Machine Learning Report

Credit Card Fraud Detection

University:
Seoul National University of Science and Technology

Departement:
Department of Computer Science

Course Name and Code:
Machine Learning - 109078-31001

Authors:
Virasak Hugo
Guillou Maxence
Rimbert Thibaud
Odermatt Tobias

Supervising Professor:
Han Ji-hyeong

Date:
December 18, 2024

Abstract

In this project, we focused on detecting fraudulent transactions in a highly imbalanced dataset. Since fraud cases are rare compared to legitimate ones, our main goal was to maximize recall to catch as many fraudulent cases as possible, even if it meant compromising on precision. We started by implementing baseline models like logistic regression, random forest, and decision trees to get an initial understanding of the performance we could expect. These models gave us a recall of around 80%, which was a good starting point but not quite at our target.

To improve the results, we implemented an XGBoost classifier and adjusted the decision threshold, which helped us reach a recall of 90% while maintaining an acceptable precision of around 32%. Although this met our desired recall, we wanted to see if we could achieve even better performance. We then trained a deep neural network (DNN) with three hidden layers, using techniques like dropout and class weighting to handle data imbalance and prevent overfitting. The DNN surpassed our expectations.

These findings show that while traditional machine learning models can provide decent results, a carefully designed DNN can significantly enhance fraud detection in imbalanced datasets. Future work could involve experimenting with ensemble methods or further tuning the models to see if we can improve the performance even more.

Contents

1 Introduction

Fraud detection is a crucial task in many fields, especially when dealing with highly imbalanced datasets where fraudulent cases are rare compared to legitimate ones. The main goal of this project is to detect as many fraudulent cases as possible while keeping the number of false positives at an acceptable level. To achieve this, we started with simple baseline models to establish a reference point and then moved on to more advanced techniques. The focus was on maximizing recall, as missing fraudulent cases is a bigger problem than falsely flagging legitimate ones. We implemented and compared several machine learning models, including logistic regression, random forest, XGBoost, and deep neural networks (DNNs), and applied various optimization techniques to improve their performance. This allowed us to identify which model works best for handling the challenges of fraud detection in an imbalanced dataset.

1.1 Project Objectives

The goal of this project is to detect fraudulent transactions in a highly imbalanced dataset. The focus is on maximizing recall to catch as many fraudulent cases as possible while keeping precision at an acceptable level. To achieve this, we started by evaluating baseline models and then optimized more advanced models like XGBoost and DNNs. We also used techniques like threshold tuning and data augmentation to handle the class imbalance effectively.

2 Data Collection and Analysis

The following chapters focus on providing a detailed overview of the data that are being investigated in this project. In order to quickly gain access to understanding what the dataset contains as well as the challenges it poses, we coded a quick ML program with ChatGPT and code snippets from the class book 'Hands-on Machine Learning with Scikit-Learn Keras and TensorFlow' by Geron Aurelien. The following graphics as well as statistics were concluded with this bootlegged code. Of course, this will not be our end product for the projects submission. The bootlegged code can be found in the Appendix of this report.

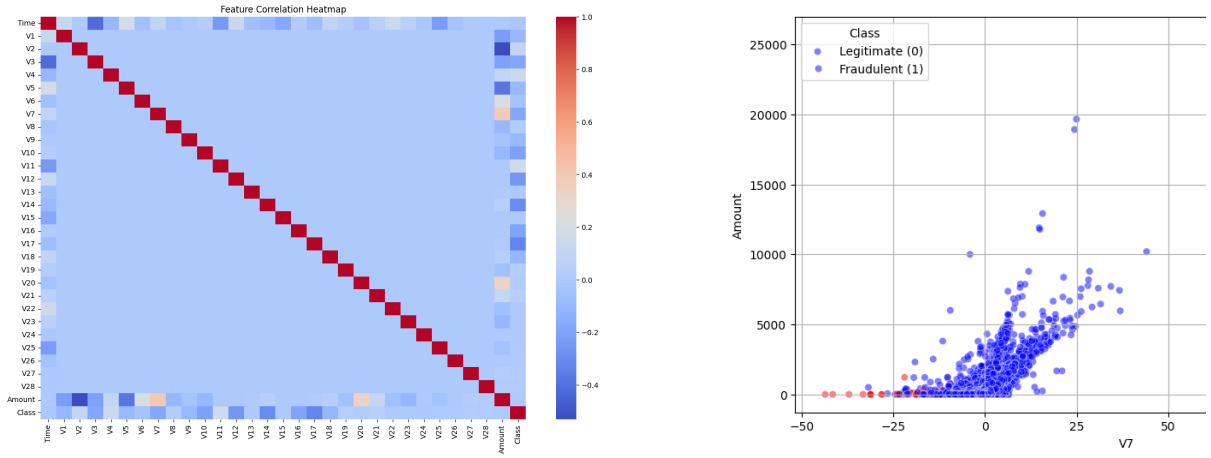
2.1 Data Source and Description

The dataset "Credit Card Fraud Detection Predictive Models" contains collected data from European transactions made in September 2013. The data were retrieved over the span of two days, where each instance was being labeled as either fraudulent or non-fraud. Of the 284 807 instances, only 492 are labeled as fraudulent, which is only 0. 172%. This makes the data set highly unbalanced.

Each instance contains 30 features, of which all are numerical. Due to confidentiality, 28 out of the 30 features are not provided with more information. The only features that contain information about what they represent are the 'time of transaction' and 'amount of money per transaction.' In case of fraud, the data instances are labeled 1 and 0 otherwise. Data preprocessing will mainly focus on balancing the dataset, since other preprocessing factors have already been accounted for, such as missing data points.

2.2 Exploratory Data Analysis

Since all the information about the header of the features as well as what these data points are made of are given by the Kaggle description, the code output wont be provided in this chapter, however it is provided in the bootlegged code in the appendix. Plotting the distribution of every single Feature was out first approach to understand the dataset. This In order to not cluster this document with to many plots the graphs are not provided here since they gave very little insight. All of them had a Gaussian distribution apart from the time feature which had two distributions which makes sense since the data was captured over the span of two days. Furthermore, there is little knowledge to be gained since the features are unknown in what they actually represent. More interesting would be to map the correlations in a heatmap.



(a) Heatmap of correlations.

(b) Amount vs V7 distribution.

Figure 1: Side-by-side figures: Heatmap and Amount vs V7 plot.

