dataset from a CSV file named 'one\_hot\_encoded\_data.csv' into a pandas data frame. It then identifies specific columns that require normalization, such as 'Age', 'Income', 'LoanAmount', and others. Utilizing the MinMaxScaler from the sklearn.preprocessing module, it scales these selected columns to a range between 0 and 1. The normalized data is stored in a new data frame called 'normalized\_df'. Finally, the code saves this processed data to a new CSV file named 'normalized\_data.csv', excluding the indices. This normalization procedure ensures that the features are on a similar scale, which can be beneficial for certain machine learning algorithms and analysis tasks.

result:



*Screenshot of normalized\_data.csv*

After getting the normalized data file, we used regression summary in order to find the p-values of the data.

Python code:

import pandas as pd

import statsmodels.api as sm

# Load the normalized DataFrame from the CSV file

normalized\_df = pd.read\_csv('normalized\_data.csv')

# Specify target column name

target\_column = 'Default'  # Replace 'Default' with the name of your target column

# Split the data into features (X) and target variable (y)

X = normalized\_df.drop(columns=[target\_column])

y = normalized\_df[target\_column]

# Add constant to features for the intercept term

X = sm.add\_constant(X)

# Fit the regression model

model = sm.OLS(y, X).fit()

# Get the summary of the regression model

regression\_summary = model.summary()

# Display the regression summary

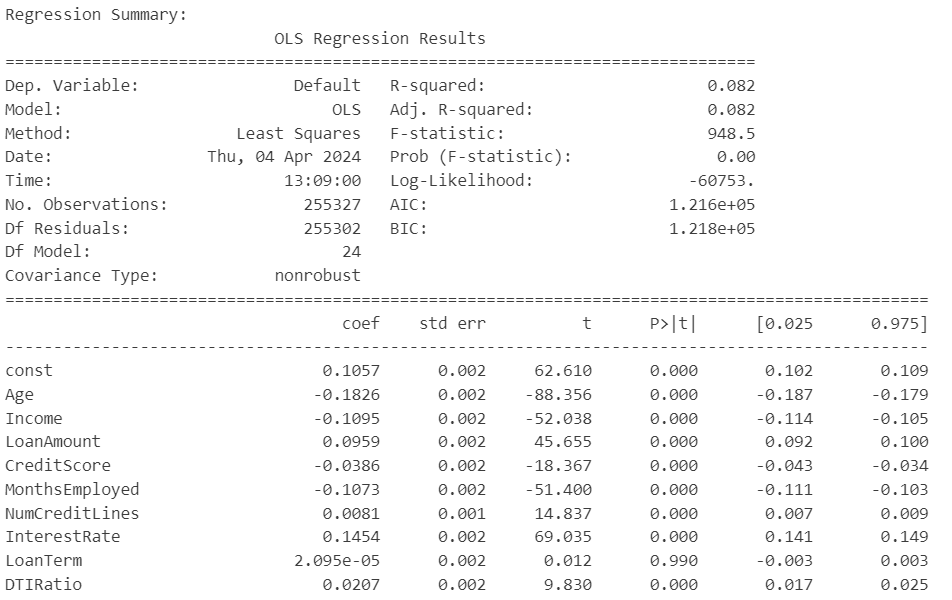
print("Regression Summary:")

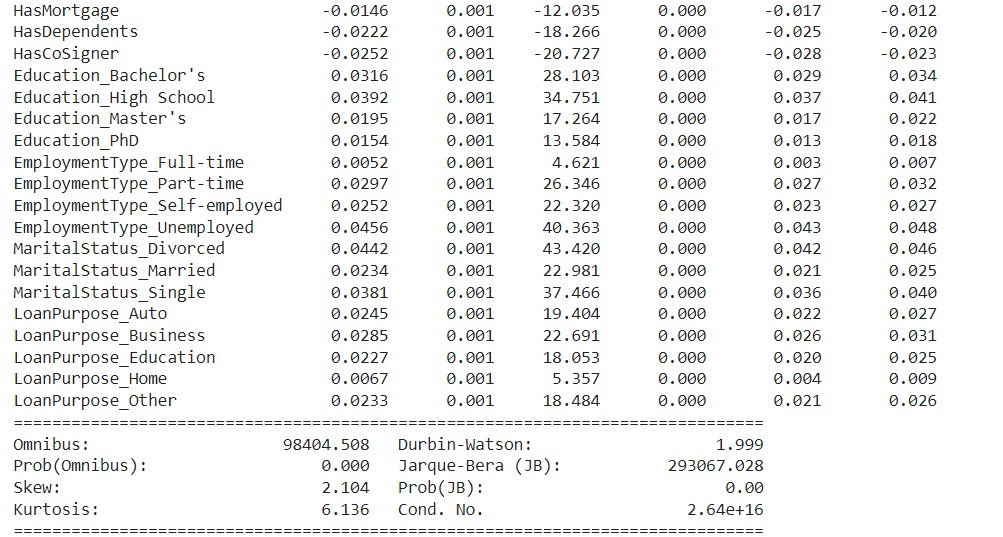
print(regression\_summary)

This code conducts multiple linear regression analysis using the statsmodels library in Python. Initially, a normalized dataset is loaded from a CSV file ('normalized\_data.csv') into a pandas data frame. The target variable for the regression analysis is specified (in this case, 'Default'). The data is then split into features (X) and the target variable (y). The statsmodels.api library is used to add a constant term to the features, which represents the intercept in the regression model.

