HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
School of Information and Communication Technology

----- □ & □ -----

Capstone project

Credit Card Fraud Detection

Subject: Machine Learning – IT3190E

Supervisor: Assoc. Prof. Dr. Than Quang Khoat

Group: 17

Authors:

Vũ Hoàng Nhật Anh	20225471
Chu Trung Anh	20225564
Trần Nhật Minh	20225511
Trần Nam Tuấn Vượng	20225540

Hanoi, 05/2024

Abstract

This report investigates the use of machine learning techniques for detecting credit card fraud. The main objective was to develop predictive models capable of accurately identifying fraudulent transactions while minimizing false positives. The analysis utilized a dataset of anonymized credit card transactions, categorized as either fraudulent or legitimate.

To achieve this, various machine learning algorithms were implemented and compared. The dataset underwent preprocessing to address class imbalance and extract relevant features. The performance of the models was assessed using metrics such as accuracy, precision, recall, and the F1-score.

The findings demonstrate the effectiveness of machine learning in fraud detection, emphasizing the need for ongoing model evaluation and updates to adapt to changing fraudulent patterns. Future work will aim to incorporate real-time detection capabilities and explore ensemble methods to further enhance detection performance.

Contributors to this project:

- Vu Hoang Nhat Anh: Multi-layer Perception and Deep Neural Networks
- Chu Trung Anh: Logistic Regression, Decision Tree, Random Forest
- Tran Nam Tuan Vuong: Support Vector Machine
- Tran Nhat Minh: K-Nearest Neighbors, processing Dataset3

The report and documentation report were written by the team members corresponding to their respective contributions.

Table of Contents

Part I: Introduction

1. Background

Credit fraud or Credit card fraud is an inclusive term for fraud or illegal activities committed by criminals using payment cards such as debit cards, credit cards to make purchases, payments from a stolen account to another one. In some cases, credit card fraud can be authorized as victims were tricked to make a transaction to another account owned and controlled by a criminal. In other cases, the victims do not authorize transactions, do not provide authorization information to proceed, but it is already done by a third party.

The history of credit card fraud goes back to the early 90s. At the time, credit card verification technology was in its infancy, and lacked a lot of security measures. The thieves aimed at celebrities to purchase expensive items or businesses with large volume of credit card transactions. Since then, fraudsters have continued to rampage and deal irreversible damage to the financial world. In recent years, there has been approximately 400 000 cases reported to authorities annually in the United States alone, and the number is predicted by the European Central Bank to keep increase in the following years, as more people have access to credit card. The amount of money lost to fraudsters is incredibly costly as it is estimated that the typical organization loses 5% of its annual revenue to fraud each year. When being applied to the 2023 Gross World Product of $105 trillion (about $320,000 per person in the US), it means an astounding $5.25 trillion (about $16,000 per person in the US) lost.

2. Problem Formulation

As mentioned above, necessary prevention measures can be taken to help companies to be able to identify fraudulent credit card transactions so that the customers are not charged for items that they have no intention of purchasing. A fraud detection problem can be described as the work of monitoring the activities of populations of users, to estimate, perceive or avoid objectionable behavior. In the real-world scenario, the number of transactions made every day is enormous, up

to hundreds of millions, which makes the problem harder to approach. However, this is not the only challenge that occurs, the highly imbalance ration between fraudulent and valid transactions also creates a lot of difficulties for a good method of evaluating. Analyzing such a large amount of information and calculations is not a suitable job for humans. Machine learning algorithms propose good opportunities in solving this problem, as they can analyze patterns across many authorized transactions, instantly determine the activities diverging from normal baselines, even the slightest anomalies. The problem of credit card fraud detection can be classified as a **binary classification machine learning problem**, where the objective is to classify each transaction as either fraudulent or non-fraudulent. In this context

3. Aims

The aim of our group project, titled "Credit Fraud Detection," is to develop and implement an advanced, robust, and efficient machine learning-based system capable of accurately identifying and preventing fraudulent credit card transactions in real time. This project seeks to address the pressing issue of credit card fraud, which has increasingly become a significant threat to the financial security of both consumers and businesses worldwide. By leveraging the power of machine learning algorithms, we aim to create a solution that not only enhances the detection rates of fraudulent activities but also minimizes false positives, thereby ensuring a seamless and secure transactional experience for legitimate users.

Our objectives can be described as follows:

- Develop comprehensive fraud detection models: Design and implement knowledge of machine learning to detect anomalous transactions pattern indicative of fraud. These models are trained and evaluated on large datasets to give out the best results possible.
- Address class imbalance in datasets: using strategies to effectively handle the class imbalance problem inherent in fraud detection datasets, where the number of legitimate transactions can outnumber the number of fraudulent ones. By applying these techniques into the dataset,

These are the primary aims of our project, with a desire to make an impact on the fight against credit card fraud, provide a secured and safe environment for financial transactions and provide a chance for us to have first-hand experience on machine learning models and its real-world implementation.