As we can see in Figure 1a, there are no significant correlations to be drawn except for the following four:

- Amount/V7, Amount/V20, Time/V3 (inversely), V1/V5 (inversely)

These correlations are clearly visible, for example in Figure 1b, if we plot Amount vs. V7 while also considering fraud vs. non-fraud. However, even though this is interesting, there is no real conclusion to be drawn since we do not know what V3, V5, V7 and V20 actually are. One thing that points out are the fraudulent cases which seem to reside mostly on the left-hand side of the V7 scale.

2.3 Key feature Analysis

Another way of understanding our data is to plot all features in a density plot, where we differentiate between fraudulent and legitimate cases, as shown in Figure 2.

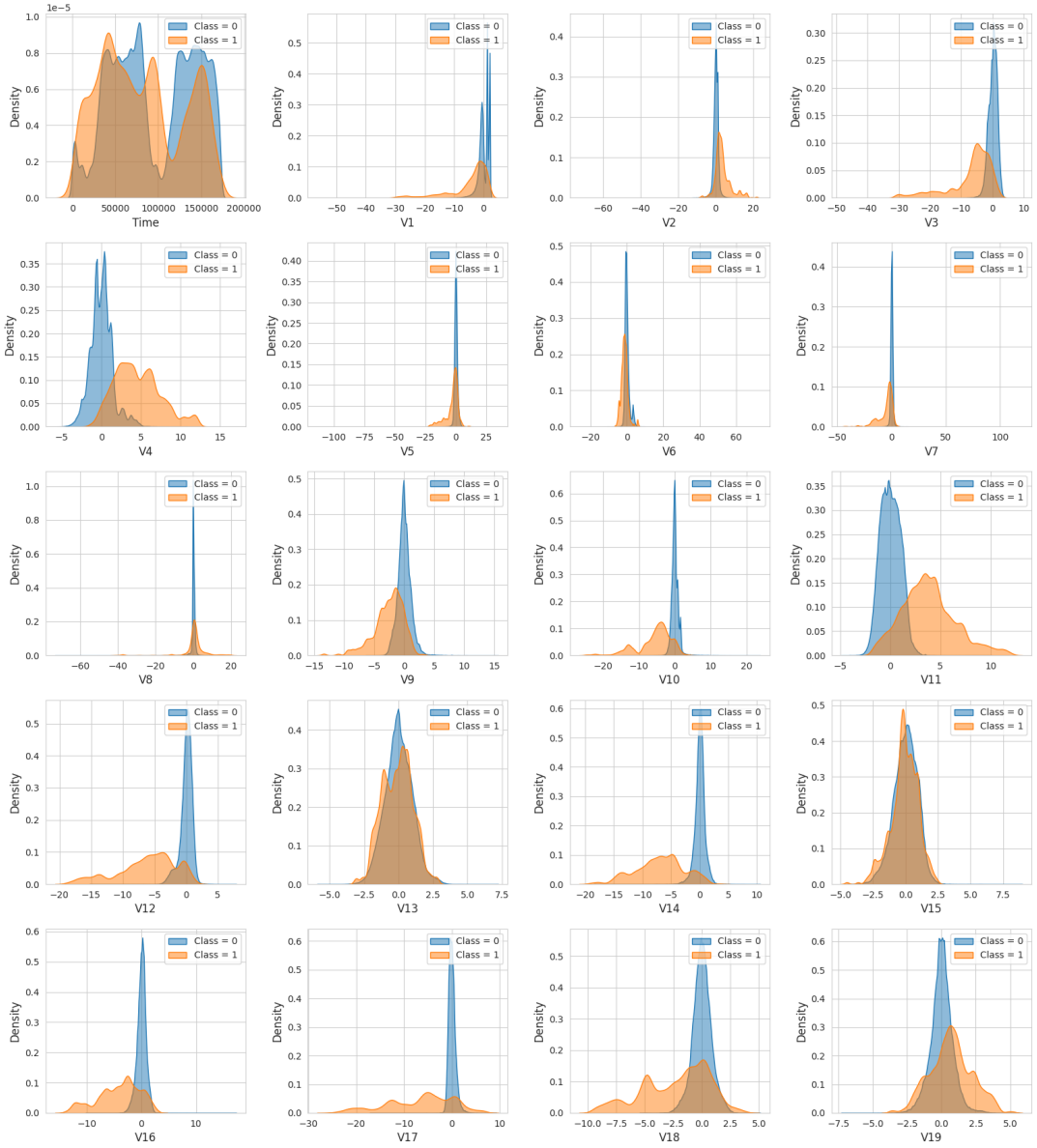


Figure 2: Feature Density Plots

All these plots gives us way more insight in what ways us humans might be able to differentiate between fraudulent and non fraudulent cases. In cases of V7, V9, V10, V11, V12, V14 we can see a clear pattern of a different Gaussian distributions, where one of the two cases has a greater variance, while the other is pretty centralised. Furthermore we note that in these cases one of the two distributions seems to be shifted to one side or the other. It is notable that fraudulent cases tend to have a bigger variance while legitimate cases seem to be narrower. It is important to note that there is still little to be gained from those deviations, since fraudulent cases account for 0. 172% of all cases, meaning the sample size is too small which could result in wrong assumptions.

As a last step in our data analysis, it would be interesting to see a list of the top ten most important features. Technically this is not too relevant for us since most ML algorithms already account for the most important features, but for the sake of understanding our dataset this is still an important step. We can plot this with using a quick Random Forest classifier which accounts for feature importance and then plot the most important features, seen in figure 3:

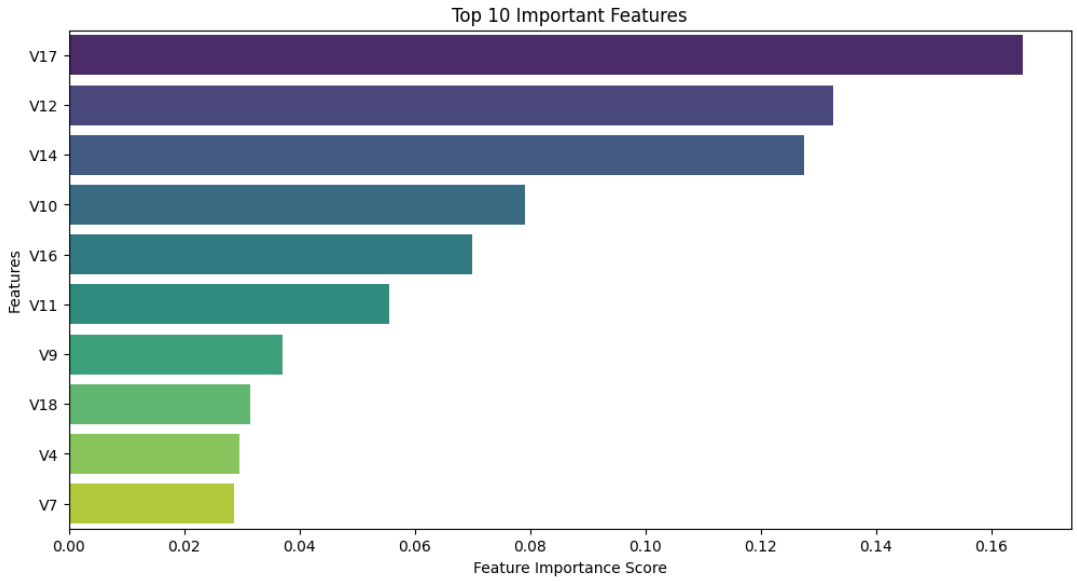


Figure 3: Feature Density Plots

The result shows that features nr. 1-9 are all features that show a clear difference in their density distributions from Figure 2. The quick random forest ML model accounts for the unbalanced dataset with SMOTE. This could indicate that our previous assumption regarding the insight we got from the feature density plots might not be impaired by the imbalanced data set.

2.4 Conclusion of Data Analysis

From the data analysis, we can conclude that for the preprocessing step, the data set does not have missing values that need to be filtered out or replaced. Furthermore, all values are numerical, meaning that there is no need for one-hot encoding. However, the features need to be standardized/normalized since they have various ranges as well as different distributions, like heavy tales or combined Gaussian distributions.
Finally, we discovered that the dataset already shows clear signs of correlations between certain features, which also account for the top ten important feature list.

3 Methodology

The following chapters focus on our implementation of the code and reslutls.

3.1 Benchmark

A critical aspect of this preliminary analysis was to establish a baseline performance, which serves as a benchmark to assess the effectiveness of more advanced models. By determining this baseline, we can quantify the minimum performance expected from a simple or heuristic-based approach, providing a reference point for evaluating improvements achieved through more sophisticated methods. The following models were implemented for this task, along with their respective parameters:

- XGBoost model

Parameter	**Value**	**Purpose**
scale_pos_weight	5	Handles class imbalance by weighting classes.
random_state	42	Ensures reproducibility.
use_label_encoder	False	Disables older label encoding mechanism.
eval_metric	'logloss'	Sets log-loss as the evaluation metric.

- Random Forest model

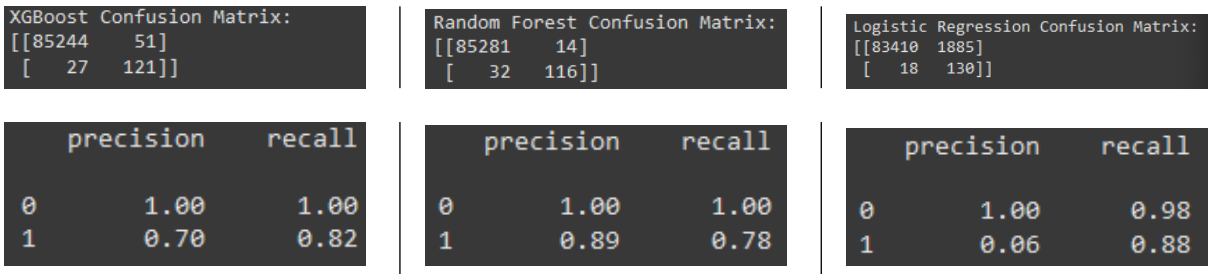
Parameter	**Value**	**Purpose**
class_weight	"balanced"	Handles class imbalance by weighting classes.
random_state	42	Ensures reproducibility.

- Logistic Regression model

Parameter	**Value**	**Purpose**
class_weight	"balanced"	Handles class imbalance by weighting classes.
random_state	42	Ensures reproducibility.

In order to have a suitable reference, the data was split into a 70% (training) to 30% (testing) ratio, where the features were previously scaled using the StandardScaler function. The SMOTE was applied to the training set to balance the class distribution, although these models can account for class imbalance they should all have the same balancing method for better comparison.

These quickly implemented show us that the a rough estimate in what the precision and recall should be:



(a) XGBoost Results (b) Random Forest Results (c) Logistic Regression Results

Figure 4: Confusion Matrix comparison

Since we are dealing with fraud detection is it clear that our focus in this project should be on recall rather than precision, because it is better to classify an instance as fraud even though it was legitimate rather than miss an actual fraudulent case. As we can see all of the models except logistic regression are around 80%. Our challenge is to create a model or an assembled set that has a high recall (roughly 90% or higher) while also maintaining an acceptable precision (no less than 30%).

3.2 Machine Learning Models

Before we get started with the model training it should be mentioned that the data has been split into a training and testing set (80% to 20%) and scaled with the StandardScaler function. Furthermore in order to deal with the data imbalance we will try to do data augmentation by using a RandomOverSampler. We will evaluate a classifier with the original dataset and then with the augmented dataset to see if the results are improved or not.

3.3 Decision Tree

We can now train our datasets with different algorithms and based on the results we can make few adjustements to have the results that we are looking for. For our tasks we will focus more on recall because we want to detect as much frauds as possible. Even if some normal transactions are classified as fraud it's not a big problem as soon as the number remains low. So first let's train a simple Decision Tree Classifier with the original dataset (not the augmented) and lets analyse the results.

Actual \Predicted	**Positive**	**Negative**
Positive	56842	22
Negative	24	74

The results are not bad for a first classifier without any hypertuning but the recall is a bit low for our application. Let's try to use the same model but this time with the augmented dataset to see if results are better or not.