Next, an Ordinary Least Squares (OLS) regression model is fitted to the data. Finally, the code generates a summary of the regression model, which includes statistical metrics such as coefficients, p-values, and R-squared value, providing insights into the relationships between the features and the target variable.

result:





After looking at the result, we have decided to remove LoanTerm from the variables.

Python code:

import pandas as pd

# Load the normalized DataFrame from the CSV file

normalized\_df = pd.read\_csv('normalized\_data.csv')

# Drop the 'LoanTerm' column

normalized\_df.drop(columns=['LoanTerm'], inplace=True)

# Save the modified DataFrame to a new CSV file

normalized\_df.to\_csv('BLA.csv', index=False)

This code will drop the ‘LoanTerm’ column and then save it into a new .csv file name ‘BLA.csv’.

**MODEL CREATION**

Afterwards, we began trying out different algorithms.

3.1 Logistics Regression

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import classification\_report, confusion\_matrix

# Load the modified DataFrame

normalized\_df = pd.read\_csv('BLA.csv')

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=['Default'])  # Features

y = normalized\_df['Default']  # Target variable

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize logistic regression classifier

classifier = LogisticRegression()

# Fit the classifier on the training data

classifier.fit(X\_train, y\_train)

# Make predictions on the test data probabilities

y\_prob = classifier.predict\_proba(X\_test)

# Manually adjust the threshold (for example, to 0.3)

threshold = 0.08

y\_pred\_adjusted = (y\_prob[:, 1] >= threshold).astype(int)

# Generate classification report

report = classification\_report(y\_test, y\_pred\_adjusted)

print("Classification Report with Threshold Adjustment:")

print(report)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_adjusted)

print("\nConfusion Matrix with Threshold Adjustment:")

print(conf\_matrix)

This code implements binary classification using logistic regression in Python, utilizing the scikit-learn library. It begins by loading a modified dataset from a CSV file ('BLA.csv') into a pandas data frame. Features (X) and the target variable (y) are defined, with the target variable being 'Default'. The dataset is then split into training and test sets using the train\_test\_split function, with 20% of the data allocated for testing and a specified random state for reproducibility.

A logistic regression classifier is initialized and fitted to the training data. Predictions are made on the test data probabilities using predict\_proba, and a threshold of 0.08 is manually applied to adjust the classification decision. Classification report and confusion matrix are generated using classification\_report and confusion\_matrix functions, providing evaluation metrics such as precision, recall, F1-score, and confusion matrix for assessing the performance of the classifier with the adjusted threshold.

result:

Classification Report with Threshold Adjustment:

precision recall f1-score support

0 0.95 0.51 0.67 45050

1 0.18 0.81 0.30 6016

accuracy 0.55 51066

macro avg 0.57 0.66 0.48 51066

weighted avg 0.86 0.55 0.62 51066

Confusion Matrix with Threshold Adjustment:

[[23118 21932]

[ 1136 4880]]

3.2 Logistics Regression with RandomizedSearchCV

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report, confusion\_matrix

from scipy.stats import uniform

from imblearn.over\_sampling import SMOTE

# Load data

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=["Default"])  # Features

y = normalized\_df["Default"]  # Target variable

# Apply standardization to the features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Handling Class Imbalance using SMOTE

smote = SMOTE(random\_state=42)

X\_resampled, y\_resampled = smote.fit\_resample(X\_scaled, y)

# Split the resampled data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_resampled, y\_resampled, test\_size=0.2, random\_state=42)

# Initialize Logistic Regression classifier

logreg = LogisticRegression()

# Define parameter distributions for RandomizedSearchCV

param\_dist = {

    'C': uniform(loc=0.01, scale=10)

}

# Initialize RandomizedSearchCV

random\_search = RandomizedSearchCV(estimator=logreg, param\_distributions=param\_dist, n\_iter=20, cv=5, scoring='f1', random\_state=42)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Get the best model

best\_logreg = random\_search.best\_estimator\_

# Evaluate model on validation set using best parameters

y\_pred\_val = best\_logreg.predict(X\_val)

print("Validation Set Performance with Best Parameters:")

print(classification\_report(y\_val, y\_pred\_val))

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_val, y\_pred\_val)

print("\nConfusion Matrix:")

print(conf\_matrix)

This code employs logistic regression for binary classification in Python, utilizing the scikit-learn library. It starts by loading data from a CSV file ('BLA.csv') into a pandas data frame. Rows with missing target values are then dropped from the data frame. Features (X) and the target variable (y) are defined accordingly. Standardization is applied to the features using StandardScaler. To address class imbalance, Synthetic Minority Over-sampling Technique (SMOTE) is employed. The resampled data is split into training and validation sets using train\_test\_split.