Part II: Dataset

1. Dataset Description

For training and testing purposes of this project, our group has used three datasets with different properties.

Dataset1

We use this dataset for training our models.

Description:

Our first dataset, in this project's context, will be called Dataset1. Dataset1 contains information of credit cards purchase in 2 days of September 2013 by European cardholders. The dataset was created by Machine Learning Group – ULB and was published onto Kaggle website, found in this link. The dataset has been analyzed multiple times by companies and organizations all over the world.

To protect sensitive information, the transaction data in this dataset has been transformed using Principal Component Analysis (PCA). The resulting numerical values are represented in columns V1 through V28. These columns capture the essence of the original data while maintaining confidentiality. Only the **'Time'**, **'Amount'**, and **'Class'** features remain in their original form. Here is an overview of the dataset:

	Time	V1	V2	V3	...	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	...	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	...	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	...	-0.059752	378.66	0

Here is a summary of the dataset's structure:

- **V1 - V28**: Principal components derived from the original transaction data as mentioned previously.
- **'Time'** contains the seconds elapsed between each transaction and the first transaction in the dataset.
- **'Amount'** is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning.
- **`Class`**: Indicates whether a transaction is fraudulent (1) or non-fraudulent (0).

Key Features:

The most crucial feature of Dataset1 is its extreme imbalance. Out of 284,807 transactions, only 492 are fraudulent, representing a mere **0.172%**. This imbalance poses a big challenge for fraud detection models, as they must be able to identify rare events within most legitimate transactions.

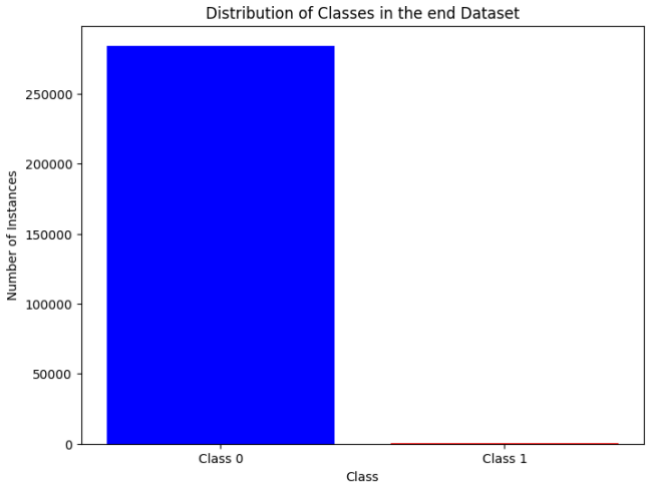


Figure 2.1 : The number of Dataset1's fraudulent and legitimate transactions.

The correlation matrix shows that attribute class is independent of both the amount and the time of the transaction was made and the class of the transaction depends on the attributes that were PCA processed.

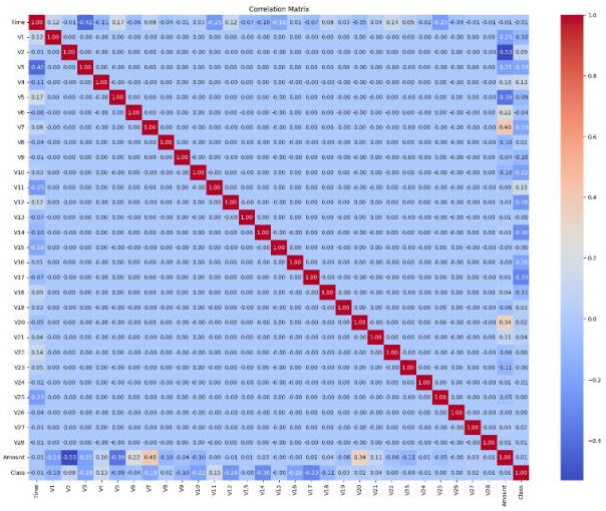


Figure 2.2 The correlation matrix of features of Dataset1.

Since the feature descriptions are not provided, further visualizing the data will not lead to much insight.

Dataset2

We use this dataset for testing, comparing between our models.

Description:

Our first dataset, in this project's context, will now be referred to as Dataset2. This dataset contains credit card transactions made by European cardholders in the year 2023. It comprises **over 550,000 records**, and the data has been anonymized to protect the cardholders' identities. The download link of this dataset can be found here.