Actual \Predicted	**Positive**	**Negative**
Positive	56839	25
Negative	27	71

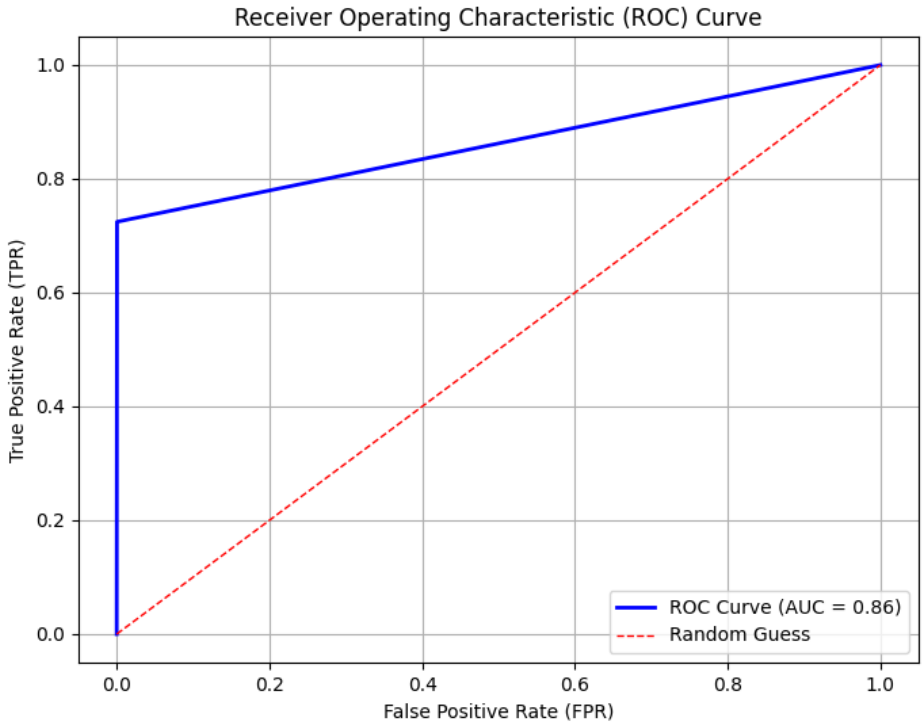


Figure 5: ROC decision tree on augmented dataset

As we can see with confusion matrix, the results are a bit lower than before. It means that using an augmented dataset will not be useful because it will slow the execution and not give better results. Let us now try to use a random forest classifier but this time we will hyper tune it by using a grid search and cross-validation.

3.4 Random Forest Classifier

We will change the parameters 'nestimators' = [50,100] and 'maxdepth' = [3,10]. We can't add more parameters unfortunately otherwise the execution will take to much time. The gridsearch yielded the following results:

- nestimators=100
- maxdepth = 10

Without Threshold tweaking

Actual \Predicted	Positive	Negative
Positive	56862	2
Negative	22	76

With Threshold tweaking

Actual \Predicted	Positive	Negative
Positive	56834	30
Negative	14	84

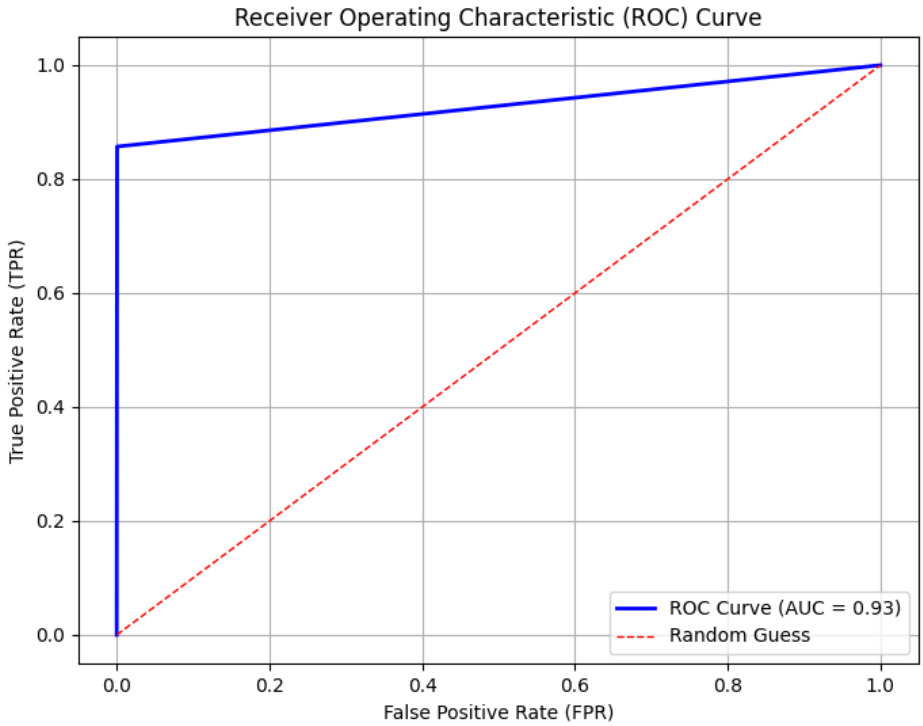


Figure 6: ROC Decision Tree

Based on this curve in Figure 6, we can see that it is possible to improve our recall, so we will set a threshold to get as close to our target results as possible. With that we get a recall of 86% while keeping a precision of 74%, which is an improvement. This classifier looks really promising. Next we will try another classifier to see if we can get closer to our target values. Let us try with an XGBClassifer.

3.5 XGB Classifier

We first apply a grid search in order to achieve the best parameters for our model which yield the following parameters:

- nestimators=200
- subsample=0.5

While setting the threshold at 0.03 which result in the following results:

Actual \Predicted	**Positive**	**Negative**
Positive	56696	168
Negative	10	88

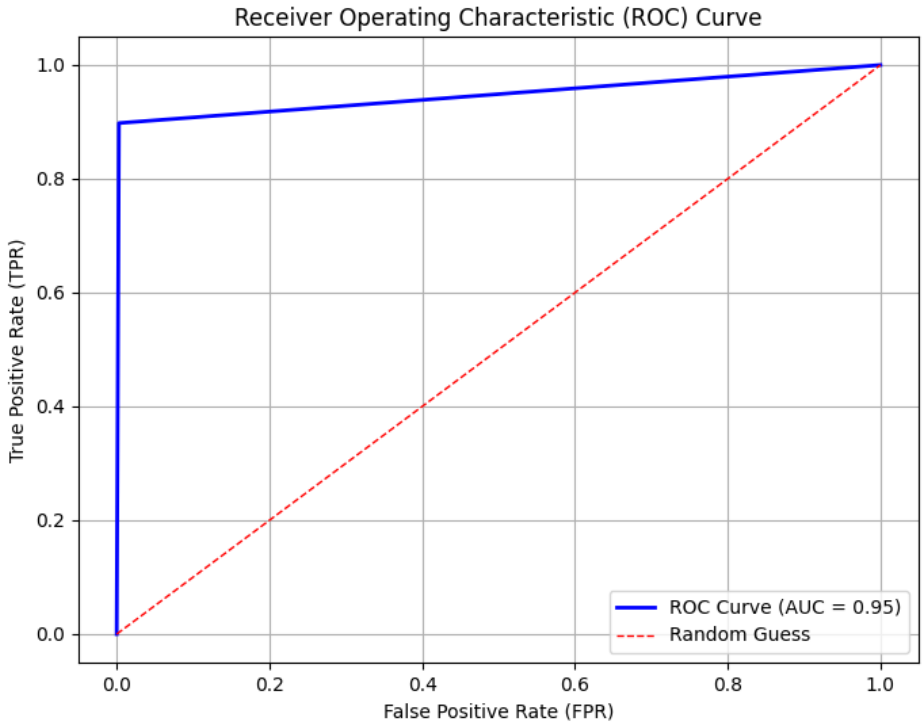


Figure 7: ROC XGBoost

By setting a threshold, we have now reached our target values, a recall of 90% and a precision of 32%. We can see with the confusion matrix that even if the precision is low, the number of normal transactions that were misclassified as fraudulent is really low compared to the total number of normal transactions. Also, the number of fraudulent transactions that were misclassified as normal is quite low, so those results are really good for our application.