A Logistic Regression classifier is then initialized. Hyperparameters for logistic regression are tuned using RandomizedSearchCV with parameters distribution defined. The best parameters are determined, and the best model is selected. The model's performance is evaluated on the validation set using classification report and confusion matrix metrics, providing insights into precision, recall, F1-score, and the model's ability to correctly classify instances with different outcomes.

result:

Best Parameters: {'C': 3.7554011884736247}

Validation Set Performance with Best Parameters:

precision recall f1-score support

0 0.71 0.69 0.70 45188

1 0.69 0.72 0.71 45084

accuracy 0.70 90272

macro avg 0.70 0.70 0.70 90272

weighted avg 0.70 0.70 0.70 90272

Confusion Matrix:

[[30997 14191]

[12771 32313]]

3.3 Random Forest

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from imblearn.over\_sampling import SMOTE

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

# Load the modified DataFrame

normalized\_df = pd.read\_csv('BLA.csv')

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=['Default'])  # Features

y = normalized\_df['Default']  # Target variable

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize SMOTE for over-sampling only the minority class (positive class)

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

# Apply SMOTE to the training data only

X\_train\_resampled, y\_train\_resampled = smote.fit\_resample(X\_train, y\_train)

# Initialize Random Forest classifier

classifier = RandomForestClassifier(n\_estimators=100, random\_state=42)

# Fit the classifier on the resampled training data

classifier.fit(X\_train\_resampled, y\_train\_resampled)

# Make predictions on the test data

y\_pred = classifier.predict(X\_test)

# Generate classification report

report = classification\_report(y\_test, y\_pred)

print("Classification Report:")

print(report)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("\nConfusion Matrix:")

print(conf\_matrix)

This code implements binary classification using a Random Forest classifier in Python, employing the scikit-learn and imbalanced-learn libraries. Initially, a modified dataset is loaded from a CSV file ('BLA.csv') into a pandas data frame. Features (X) and the target variable (y) are defined accordingly. The data is then split into training and test sets using train\_test\_split function, with 20% of the data reserved for testing and a specified random state for reproducibility. SMOTE, a technique for over-sampling the minority class, is applied to the training data only to address class imbalance.

A Random Forest classifier is then initialized with 100 decision trees. The classifier is fitted to the resampled training data. Predictions are made on the test data, and a classification report is generated, providing metrics such as precision, recall, F1-score, and support for each class. Additionally, a confusion matrix is computed, displaying the counts of true positive, false positive, true negative, and false negative predictions.

result:

Classification Report:

precision recall f1-score support

0 0.91 0.85 0.88 45050

1 0.25 0.37 0.30 6016

accuracy 0.79 51066

macro avg 0.58 0.61 0.59 51066

weighted avg 0.83 0.79 0.81 51066

Confusion Matrix:

[[38226 6824]

[ 3774 2242]]

3.4 Random Forest with RandomizedSearchCV

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from sklearn.ensemble import RandomForestClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report

from scipy.stats import randint

import multiprocessing

# Load data

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=["Default"])  # Features

y = normalized\_df["Default"]  # Target variable

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize Random Forest classifier

rf = RandomForestClassifier()

# Define parameter distributions for RandomizedSearchCV

param\_dist = {

'n\_estimators': [int(x) for x in range(100, 1000, 100)],

'max\_depth': [None, 10, 20, 30, 40, 50],

'min\_samples\_split': [2, 5, 10],

'min\_samples\_leaf': [1, 2, 4],

'bootstrap': [True, False],

'criterion': ['gini', 'entropy']

}

# Initialize RandomizedSearchCV with parallel processing

n\_cores = multiprocessing.cpu\_count()

random\_search = RandomizedSearchCV(estimator=rf, param\_distributions=param\_dist, n\_iter=5, cv=3, scoring='f1', random\_state=42, n\_jobs=n\_cores)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Get the best model

best\_rf = random\_search.best\_estimator\_

# Evaluate model on validation set using best parameters

y\_pred\_val = best\_rf.predict(X\_val)

print("Validation Set Performance with Best Parameters:")

print(classification\_report(y\_val, y\_pred\_val))

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_val, y\_pred\_val)

print("\nConfusion Matrix:")

print(conf\_matrix)

This code utilizes Random Forest classification for predictive modeling in Python, employing the scikit-learn library. Initially, data is loaded from a CSV file ('BLA.csv') into a pandas data frame. Rows with missing target values are then dropped from the data frame. Features (X) and the target variable (y) are defined accordingly. The data is split into training and validation sets using the train\_test\_split function, with 20% of the data reserved for validation and a specified random state for reproducibility.