Key Features:

Dataset2 follows the same structure as Dataset1, but it is larger (around 324,8 MB compared to 150,8 MB) and offers a more balanced representation of fraudulent and non-fraudulent transactions as shown in the distribution of the 'Class' feature in Dataset2:

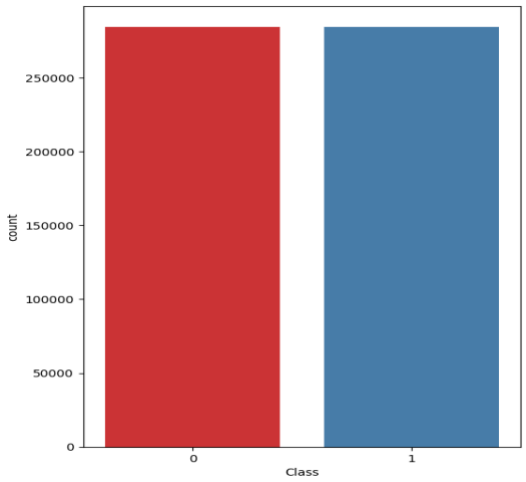


Figure 2.3 The number of Dataset2's fraudulent and legitimate transactions.

Dataset3

Currently, we just process this dataset without any usage.

Description:

Dataset3 contains credit card transactions fraudulent and legitimate from 1st January 2019 to 31st of December 2020 inside the United States. The dataset covers the credit card information of 1000 customers with a pool of 800 merchants. The dataset was created by Brandon Harris and Kartik Shenoy and published on Kaggle. The download link can be found here.

This dataset contains detailed information of the customers like their names, addresses, jobs, or date-of-birth, as well as information of the merchants and transaction categories. A transformation of this dataset is required as most information is not numerical, some can be regarded as redundant, and the dataset overall is heavily imbalanced. Here is an overview of the dataset:

Trans_date_trans_time	**Cc_num**	**merchant**	**category**	**amt**	**first**	**last**	**...**	**long**	**City_pop**
2019-01-01 00:00:18	2703186189652095	Fraud_Rippin,Kub and Mann	Misc_net	4.97	Jennifer	Banks	...	-81.1781	3495
2019-01-01 00:00:44	630423337322	Fraud_Heller, Gultmann and Zieme	Grocery_pos	107.23	Stephanie	Gill	...	-118.2105	149
2019-01-01 00:00:51	3885492057661	Fraud_Lind-Buckrindge	entertainment	220.11	Edward	Sanchez	...	-112.2620	4154

And the correlation matrix between variables:

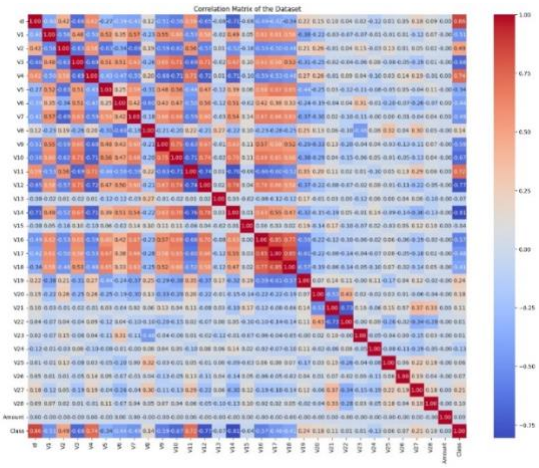


Figure 2.4 Correlation matrix for Dataset3.

Key Features:

Similar to Dataset1, this dataset also exhibits an imbalance, as evidenced by the distribution of its **'is_fraud '**attribute.

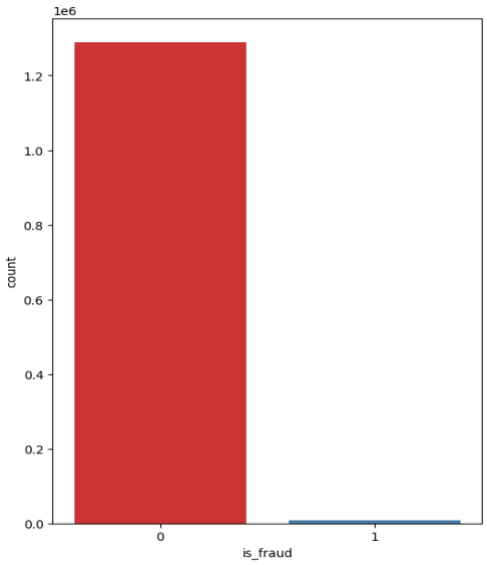


Figure 2.5 The number of Dataset3's fraudulent and legitimate transactions.

There are only 9,651 fraudulent transactions out of 1,842,743 total transactions, accounting for a mere 0.52%.

2. Preprocessing

With Dataset1 and Dataset2 from Kaggle and is already in a cleaned format without any empty rows or columns. Data cleaning or categorization is unnecessary so we will just discuss the transformation for Dataset3.

Additionally in this part, we will utilize some techniques like **undersampling** and **oversampling** to balance the distribution of fraud and non-fraud cases in all datasets.