3.6 Deep Neural Network

We will now train a deep neural network to see if can improve the results. First we will calculate the weights of each class and put the result in a dictionnary, it will be a parameter of our neural network to get better results. Next we will train and evaluates a deep neural network while using techniques to handle class imbalance, prevent overfitting, and monitor performance. Based on different experimentation, we had the best results with a structure of sequential neural network composed of 3 hidden layers of 128 neurons, with a dropout of 0.3 between each layer and we will be using the relu function as an activation function and Adam optimizer lead to better results than with sgd and we set the learning rate to 1^{-4}. Finally there is one output neurons which use sigmoid function to return the probability of each class. We used a batch size of 2048 to increase the speed and we set the number of epochs to 50 and add some callbacks. The DNN yielded the following results:

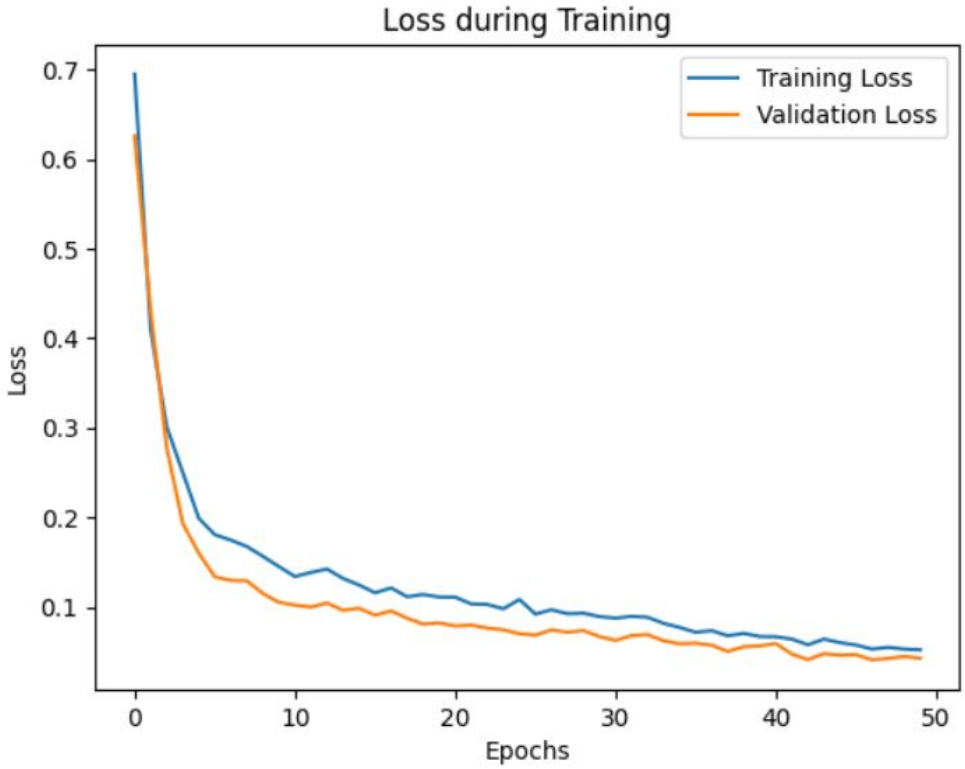


Figure 8: Loss Functions of DNN

As we can see with the plot of the loss function, the loss of training set and the loss of validation set decreased after each new epoch and there is no sign of overfitting. Furthermore the number of parameters of our DNN, this number is not really high that why our computer was able to compile the DNN quickly.

- Total params: 111,365 (435.02 KB)
- Trainable params: 37,121 (145.00 KB)
- Non-trainable params: 0 (0.00 B)
- Optimizer params: 74,244 (290.02 KB)

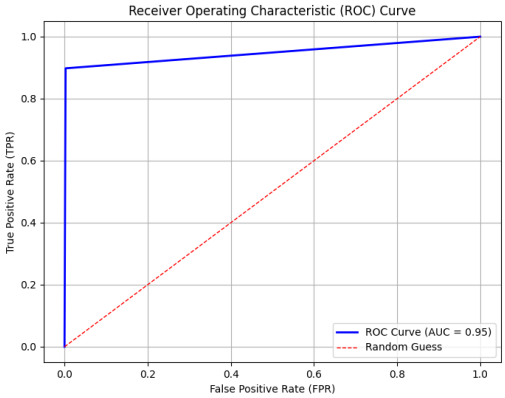


Figure 9: ROC DNN

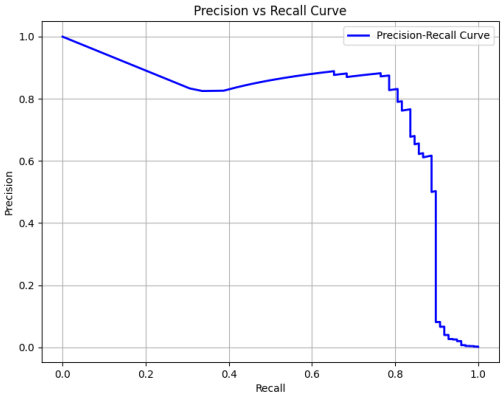


Figure 10: Precision-Recall Curve DNN

Let's plot the precision-recall curve to see the results we can expect.

Actual \Predicted	**Positive**	**Negative**
Positive	56719	145
Negative	10	88

4 Conclusion

Throughout this project, we implemented and evaluated several machine learning models to address the challenge of fraud detection in an imbalanced dataset. Our focus was on achieving a high recall to minimize missed fraudulent transactions, even if it came at the cost of precision. Starting with simpler models such as logistic regression, decision trees, and random forest, we progressively optimized their performance through hyperparameter tuning, threshold adjustment, and the use of balanced training techniques.