A Random Forest classifier is initialized with default hyperparameters. Hyperparameters are tuned using RandomizedSearchCV, with a predefined parameter space. The search is performed in parallel using the available CPU cores to speed up the process. The best parameters found are printed, and the best model is selected. The performance of the best model is evaluated on the validation set using classification report, providing metrics such as precision, recall, F1-score, and support for each class. Additionally, a confusion matrix is computed, displaying the counts of true positive, false positive, true negative, and false negative predictions.

result:

Best Parameters: {'n\_estimators': 200, 'min\_samples\_split': 2, 'min\_samples\_leaf': 1, 'max\_depth': None, 'criterion': 'entropy', 'bootstrap': False}

Validation Set Performance with Best Parameters:

precision recall f1-score support

0 0.89 1.00 0.94 45050

1 0.64 0.05 0.09 6016

accuracy 0.88 51066

macro avg 0.76 0.52 0.51 51066

weighted avg 0.86 0.88 0.84 51066

Confusion Matrix:

[[44885 165]

[ 5726 290]]

3.5 Random Forest with GridSearchCV

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split, GridSearchCV, StratifiedKFold

from imblearn.over\_sampling import SMOTE

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

# Load the modified DataFrame

normalized\_df = pd.read\_csv('BLA.csv')

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=['Default'])  # Features

y = normalized\_df['Default']  # Target variable

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize SMOTE for over-sampling only the minority class (positive class)

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

# Apply SMOTE to the training data only

X\_train\_resampled, y\_train\_resampled = smote.fit\_resample(X\_train, y\_train)

# Initialize Random Forest classifier

rf\_classifier = RandomForestClassifier(random\_state=42)

# Define hyperparameters for tuning

param\_grid = {

    'n\_estimators': [100, 200],

    'max\_depth': [5, 10],

    'min\_samples\_split': [2, 5],

    'min\_samples\_leaf': [1, 2]

}

# Define hyperparameters for tuning

param\_grid = {

    'n\_estimators': [100, 200],  # Reduced number of estimators

    'max\_depth': [5, 10],  # Reduced depth

    'min\_samples\_split': [2, 5],  # Reduced number of splits

    'min\_samples\_leaf': [1]  # Kept only one value for leaf samples

}

# Perform Grid Search with cross-validation

cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

grid\_search = GridSearchCV(estimator=rf\_classifier, param\_grid=param\_grid, cv=cv, scoring='f1', n\_jobs=-1)

grid\_search.fit(X\_train\_resampled, y\_train\_resampled)

# Get the best model from Grid Search

best\_rf\_model = grid\_search.best\_estimator\_

# Make predictions on the test data

y\_pred = best\_rf\_model.predict(X\_test)

# Generate classification report

report = classification\_report(y\_test, y\_pred)

print("Classification Report:")

print(report)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("\nConfusion Matrix:")

print(conf\_matrix)

This code applies Random Forest classification for predictive modeling in Python, utilizing the scikit-learn and imbalanced-learn libraries. It starts by loading a modified dataset from a CSV file ('BLA.csv') into a pandas data frame. Features (X) and the target variable (y) are defined accordingly, and the data is split into training and test sets using the train\_test\_split function. To address class imbalance, Synthetic Minority Over-sampling Technique (SMOTE) is applied to the training data only.

A Random Forest classifier is then initialized with default hyperparameters. The hyperparameters are tuned using Grid Search with cross-validation, defining a parameter grid to search over. Stratified k-fold cross-validation is employed to ensure class balance within each fold. The best model obtained from the grid search is used to make predictions on the test data. A classification report is generated, presenting metrics such as precision, recall, F1-score, and support for each class. Additionally, a confusion matrix is computed, displaying the counts of true positive, false positive, true negative, and false negative predictions, aiding in assessing the model's performance.

result:

Classification Report:

precision recall f1-score support

0 0.93 0.72 0.81 45050

1 0.22 0.59 0.32 6016

accuracy 0.70 51066

macro avg 0.57 0.65 0.56 51066

weighted avg 0.84 0.70 0.75 51066

Confusion Matrix:

[[32345 12705]

[ 2492 3524]]

3.6 Decision Tree with RandomizedSearchCV

Python code:

import pandas as pd

from sklearn.model\_selection import cross\_val\_score, RandomizedSearchCV, train\_test\_split

from imblearn.over\_sampling import SMOTE

from sklearn.metrics import classification\_report, confusion\_matrix

import multiprocessing

from sklearn.tree import DecisionTreeClassifier

# Load data

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=["Default"])  # Features

y = normalized\_df["Default"]  # Target variable

# Initialize Decision Tree classifier

dt = DecisionTreeClassifier()

# Apply SMOTE to the training data only

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

X\_resampled, y\_resampled = smote.fit\_resample(X, y)

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_resampled, y\_resampled, test\_size=0.2, random\_state=42)

# Define parameter distributions for RandomizedSearchCV

param\_dist = {

    'criterion': ['gini', 'entropy'],

    'max\_depth': [None, 10, 20, 30, 40, 50],

    'min\_samples\_split': [2, 5, 10],

    'min\_samples\_leaf': [1, 2, 4]