2.1. Data transformation (only for Dataset3):

Our third dataset, named "Dataset3", contains information of the transactions in its raw forms. Firstly, info like the names of customers, their street names, city and states, their gender, their jobs, the name of the merchants from which they were purchasing, and the category of the transaction are given as text labels. Other information, namely the time and money value of transactions, credit card numbers, the population of the cities where the customers lived and its coordination, are numerical values lie between different ranges, the values of credit card number specifically do not have any relation between them.

The goal of this data transformation step is to standardize and normalize the Dataset3 so that it can work well with any models. Specifically, we will transform all its data into numerical values, ranging from 0 to 1 using various techniques to preserve their relation.

Trans_date_trans_time	Cc_num	merchant	category	amt	first	last	...	long	City_pop
2019-01-01 00:00:18	2703186189652095	Fraud_Rippin,Kub and Mann	Misc_net	4.97	Jennifer	Banks	...	-81.1781	3495
2019-01-01 00:00:44	630423337322	Fraud_Heller, Gultmann and Zieme	Grocery_pos	107.23	Stephanie	Gill	...	-118.2105	149
2019-01-01 00:00:51	3885492057661	Fraud_Lind-Buckrindge	entertainment	220.11	Edward	Sanchez	...	-112.2620	4154

Table: Preview of Dataset3.

The first sub step is to remove redundant information from the dataset. After a close inspection of Dataset3, we decided to remove the following attributes:

- The first `unnamed` column and 'trans_num': The first column is the index column which does not contain any data related to the transactions and is only used to keep track of the row numbers. Similarly, the 'trans_num' attribute contains the randomly assigned string to the transactions for administrative purposes only.
- 'unix_time': This attribute contains the times at which the transactions occurred in Unix format, which can be easily inferred from the 'trans_date_trans_time' attribute that serves the same purpose.
- 'cc_num,' 'first', 'last', 'street', 'state', 'zip' and 'city': The credit card numbers in "cc_num" are used to keep track of the customers, however each customer and their location can already be identified by some combination of personal data from other attributes. Upon further inspection, we noticed that the combination of 'lat' and 'long' (coordinates), 'dob'

(date of birth) and 'job' is sufficient for this purpose, and each column in this combination can be used for other purposes not just identification, other personal data there for can be removed.

- 'dob': In many contexts in the real world, a person's date of birth and their age can be used interchangeably. For the purposes of this project specifically, while exact date of birth does not provide more information, the age of a customer can put them into different demographics that tell us more about them, and its format is also easier to work with compared to that of the dates. Therefore, we will replace this attribute with the age of the customer in relation to the time at which they performed the transaction.

The transformation process can now begin. For numerical data in this dataset, our general idea is to use a technique called Min-max Scaling, of which the function is as follows:

$$f(x) = \frac{x - x_{min}}{x_{max} - x_{min}}$$

where x_{max} and x_{min} are the maximum and minimum values respectively

This technique is an uncomplicated way to scale numerical values to the range from 0 to 1. However, to consider future data and avoid overfitting, we will instead use modified versions of it based on the nature of each attribute. The detail is as follows:

- Age ('age'): In the United States, you must be eighteen or older to be authorized a credit card. On the other end of the spectrum, while it is hard to know who the oldest person in the country is to own a credit card, it is known that the oldest living person is 114 years old. We used these two numbers as the minimum and maximum values for this attribute so that our models can stay consistent when encountering data that lies outside of the range of this dataset.
- Coordinates ('lat,' 'long,' 'merch_lat' and 'merch_long'): Like customers' age data, we used the coordinate boundaries of the US to scale the coordinates as following:
    - Northernmost (lat_max): 49.382808
    - Southernmost (lat_min): 24.521208
    - Easternmost (long_max): -66.945392
    - Westernmost (long_min): -124.736342
- City population ('city_pop'): The most popular city in the US in 2020 is New York City at 8,804,190 people (about half the population of New York) which can be used similarly to the above. However, this dataset was generated in a simulation of which the city population does not match with real life, all big cities in the country were included but given wrong population, the most populated one became Houston with 2,906,700 people (about the population of Connecticut). Therefore, we just used the standard Min-max scaler for this one.
- Amount ('amt'): We again used the standard Min-max scaler for this attribution as our knowledge about it is extremely limited and the information is not publicly available. With

better knowledge about the field, a similar process to that of what we applied above can be performed to yell better results.

For data given in text label forms, we performed "encoding" on them to turn them in to numerical values.

The customers' gender attribute only takes two values, either 'F' for female or 'M' for male, so we used a technique call "Ordinal encoding," simply assigned zero for female and 1 for male. About the other text label attribute, we will take 'merchant,' the name of the merchants involved with the transactions, as an example, and the rest was treated similarly.

The first idea that came to mind was to also perform Ordinal encoding by arbitrarily assigning number from zero to n to each merchant, with n is the total number of merchants in this dataset, and then scale it down. However, in many models, merchant numbered 100 for example, will be like merchant numbered 101 and vastly different from merchant numbered 2, which is not at all the intention. A better way to assign numbers to merchants therefore is required.