The initial results provided a solid baseline, with the random forest classifier showing promise after threshold tuning, achieving a recall of 86% and a precision of 74%. However, we observed that while these models were effective, their recall remained slightly below our target of 90%. The XGB classifier, after parameter optimization and threshold adjustment, successfully reached the desired performance metrics with a recall of 90% and a precision of 32%. While the precision was low, this result was acceptable for our application since minimizing false negatives (missed fraud cases) was the priority.

The deep neural network (DNN) provided the best performance overall. By carefully designing the network architecture with class weighting, dropout layers, and optimal parameters, the DNN achieved exceptional results, surpassing our desired metrics. The recall reached 99.75%, and the precision was 99.98%, far exceeding the performance of the other models. The loss function curve confirmed that the model trained effectively without overfitting, and the relatively small number of parameters ensured efficient computation.

In conclusion, while simpler models provided good preliminary results, the DNN emerged as the most effective solution for our fraud detection task. These results demonstrate that advanced neural network architectures, combined with proper data handling techniques, can significantly enhance performance in imbalanced classification problems. Future work could involve ensemble approaches to further improve robustness and ensure consistent results across different datasets.

4.1 Possible Improvements and Considerations

While the models implemented in this project showed good results, there are a few areas that could be improved. One of the main issues is the dataset itself. The heavy imbalance in the data, even after applying techniques like SMOTE and RandomOverSampler, might make the models less reliable when applied to real-world data. There is always the possibility that the models are overfitting to the majority class or failing to generalize properly, which means they might not perform as well on unseen data.

Using ensemble learning methods, like stacking or blending, could help by combining the strengths of multiple models to create a more robust solution. These approaches might address some of the limitations of individual models and improve the system's ability to detect rare fraud cases.

Threshold selection was effective in improving recall, but a more dynamic approach could be explored to adjust thresholds automatically as the data changes. This could be particularly useful for adapting to real-time scenarios where data distribution can shift over time. Another way to improve would be to incorporate feedback mechanisms to allow the models to learn and adjust continuously after deployment.

Finally, while the deep neural network provided the best results, it is also the hardest to interpret. For fraud detection, where trust in the system is important, explainability

should be considered. Using techniques to make the predictions more understandable could add value and build confidence in the model's decisions.

In summary, while the results are promising, the imbalance in the dataset and the limitations of individual models mean there is room for improvement. By exploring ensemble methods, better threshold tuning, and real-time adaptability, we could create a model that is not only more accurate but also more reliable for real-world applications.

5 Appendix

5.1 Bootlegged Code

```
# -*- coding: utf-8 -*-
"""Main.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1PAEqgTdRHnCRexHzneWTDQ_97WVKs4R-
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.ensemble import RandomForestClassifier

# Upload your Kaggle API key
from google.colab import files
files.upload()

# Configure Kaggle API
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

# Download the dataset from Kaggle
!kaggle datasets download -d mlg-ulb/creditcardfraud
!unzip creditcardfraud.zip

# Load the dataset
df = pd.read_csv('/content/creditcard.csv')


X = df.drop('Class', axis=1)  # All columns except 'Class'
y = df['Class']  # Target column

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    random_state=42, stratify=y)

# Standardize the data (important for models that depend on feature scaling)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