}

# Initialize RandomizedSearchCV with parallel processing

n\_cores = multiprocessing.cpu\_count()

random\_search = RandomizedSearchCV(estimator=dt, param\_distributions=param\_dist, n\_iter=20, cv=3, scoring='f1', random\_state=42, n\_jobs=n\_cores)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Evaluate model on validation set using best parameters

best\_dt = random\_search.best\_estimator\_

y\_pred\_val = best\_dt.predict(X\_val)

print("Validation Set Performance with Best Parameters:")

print(classification\_report(y\_val, y\_pred\_val))

This code uses a Decision Tree classifier for binary classification tasks in Python, utilizing the scikit-learn and imbalanced-learn libraries. Initially, data is loaded from a CSV file ('BLA.csv') into a pandas data frame, with rows containing missing target values dropped. Features (X) and the target variable (y) are then defined accordingly. The script proceeds to initialize a Decision Tree classifier. To address class imbalance, Synthetic Minority Over-sampling Technique (SMOTE) is applied to the entire dataset. The resampled data is then split into training and validation sets using the train\_test\_split function.

The hyperparameters for the Decision Tree classifier are tuned using RandomizedSearchCV with a predefined parameter distribution. The search is conducted in parallel using the available CPU cores. The best parameters found are printed, and the best model is selected. The performance of the best model is evaluated on the validation set using a classification report, providing metrics such as precision, recall, F1-score, and support for each class.

result:

Best Parameters: {'min\_samples\_split': 10, 'min\_samples\_leaf': 4, 'max\_depth': None, 'criterion': 'entropy'}

Validation Set Performance with Best Parameters:

precision recall f1-score support

0.0 0.86 0.87 0.86 27321

1.0 0.87 0.86 0.86 27287

accuracy 0.86 54608

macro avg 0.86 0.86 0.86 54608

weighted avg 0.86 0.86 0.86 54608

3.7 Decision Tree using AdaBoost Classifier

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import classification\_report

import multiprocessing

# Load data

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=["Default"])  # Features

y = normalized\_df["Default"]  # Target variable

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize base classifier (decision tree)

base\_classifier = DecisionTreeClassifier()

# Initialize AdaBoost Classifier with the base classifier

adaboost = AdaBoostClassifier(base\_estimator=base\_classifier)

# Define parameter distributions for RandomizedSearchCV

param\_dist = {

    'n\_estimators': [50, 100, 150, 200],

    'learning\_rate': [0.01, 0.05, 0.1, 0.5, 1.0],

}

# Initialize RandomizedSearchCV with parallel processing

n\_cores = multiprocessing.cpu\_count()

random\_search = RandomizedSearchCV(estimator=adaboost, param\_distributions=param\_dist, n\_iter=20, cv=3, scoring='f1', random\_state=42, n\_jobs=n\_cores)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Get the best model

best\_adaboost = random\_search.best\_estimator\_

# Evaluate model on validation set using best parameters

y\_pred\_val = best\_adaboost.predict(X\_val)

print("Validation Set Performance with Best Parameters:")

print(classification\_report(y\_val, y\_pred\_val))

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_val, y\_pred\_val)

print("\nConfusion Matrix:")

print(conf\_matrix)

This code employs AdaBoost, an ensemble learning method, for binary classification tasks in Python, utilizing the scikit-learn library. Initially, data is loaded from a CSV file ('BLA.csv') into a pandas data frame, with rows containing missing target values dropped. Features (X) and the target variable (y) are defined accordingly, and the data is split into training and validation sets using the train\_test\_split function.

An AdaBoost classifier is then initialized, with a decision tree as the base estimator. Hyperparameters for AdaBoost are tuned using RandomizedSearchCV with predefined parameter distributions. The search is performed in parallel using the available CPU cores. The best parameters found are printed, and the best model is selected. The performance of the best model is evaluated on the validation set using a classification report, providing metrics such as precision, recall, F1-score, and support for each class. Additionally, a confusion matrix is computed, displaying the counts of true positive, false positive, true negative, and false negative predictions, aiding in assessing the model's performance.

result:

Best Parameters: {'n\_estimators': 100, 'learning\_rate': 1.0}

Validation Set Performance with Best Parameters:

precision recall f1-score support

0 0.89 0.88 0.89 45050

1 0.20 0.23 0.21 6016

accuracy 0.80 51066

macro avg 0.55 0.55 0.55 51066

weighted avg 0.81 0.80 0.81 51066

Confusion Matrix:

[[39489 5561]

[ 4636 1380]]

3.8 Support Vector Machine (SVM)

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from imblearn.over\_sampling import SMOTE

from sklearn.svm import SVC

from sklearn.metrics import classification\_report, confusion\_matrix

# Load the modified DataFrame

normalized\_df = pd.read\_csv('BLA.csv')

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=['Default'])  # Features

y = normalized\_df['Default']  # Target variable

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize SMOTE for over-sampling only the minority class (positive class)

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

# Apply SMOTE to the training data only

X\_train\_resampled, y\_train\_resampled = smote.fit\_resample(X\_train, y\_train)

# Initialize Support Vector Machine (SVM) classifier

classifier = SVC(kernel='rbf', random\_state=42)