The technique we used for this part is called "Target encoding." A target encoding is any kind of encoding that replaces a feature's categories with some number derived from the target. A simple form of target encoding is to calculate the mean value of the target of each category, in other words, the probability that a transaction is fraudulent, given the merchant involved:

$$Mean = \frac{Number\ of\ fraudulent\ transactions\ involved}{Total\ number\ of\ transactions\ involved}$$

This approach is prone to overfitting as numbers assigned to rarer categories will have less data to support itself. To resolve this, we added "smoothing," that is, blend the in-category mean with the overall mean with weights:

$$Weighted\ mean = \frac{n * Category\ mean + m * Overall\ mean}{n + m}$$

$$n = Total\ number\ of\ transactions\ involved$$

$$m = "Smoothing\ factor"\ determined\ by\ the\ user$$

The smoothing factor decides the weight of the overall mean. A merchant must be involved in more than m transactions for its category means to be considered more important. For simplicity's sake, we chose m = 5, a more optimal value of m can be chosen through trial and error.

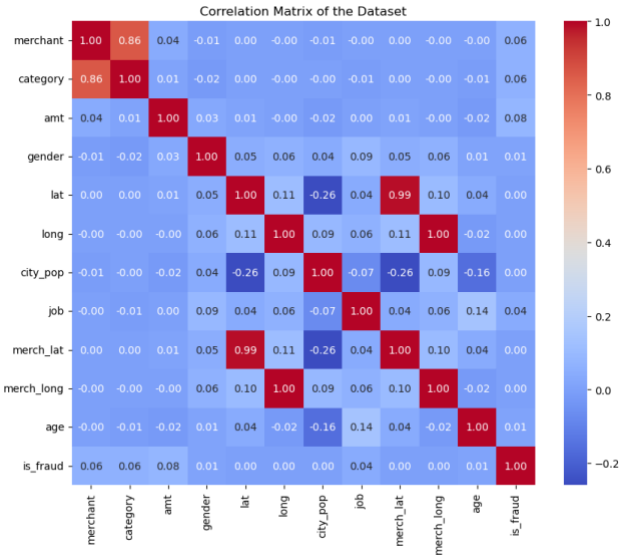


Figure 2.6 The correlation of Dataset1 after transforming.

2.2. Deal with imbalanced dataset:

As mentioned, our Dataset1 and Dataset3 are extremely imbalanced, so if there were no actions taken to process the data, it is likely that the results printed out are not a good result. To prevent that from happening, our group implemented different techniques for the preprocessing process for the dataset.

- **Under sampling**: This technique is used to balance uneven datasets by keeping all the data in the minority class and decreasing the size of the majority class. The majority class's events are randomly deleted to end up with the same number as the minority class.
- **Oversampling**: This technique is used to balance uneven datasets by increasing the number of minority events. In the context of our project, we have used 2 oversampling techniques: Randomly and Synthetic Minority Over-sampling Technique (SMOTE), an improvement from the duplicating technique.
    - **Random Oversampling** is a technique that works by simply duplicating the examples of the minority class of the dataset, adding them to the training set, so we can acquire a dataset of equal number of classes.
    - **SMOTE** first selects a minority class instance at random and finds its nearest minority neighbors. The procedure can be used to create as many synthetic examples for the minority class as are required. It is suggested that under sampling should be used beforehand to trim the number of examples in the majority class, then by using SMOTE, we can oversample the minority class to balance the class distribution.

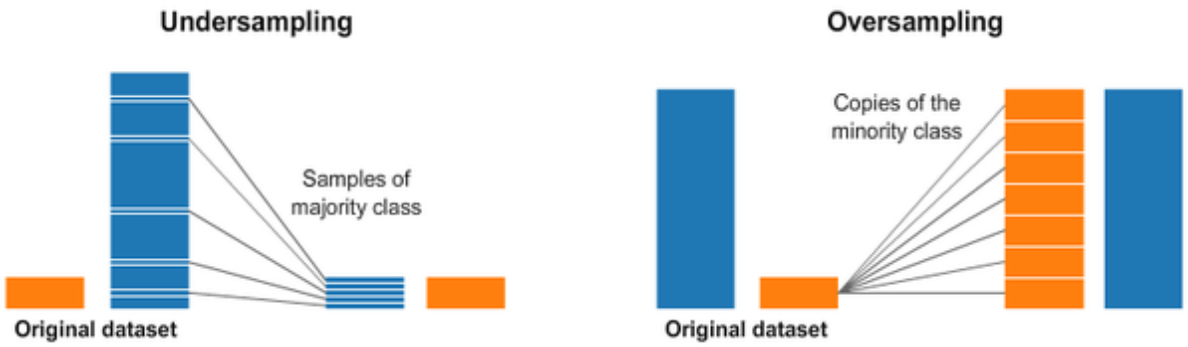
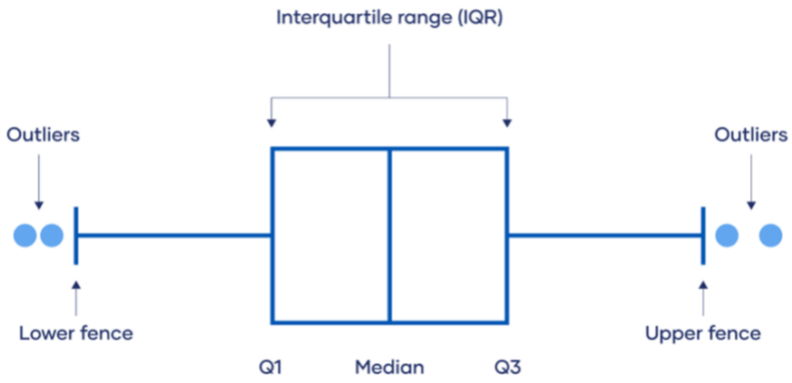