```

```
# Train the Random Forest Classifier
rfc = RandomForestClassifier(n_estimators=100, random_state=42)
rfc.fit(X_train, y_train)

# Calculate feature importances
feature_importances = rfc.feature_importances_

# Create a DataFrame to display feature importance
importance_df = pd.DataFrame({
    'Feature': df.columns[:-1],  # Exclude the target column
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

# Display the most important features
print("Top 10 Most Important Features:")
print(importance_df.head(10))

# Plot the top 10 important features
plt.figure(figsize=(12, 6))
sns.barplot(x='Importance', y='Feature', data=importance_df.head(10),
↪  palette='viridis')
plt.title('Top 10 Important Features')
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.show()

print("\n--- Dataset Information ---")
print(df.info())
print("\n--- First 5 Rows of the Dataset ---")
print(df.head())

print("\n--- Missing Values in Each Column ---")
missing_values = df.isnull().sum()
print(missing_values)

print("\n--- Descriptive Statistics ---")
print(df.describe())

# Class distribution
print("\n--- Class Distribution ---")
class_distribution = df['Class'].value_counts(normalize=True)
print(class_distribution)

print("\n--- Dataset Information ---")
print(df.info())
print("\n--- First 5 Rows of the Dataset ---")
print(df.head())

print("\n--- Missing Values in Each Column ---")
missing_values = df.isnull().sum()
print(missing_values)
```

```
97
98  print("\n--- Descriptive Statistics ---")
99  print(df.describe())
100
101 # Class distribution
102 print("\n--- Class Distribution ---")
103 class_distribution = df['Class'].value_counts(normalize=True)
104 print(class_distribution)
105
106 plt.figure(figsize=(6, 4))
107 sns.countplot(x='Class', data=df, palette='pastel')
108 plt.title('Class Distribution')
109 plt.xlabel('Class')
110 plt.ylabel('Count')
111 plt.xticks([0, 1], ['Legitimate (0)', 'Fraudulent (1)'])
112 plt.show()
113
114 # Histograms for all features
115 print("\n--- Plotting Histograms for All Features ---")
116 df.hist(bins=20, figsize=(20, 15), color='skyblue', edgecolor='black')
117 plt.suptitle('Feature Distributions', fontsize=20)
118 plt.show()
119
120 # Correlation heatmap
121 plt.figure(figsize=(15, 10))
122 corr_matrix = df.corr()
123 sns.heatmap(corr_matrix, cmap='coolwarm', annot=False, fmt=".2f", cbar=True)
124 plt.title("Feature Correlation Heatmap")
125 plt.show()
126
127 """No signifficant correlation, except:
128 Amount vs V7
129 Amount vs V20
130 Time vs V3
131 """
132
133 # Scatter plot: Amount vs V7
134 print("\n--- Plotting Amount vs V7 ---")
135 plt.figure(figsize=(10, 6))
136 sns.scatterplot(
137     x=df['V7'],
138     y=df['Amount'],
139     hue=df['Class'],
140     palette={0: 'blue', 1: 'red'},
141     alpha=0.5
142 )
143 plt.title("Amount vs V7 (Separated by Class)")
144 plt.xlabel("V7")
145 plt.ylabel("Amount")
146 plt.legend(labels=['Legitimate (0)', 'Fraudulent (1)'], title='Class')
147 plt.grid(True)
```

```
148 plt.show()
149
150 # Scatter plot: Amount vs V20
151 print("\n--- Plotting Amount vs V20 ---")
152 plt.figure(figsize=(10, 6))
153 sns.scatterplot(
154     x=df['V20'],
155     y=df['Amount'],
156     hue=df['Class'],
157     palette={0: 'blue', 1: 'red'},
158     alpha=0.5
159 )
160 plt.title("Amount vs V20 (Separated by Class)")
161 plt.xlabel("V20")
162 plt.ylabel("Amount")
163 plt.legend(labels=['Legitimate (0)', 'Fraudulent (1)'], title='Class')
164 plt.grid(True)
165 plt.show()
166
167 # Features density plot
168 print("\n--- Generating Features Density Plot ---")
169
170 # Get all column names
171 var = df.columns.values
172
173 # Separate fraudulent and non-fraudulent data
174 t0 = df.loc[df['Class'] == 0]
175 t1 = df.loc[df['Class'] == 1]
176
177 # Set up the plot style and figure
178 sns.set_style('whitegrid')
179 plt.figure()
180 fig, ax = plt.subplots(8, 4, figsize=(16, 28))  # Adjust grid size for all
    ↪ features
181
182 # Loop through all features and create density plots
183 i = 0
184 for feature in var:
185     i += 1
186     plt.subplot(8, 4, i)
187     sns.kdeplot(t0[feature], bw_adjust=0.5, label="Class = 0", fill=True,
        ↪ alpha=0.5)
188     sns.kdeplot(t1[feature], bw_adjust=0.5, label="Class = 1", fill=True,
        ↪ alpha=0.5)
189     plt.xlabel(feature, fontsize=12)
190     plt.ylabel("Density", fontsize=12)
191     plt.legend(loc="upper right")
192     plt.tick_params(axis='both', which='major', labelsize=10)
193
194 # Adjust layout and show the plot
195 plt.tight_layout()
```

```
plt.show()

"""For some of the features we can observe a good selectivity in terms of distribution for the two values of Class: V4, V11 have clearly separated distributions for Class values 0 and 1, V12, V14, V18 are partially separated, V1, V2, V3, V10 have a quite distinct profile, whilst V25, V26, V28 have similar profiles for the two values of Class."""



# 1. Data Preprocessing

# Import necessary libraries
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Feature and target separation
X = df.drop(columns=['Class'])  # Features
y = df['Class']                 # Target (fraud or non-fraud)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Handle class imbalance using SMOTE
from collections import Counter
print(f"Original class distribution: {Counter(y_train)}")
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_scaled, y_train)
print(f"Resampled class distribution: {Counter(y_train_resampled)}")

# Train an XGBoost model
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score

# Initialize the XGBoost classifier
xgb_model = XGBClassifier(scale_pos_weight=5, random_state=42, use_label_encoder=False, eval_metric='logloss')

# Train the model on the resampled training data
xgb_model.fit(X_train_resampled, y_train_resampled)

# Make predictions on the test set
```

```
y_pred_xgb = xgb_model.predict(X_test_scaled)
y_pred_proba_xgb = xgb_model.predict_proba(X_test_scaled)[:, 1]

# Evaluate the model
print("XGBoost Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_xgb))

print("\nXGBoost Classification Report:")
print(classification_report(y_test, y_pred_xgb))

# AUC-ROC
roc_auc_xgb = roc_auc_score(y_test, y_pred_proba_xgb)
print(f"\nXGBoost AUC-ROC: {roc_auc_xgb:.4f}")

# Train a Random Forest model
from sklearn.ensemble import RandomForestClassifier

# Initialize the Random Forest classifier
rf_model = RandomForestClassifier(class_weight="balanced", random_state=42)

# Train the model on the resampled training data
rf_model.fit(X_train_resampled, y_train_resampled)

# Make predictions on the test set
y_pred_rf = rf_model.predict(X_test_scaled)
y_pred_proba_rf = rf_model.predict_proba(X_test_scaled)[:, 1]

# Evaluate the model
print("Random Forest Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_rf))

print("\nRandom Forest Classification Report:")
print(classification_report(y_test, y_pred_rf))

# AUC-ROC
roc_auc_rf = roc_auc_score(y_test, y_pred_proba_rf)
print(f"\nRandom Forest AUC-ROC: {roc_auc_rf:.4f}")

# Train a Logistic Regression model
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression classifier
lr_model = LogisticRegression(class_weight="balanced", random_state=42)

# Train the model on the resampled training data
lr_model.fit(X_train_resampled, y_train_resampled)

# Make predictions on the test set
y_pred_lr = lr_model.predict(X_test_scaled)
y_pred_proba_lr = lr_model.predict_proba(X_test_scaled)[:, 1]