# Fit the classifier on the resampled training data

classifier.fit(X\_train\_resampled, y\_train\_resampled)

# Make predictions on the test data

y\_pred = classifier.predict(X\_test)

# Generate classification report

report = classification\_report(y\_test, y\_pred)

print("Classification Report:")

print(report)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

print("\nConfusion Matrix:")

print(conf\_matrix)

This script utilizes Support Vector Machine (SVM) classification for binary classification tasks in Python, leveraging the scikit-learn and imbalanced-learn libraries. Initially, a modified dataset is loaded from a CSV file ('BLA.csv') into a pandas data frame. Features (X) and the target variable (y) are defined accordingly, and the data is split into training and test sets using the train\_test\_split function. To address class imbalance, Synthetic Minority Over-sampling Technique (SMOTE) is applied to the training data only.

An SVM classifier with a radial basis function (RBF) kernel is then initialized. The classifier is fitted to the resampled training data. Predictions are made on the test data, and a classification report is generated, presenting metrics such as precision, recall, F1-score, and support for each class. Additionally, a confusion matrix is computed, displaying the counts of true positive, false positive, true negative, and false negative predictions, aiding in assessing the model's performance.

result:

Classification Report:

precision recall f1-score support

0 0.93 0.72 0.81 45050

1 0.22 0.58 0.31 6016

accuracy 0.70 51066

macro avg 0.57 0.65 0.56 51066

weighted avg 0.84 0.70 0.75 51066

Confusion Matrix:

[[32261 12789]

[ 2502 3514]]

3.9 Gradient Boosting with RandomizedSearchCV

Python code:

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from imblearn.over\_sampling import SMOTE

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

from scipy.stats import randint

import multiprocessing

# Load the modified DataFrame

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=['Default'])  # Features

y = normalized\_df['Default']  # Target variable

# Apply SMOTE to the training data only

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

X\_resampled, y\_resampled = smote.fit\_resample(X, y)

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_resampled, y\_resampled, test\_size=0.2, random\_state=42)

# Define hyperparameter grid

param\_grid = {

    'n\_estimators': randint(50, 200),

    'max\_depth': randint(3, 10),

    'learning\_rate': [0.1, 0.05, 0.01]

}

# Initialize Gradient Boosting classifier

classifier = GradientBoostingClassifier(random\_state=42)

# Initialize RandomizedSearchCV with parallel processing

n\_cores = multiprocessing.cpu\_count()

random\_search = RandomizedSearchCV(estimator=classifier, param\_distributions=param\_grid, n\_iter=20, cv=3, scoring='f1', random\_state=42, n\_jobs=n\_cores)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Make predictions on the validation data using the best model

best\_model = random\_search.best\_estimator\_

y\_pred = best\_model.predict(X\_val)

# Generate classification report

report = classification\_report(y\_val, y\_pred)

print("Classification Report:")

print(report)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_val, y\_pred)

print("Confusion Matrix:")

print(conf\_matrix)

This code utilizes the Gradient Boosting Classifier, a machine learning algorithm, for binary classification tasks in Python, employing the scikit-learn and imbalanced-learn libraries. Initially, a modified dataset is loaded from a CSV file ('BLA.csv') into a pandas data frame, with rows containing missing target values dropped. Features (X) and the target variable (y) are defined accordingly. Synthetic Minority Over-sampling Technique (SMOTE) is then applied to the training data only to address class imbalance. The resampled data is split into training and validation sets using the train\_test\_split function.

A hyperparameter grid is defined for the Gradient Boosting classifier, specifying ranges for the number of estimators, maximum depth, and learning rate. The RandomizedSearchCV function is employed to search for the best combination of hyperparameters using cross-validation. The best parameters found are printed, and the best model is selected. Predictions are made on the validation data using the best model, and a classification report and confusion matrix are generated to evaluate its performance, providing insights into precision, recall, F1-score, and support for each class, as well as the counts of true positive, false positive, true negative, and false negative predictions.

result:

Best Parameters: {'learning\_rate': 0.1, 'max\_depth': 7, 'n\_estimators': 152}

Classification Report:

precision recall f1-score support

0 0.88 0.94 0.91 45188

1 0.93 0.88 0.90 45084

accuracy 0.91 90272

macro avg 0.91 0.91 0.91 90272

weighted avg 0.91 0.91 0.91 90272

Confusion Matrix:

[[42340 2848]

[ 5523 39561]]

3.10 Naïve Bayes with RandomizedSearchCV

Python code:

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from imblearn.over\_sampling import SMOTE

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import classification\_report, confusion\_matrix

import multiprocessing

import pandas as pd

# Load the modified DataFrame

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=['Default'])  # Features

y = normalized\_df['Default']  # Target variable

# Apply SMOTE to the training data only

smote = SMOTE(sampling\_strategy='auto', random\_state=42)

X\_resampled, y\_resampled = smote.fit\_resample(X, y)

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X\_resampled, y\_resampled, test\_size=0.2, random\_state=42)

# Initialize Gaussian Naive Bayes classifier

classifier = GaussianNB()

# Fit the classifier to the training data

classifier.fit(X\_train, y\_train)

# Make predictions on the validation data

y\_pred = classifier.predict(X\_val)

# Generate classification report

report = classification\_report(y\_val, y\_pred)

print("Classification Report:")

print(report)

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_val, y\_pred)

print("Confusion Matrix:")

print(conf\_matrix)

This script utilizes the Multinomial Naive Bayes classifier for binary classification tasks in Python, employing the scikit-learn library. The modified dataset is loaded from a CSV file ('BLA.csv') into a pandas data frame, with rows containing missing target values dropped. Features (X) and the target variable (y) are defined accordingly, and the data is split into training and validation sets using the train\_test\_split function. The Multinomial Naive Bayes classifier is then initialized.