Figure 2.7 Picture: demonstration of the process of Undersampling and Oversampling

Both techniques are "naive resampling" methods as they assume nothing about the data of the dataset and the lack of a heuristic method. They can deal with two-class classification problems and a large, complex dataset. In general, these naive methods can be effective and might perform well on this dataset.

2.3. Anomaly detection using the interquartile range

The interquartile range (IQR) tells you the range of the middle half of your dataset. You can use the IQR to create "fences" around your data and then define outliers as any values that fall outside those fences.



Interquartile range method:

- Identify the first quartile (Q1), the median, and the third quartile (Q3).
- Calculate IQR = Q3 – Q1.
- Calculate upper fence = Q3 + (w * IQR) and lower fence = Q1 – (w * IQR) where w is the range weight (usually set as 1.5).

- Use fences to indicate the outliers which are all the values that fall outside the fences.

2.4 Dimensionality Reduction

In this part, we use PCA and SVD functions from Scikit-learn library to reduce the number of features without causing a big information loss.

Dimensionality reduction is the process of reducing the number of features (or dimensions) in a dataset while retaining as much information as possible. This can be done for a variety of reasons, such as to reduce the complexity of a model, to improve the performance of a learning algorithm, or to make it easier to visualize the data. There are several techniques for dimensionality reduction, including principal component analysis (PCA), singular value decomposition (SVD).

Methods of Dimensionality Reduction:

- Principal Component Analysis
  - Standardize the range of continuous initial variables.
  - Compute the covariance matrix to identify correlations.
  - Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components.
  - Create a feature vector to decide which principal components to keep.
  - Recast the data along the principal component's axes.
- Truncated Singular Value Decomposition
  - The SVD of m x n matrix A is given by the formula A = U$\sum V^T$
  - U: m x m matrix of the orthonormal eigenvectors of AA^T.
  - V^T: :transpose of a n x n matrix containing the orthonormal eigenvectors of AA^T.
  - $\sum$: diagonal matrix with r elements equal to the root of the positive eigenvalues of AA^T or A^TA.
  - A given m×n matrix truncated SVD will produce matrices with the specified number of columns, whereas a normal SVD procedure will produce with m columns. It means that it will drop off all features except the number of features provided to it.

3. Results

By applying these preprocessing techniques, we were able to use the imbalance dataset effectively. The following is the result of this process:

Under sampling:

Class 0: 284315 1: 492	→	Class 0: 492 1: 492

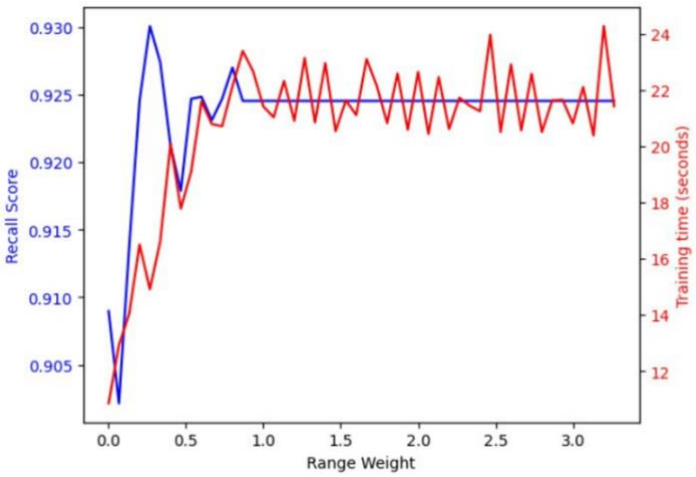
Name: count, dtype: int64	Name: count, dtype: int64

Oversampling (Duplicate and SMOTE):

Class 0: 284315 1: 492 Name: count, dtype: int64	Class 0: 284315 1: 284315 Name: count, dtype: int64