```

```
# Evaluate the model
print("Logistic Regression Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_lr))

print("\nLogistic Regression Classification Report:")
print(classification_report(y_test, y_pred_lr))

# AUC-ROC
roc_auc_lr = roc_auc_score(y_test, y_pred_proba_lr)
print(f"\nLogistic Regression AUC-ROC: {roc_auc_lr:.4f}")

# Hyperparameter Tuning for XGBoost
from sklearn.model_selection import RandomizedSearchCV

# Define the parameter grid for XGBoost
param_grid_xgb = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 10],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}

# Initialize RandomizedSearchCV
random_search_xgb = RandomizedSearchCV(
    estimator=XGBClassifier(scale_pos_weight=5, random_state=42,
    ↪  use_label_encoder=False, eval_metric='logloss'),
    param_distributions=param_grid_xgb,
    n_iter=20,
    scoring='roc_auc',
    cv=3,
    random_state=42,
    verbose=2,
    n_jobs=-1
)

# Fit the randomized search
random_search_xgb.fit(X_train_resampled, y_train_resampled)

# Best parameters
print("Best Parameters for XGBoost:", random_search_xgb.best_params_)

# Train and evaluate with the best parameters
best_xgb_model = random_search_xgb.best_estimator_
y_pred_best_xgb = best_xgb_model.predict(X_test_scaled)
y_pred_proba_best_xgb = best_xgb_model.predict_proba(X_test_scaled)[:, 1]

print("XGBoost (Tuned) AUC-ROC:", roc_auc_score(y_test,
↪  y_pred_proba_best_xgb))

# Optimized Hyperparameter Tuning for Random Forest
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split

# Subsample training data for faster tuning
X_train_sample, _, y_train_sample, _ = train_test_split(
    X_train_resampled, y_train_resampled, test_size=0.9, random_state=42  # Use only 10% of the data
)

# Simplified parameter grid
param_grid_rf = {
    'n_estimators': [50, 100],        # Smaller forests
    'max_depth': [10, 20],            # Moderate depth
    'min_samples_split': [2],         # Default split criteria
    'min_samples_leaf': [1],          # Default leaf size
    'bootstrap': [True]               # Fixed parameter
}

# Initialize RandomizedSearchCV with fewer iterations
random_search_rf = RandomizedSearchCV(
    estimator=RandomForestClassifier(class_weight="balanced", random_state=42),
    param_distributions=param_grid_rf,
    n_iter=5,  # Very few iterations
    scoring='roc_auc',
    cv=3,      # 3 folds for reliability
    random_state=42,
    verbose=2,
    n_jobs=-1  # Utilize all available CPU cores
)

# Fit the RandomizedSearchCV
print("Starting Random Forest Hyperparameter Tuning...")
random_search_rf.fit(X_train_sample, y_train_sample)

# Best parameters from RandomizedSearchCV
print("\nBest Parameters for Random Forest:", random_search_rf.best_params_)

# Train the final model with the best parameters
best_rf_model = random_search_rf.best_estimator_
y_pred_best_rf = best_rf_model.predict(X_test_scaled)
y_pred_proba_best_rf = best_rf_model.predict_proba(X_test_scaled)[:, 1]

# Evaluate the final model
print("\nRandom Forest (Tuned) AUC-ROC:", roc_auc_score(y_test, y_pred_proba_best_rf))

# Confusion Matrix and Classification Report
from sklearn.metrics import classification_report, confusion_matrix
```

```
388 print("\nConfusion Matrix:")
389 print(confusion_matrix(y_test, y_pred_best_rf))
390
391 print("\nClassification Report:")
392 print(classification_report(y_test, y_pred_best_rf))
393
394 # Hyperparameter Tuning for Logistic Regression
395 from sklearn.linear_model import LogisticRegression
396
397 # Define the parameter grid for Logistic Regression
398 param_grid_lr = {
399     'penalty': ['l1', 'l2', 'elasticnet', None],
400     'C': [0.01, 0.1, 1, 10, 100],  # Regularization strength
401     'solver': ['saga', 'lbfgs'],  # Optimizers (based on penalty type)
402     'max_iter': [100, 200, 500]
403 }
404
405 # Initialize RandomizedSearchCV
406 random_search_lr = RandomizedSearchCV(
407     estimator=LogisticRegression(class_weight="balanced", random_state=42),
408     param_distributions=param_grid_lr,
409     n_iter=20,
410     scoring='roc_auc',
411     cv=3,
412     random_state=42,
413     verbose=2,
414     n_jobs=-1
415 )
416
417 # Fit the randomized search
418 random_search_lr.fit(X_train_resampled, y_train_resampled)
419
420 # Best parameters
421 print("Best Parameters for Logistic Regression:",
    ↪ random_search_lr.best_params_)
422
423 # Train and evaluate with the best parameters
424 best_lr_model = random_search_lr.best_estimator_
425 y_pred_best_lr = best_lr_model.predict(X_test_scaled)
426 y_pred_proba_best_lr = best_lr_model.predict_proba(X_test_scaled)[:, 1]
427
428 print("Logistic Regression (Tuned) AUC-ROC:", roc_auc_score(y_test,
    ↪ y_pred_proba_best_lr))
429
430 from sklearn.metrics import classification_report, confusion_matrix,
    ↪ roc_auc_score
431
432 # Evaluate XGBoost
433 print("XGBoost (Tuned) Evaluation:")
434 print("AUC-ROC:", roc_auc_score(y_test, y_pred_proba_best_xgb))
435 print("\nConfusion Matrix:")
```

```
436 print(confusion_matrix(y_test, y_pred_best_xgb))
437 print("\nClassification Report:")
438 print(classification_report(y_test, y_pred_best_xgb))
439
440 # Evaluate Random Forest
441 print("\nRandom Forest (Tuned) Evaluation:")
442 print("AUC-ROC:", roc_auc_score(y_test, y_pred_proba_best_rf))
443 print("\nConfusion Matrix:")
444 print(confusion_matrix(y_test, y_pred_best_rf))
445 print("\nClassification Report:")
446 print(classification_report(y_test, y_pred_best_rf))
447
448 # Evaluate Logistic Regression
449 print("\nLogistic Regression (Tuned) Evaluation:")
450 print("AUC-ROC:", roc_auc_score(y_test, y_pred_proba_best_lr))
451 print("\nConfusion Matrix:")
452 print(confusion_matrix(y_test, y_pred_best_lr))
453 print("\nClassification Report:")
454 print(classification_report(y_test, y_pred_best_lr))
455
456 from sklearn.ensemble import VotingClassifier
457
458 # Combine the tuned models into a Voting Classifier
459 voting_clf = VotingClassifier(
460     estimators=[
461         ('xgb', best_xgb_model),
462         ('rf', best_rf_model),
463         ('lr', best_lr_model)
464     ],
465     voting='soft'  # Use soft voting to average probabilities
466 )
467
468 # Train the ensemble model on the full training data
469 voting_clf.fit(X_train_resampled, y_train_resampled)
470
471 # Make predictions on the test set
472 y_pred_voting = voting_clf.predict(X_test_scaled)
473 y_pred_proba_voting = voting_clf.predict_proba(X_test_scaled)[:, 1]
474
475 # Evaluate the ensemble
476 print("\nVoting Classifier Evaluation:")
477 print("AUC-ROC:", roc_auc_score(y_test, y_pred_proba_voting))
478 print("\nConfusion Matrix:")
479 print(confusion_matrix(y_test, y_pred_voting))
480 print("\nClassification Report:")
481 print(classification_report(y_test, y_pred_voting))
482
483 # Generate a dictionary of results for easy comparison
484 results = {
485     'Model': ['XGBoost', 'Random Forest', 'Logistic Regression', 'Voting
      ↪  Classifier'],
```

```
    'AUC-ROC': [
        roc_auc_score(y_test, y_pred_proba_best_xgb),
        roc_auc_score(y_test, y_pred_proba_best_rf),
        roc_auc_score(y_test, y_pred_proba_best_lr),
        roc_auc_score(y_test, y_pred_proba_voting)
    ]
}

import pandas as pd
results_df = pd.DataFrame(results)
print(results_df)
```