As Naive Bayes classifiers do not have hyperparameters to tune, RandomizedSearchCV is not used for hyperparameter tuning. The classifier is directly fitted to the training data. Predictions are made on the validation set, and a classification report is generated to evaluate the model's performance, providing insights into precision, recall, F1-score, and support for each class. Additionally, a confusion matrix is computed, displaying the counts of true positive, false positive, true negative, and false negative predictions, aiding in assessing the model's effectiveness.

result:

Classification Report:

precision recall f1-score support

0 0.74 0.71 0.72 45188

1 0.72 0.75 0.73 45084

accuracy 0.73 90272

macro avg 0.73 0.73 0.73 90272

weighted avg 0.73 0.73 0.73 90272

Confusion Matrix:

[[32151 13037]

[11356 33728]]

3.11 Neural Network with RandomizedSearchCV

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from sklearn.neural\_network import MLPClassifier

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report

import multiprocessing

# Load data

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=["Default"])  # Features

y = normalized\_df["Default"]  # Target variable

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize Multi-layer Perceptron (Neural Network) classifier

mlp = MLPClassifier()

# Define parameter distributions for RandomizedSearchCV

param\_dist = {

    'hidden\_layer\_sizes': [(100,), (50, 50), (100, 50), (50, 25)],

    'activation': ['logistic', 'tanh', 'relu'],

    'solver': ['lbfgs', 'sgd', 'adam'],

    'alpha': [0.0001, 0.001, 0.01, 0.1],

    'learning\_rate': ['constant', 'invscaling', 'adaptive']

}

# Initialize RandomizedSearchCV with parallel processing

n\_cores = multiprocessing.cpu\_count()

random\_search = RandomizedSearchCV(estimator=mlp, param\_distributions=param\_dist, n\_iter=20, cv=3, scoring='f1', random\_state=42, n\_jobs=n\_cores)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Get the best model

best\_mlp = random\_search.best\_estimator\_

# Evaluate model on validation set using best parameters

y\_pred\_val = best\_mlp.predict(X\_val)

print("Validation Set Performance with Best Parameters:")

print(classification\_report(y\_val, y\_pred\_val))

This script employs the Multi-layer Perceptron (MLP) classifier, also known as a neural network, for binary classification tasks in Python, utilizing the scikit-learn library. The modified dataset is loaded from a CSV file ('BLA.csv') into a pandas DataFrame, with rows containing missing target values dropped. Features (X) and the target variable (y) are defined accordingly, and the data is split into training and validation sets using the train\_test\_split function.

The MLP classifier is then initialized. Hyperparameters for the MLP classifier, such as the architecture of hidden layers, activation functions, solvers, regularization parameter (alpha), and learning rate schedule, are specified with a parameter grid for RandomizedSearchCV. RandomizedSearchCV is utilized with parallel processing to search for the best combination of hyperparameters. The best model obtained from RandomizedSearchCV is then evaluated on the validation set using the best parameters. A classification report is generated to evaluate the model's performance, providing insights into precision, recall, F1-score, and support for each class. Finally, the DataFrame is utilized to store and manipulate the dataset throughout the process.

result:

Best Parameters: {'solver': 'adam', 'learning\_rate': 'constant', 'hidden\_layer\_sizes': (50, 25), 'alpha': 0.0001, 'activation': 'tanh'}

Validation Set Performance with Best Parameters:

precision recall f1-score support

0 0.89 0.99 0.94 45050

1 0.50 0.10 0.17 6016

accuracy 0.88 51066

macro avg 0.70 0.54 0.55 51066

weighted avg 0.85 0.88 0.85 51066

3.12 Gradient Boosting Using Cat Boost Classifier

Python code:

import pandas as pd

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

from catboost import CatBoostClassifier

from sklearn.metrics import classification\_report

import multiprocessing

# Load data

normalized\_df = pd.read\_csv('BLA.csv')

# Check for and drop rows with missing target values

normalized\_df = normalized\_df.dropna(subset=['Default'])

# Define the features (X) and target variable (y)

X = normalized\_df.drop(columns=["Default"])  # Features

y = normalized\_df["Default"]  # Target variable

# Split the data into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize CatBoost Classifier

catboost = CatBoostClassifier()