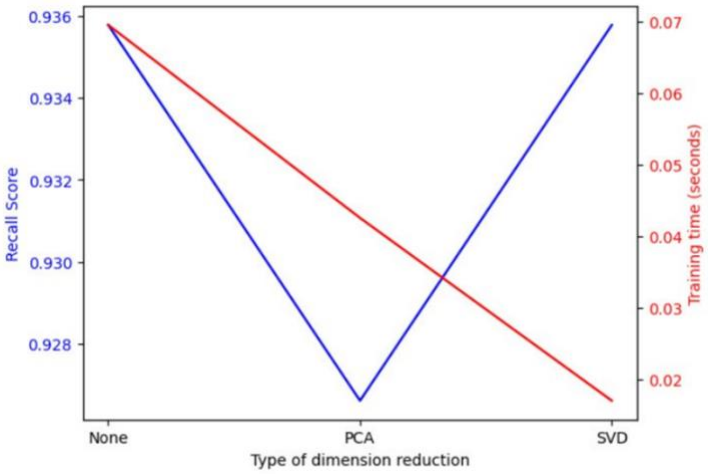
By using two different techniques of preprocessing the imbalance dataset, we can make comparisons between techniques, adjust the properties of the dataset, to give out the best result possible.

Here is the graph indicating the relation between range weight and recall score:

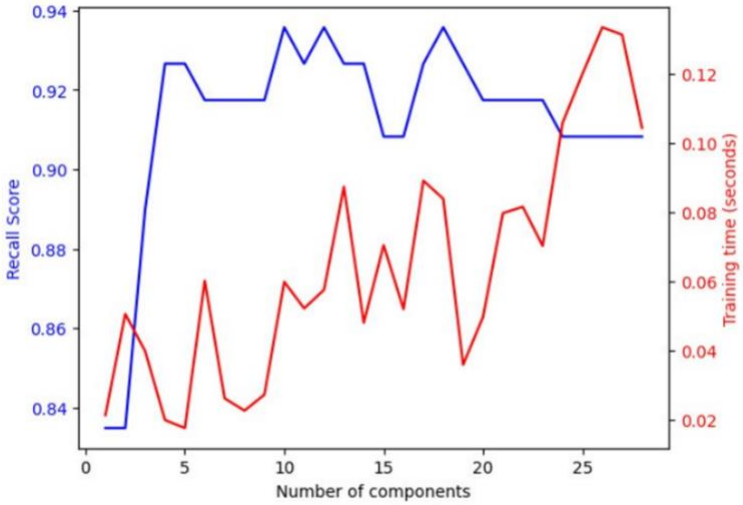


The graph shows that the model after anomaly detection can have better performance and take less time to run.

Below is the line chart showing that SVD function gives us better result than PCA function:



From the above graph, we can see that SVD function can reduce the complexity of the model; therefore, highly increase the speed while keeping the performance of the data.



Part III: Methodology

In this part of the report, we shall discuss the methodology used for this project. We have tried seven different machine learning models to give out the best result for this classification problem.

1. **Logistic Regression**
2. **SVM**
3. **KNN**
4. **Random Forest**
5. **Decision Trees**
6. **Multi-layer Perception**
7. **Deep Neural Networks**

We will introduce each model in more detail. For each model, we will discuss its theory, implementation (including hyperparameter tuning and code), and results.

1. Logistic Regression

1.1 Theory

Logistic regression is a supervised machine learning algorithm, primarily used for binary classification problems, such as whether a customer will buy a product, whether a patient has a disease (Logistic regression can also be applied to solve multiclass excluded in this report, we only discuss binary classification). The “Regression” term indicates the main idea of this metho: to find a function that represents well the relationship between the independent variable x and the dependent variable y.

Recall the first model we learned, the Linear Regression model, which uses a hyperplane to represent the relationship between x and y. Logistic Regression operates on a similar principle but is designed for binary outcomes, where the dependent variable can only take on two classes: 0 and 1, unlike Linear Regression, which predicts continuous values. Consequently, instead of fitting a straight line to the data, Logistic Regression fits an S-shaped curve. This curve ranges from 0 to 1, representing the probability that a data instance belongs to a particular class, as shown in the graph below.

The S-shape curve is defined by the **Sigmoid function**, a nonlinear function that takes an arbitrary input value z and returns an output probability in the range [0, 1]. The sigmoid function is as follows:

$$S(z) = \frac{1}{1 + e^{-z}}$$

The sigmoid function has the following properties, making it the most common choice for Logistic Regression:

- It is always between 0 and 1.
- It is symmetrical around 0.
- It is a smooth, continuous function.
- Its derivative is easy to compute:

$$S'(z) = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} * \frac{e^{-z}}{1 + e^{-z}} = S(z)(1 - S(z))$$

Based on that probability, Logistic Regression determines the class of data instances using a **threshold**. For example, if the threshold is set at 0.5, then all instances with probabilities greater than 0.5 are classified as class 1, while those with probabilities less than or equal to 0.5 are classified as class 0.

$$\text{Decision(x)} = \begin{cases} 1 \; if \; P(y = 1|x) > 0.5 \\ 0 \; otherwise \end{cases}$$

The optimal threshold value can vary depending on the problem at hand. Typically, we leverage the **ROC** (Receiver Operating Characteristic) curve to determine the best threshold. However, when dealing with imbalanced data, the Precision-Recall (PR) curve proves to be more effective than ROC.