# Define parameter distributions for RandomizedSearchCV

param\_dist = {

    'iterations': [100, 200, 300],

    'learning\_rate': [0.01, 0.05, 0.1],

    'depth': [4, 6, 8, 10],

    'l2\_leaf\_reg': [1, 3, 5, 7, 9],

    'border\_count': [32, 64, 128],

}

# Initialize RandomizedSearchCV with parallel processing

n\_cores = multiprocessing.cpu\_count()

random\_search = RandomizedSearchCV(estimator=catboost, param\_distributions=param\_dist, n\_iter=20, cv=3, scoring='f1', random\_state=42, n\_jobs=n\_cores)

# Fit RandomizedSearchCV on the training data

random\_search.fit(X\_train, y\_train)

# Print the best parameters found

print("Best Parameters:", random\_search.best\_params\_)

# Get the best model

best\_catboost = random\_search.best\_estimator\_

# Evaluate model on validation set using best parameters

y\_pred\_val = best\_catboost.predict(X\_val)

print("Validation Set Performance with Best Parameters:")

print(classification\_report(y\_val, y\_pred\_val))

# Generate confusion matrix

conf\_matrix = confusion\_matrix(y\_val, y\_pred)

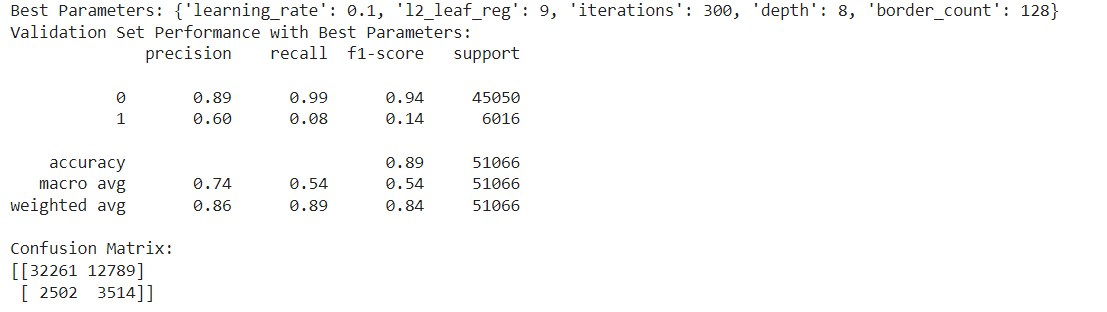
print("Confusion Matrix:")

print(conf\_matrix)

This script utilizes the CatBoost classifier, a gradient boosting algorithm, for binary classification tasks in Python, leveraging the CatBoost library. The dataset is loaded from a CSV file ('BLA.csv') into a pandas DataFrame, where rows containing missing target values are dropped. Features (X) and the target variable (y) are defined accordingly, and the data is split into training and validation sets using the train\_test\_split function.

The CatBoost classifier is then initialized. Hyperparameters for the CatBoost classifier, such as the number of iterations, learning rate, tree depth, L2 regularization coefficient, and border count, are specified with a parameter grid for RandomizedSearchCV. RandomizedSearchCV is employed with parallel processing to search for the best combination of hyperparameters. The best model obtained from RandomizedSearchCV is then evaluated on the validation set using the best parameters, and a classification report is generated to assess the model's performance, providing metrics such as precision, recall, F1-score, and support for each class. Finally, the DataFrame is utilized to store and manipulate the dataset throughout the process. Additionally, there's an attempt to print the confusion matrix, although the variable `conf\_matrix` is not defined correctly in this context.

result:



**Analysis**

In total, we had tested 12 algorithms with varying results. In the end, we could not run SVM with RandomizedSearchCV because it had taken over 12 hours with no result. Decision Tree with RandomizedSearchCV, Logistics Regression with RandomizedSearchCV and Naïve Bayes with RandomizedSearchCV are among the top performers of the 12 algorithms. However, it is Gradient Boosting with RandomizedSearchCV that is selected as the best algorithm with its high precision, high recall value, high f1-score, high accuracy and overall performance metrics.

**Conclusion**

Hence, we will be using Gradient Boosting with RandomizedSearchCV as our model in predicting the approval outcome of the 20 new applicant for their bank loan.

Python code:

# Load new loan applications data

new\_applicants = pd.read\_csv("NewApplicants.csv")

# Drop the 'Default' column if it exists in the new applicants dataset

if 'Default' in new\_applicants.columns:

    new\_applicants = new\_applicants.drop(columns=['Default'])

# Make predictions for new loan applications

new\_predictions = best\_model.predict(new\_applicants)

# Print predictions for new loan applications

print("Predictions for new loan applications:", new\_predictions)

Result:

Predictions for new loan applications: [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]

In conclusion, there will only be 2 successful applications among the 20 applicants.

**References**

1. Lecture slides

2. Garg, S. (2019, March 27). One Hot Encoding in Machine Learning. GeeksforGeeks. <https://www.geeksforgeeks.org/ml-one-hot-encoding/>

3. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (n.d.). sklearn.ensemble.GradientBoostingClassifier. scikit-learn. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html>