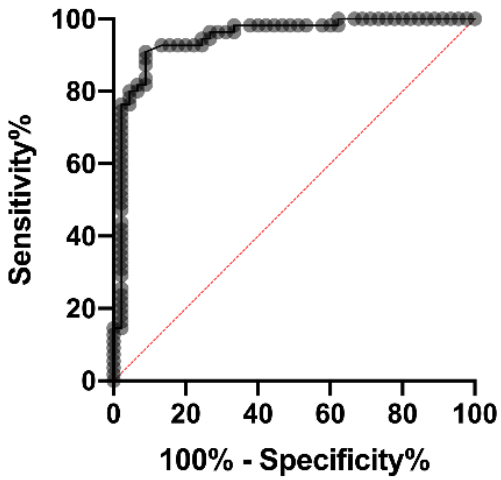


Figure 3.1 Example of a ROC graph

- The **True Positive Rate**, also known as S**ensitivity** / Recall (plotted on the y-axis), represents the proportion of actual positive cases (in our context, fraudulent transactions) that are correctly identified. A higher value indicates a better ability of the model to correctly classify positive instances.
- The **False Positive Rate** (plotted on the x-axis) refers to the proportion of negative instances that are incorrectly classified as positive out of all actual negative instances. In simple words, this is the wrong alert rate.

In our credit fraud detection system, we aim for a point on the graph where the y- is maximized (ensures we minimize missed fraud cases, which cause economic loss when happen), while keeping the x-axis low. Though there is a trade-off between these two metrics, we prioritize the y-axis value since misclassifying non-fraudulent transactions as fraud (which corresponds to higher x-axis values) typically does not incur financial losses for the bank.

Now, applied the logistic regression model with the dataset D = $\{(x_1, y_1), (x_2, y_2) \ldots, (x_n, y_n)$, each observation of x_i is represented by a n-dimensional vector, e.g., $x_i = \{x_{i1}, x_{i2}, \ldots, x_{im}\}^T$. Each dimension represents an attribute/feature.

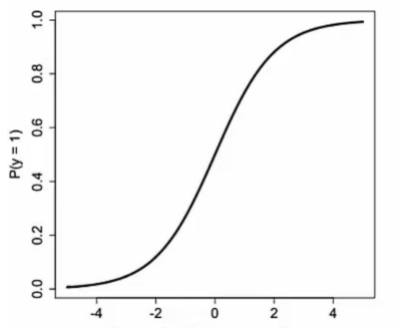
$$\begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}$$

We then re-write z by; $z = w \cdot X + b = \sum_{i=1}^{n} w_i * x_i + b$

Here x_i is the i^{th} observation of X, $w_i = [w_1, w_2, w_3, \cdots, w_n]$ is the weights or Coefficient, and b is the bias term also known as intercept. Some other documentations do not write b as a bias, they use the term w_0 and put that into the weight matrix, $w_i = [w_0, w_1, w_2, w_3, \cdots, w_m]$, w_0 will go later go with $x_0 = 0$, so in this case w_0 is the bias.

The formula of Logistic Regression becomes:

$$P_{(y=1)} = \frac{1}{1 + e^{-(w_0 + w_1 x_1 + \cdots + w_n x_n)}}$$



By this formula, Logistic Regression can be seen as a kind of ANN:

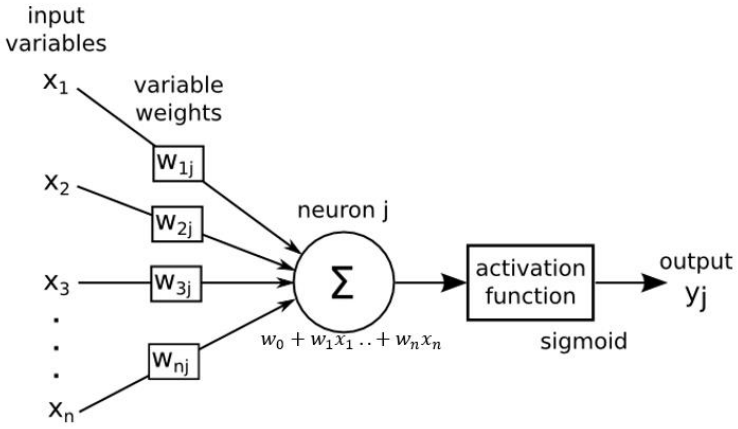


Figure 3.2 Logistic Regression from ANN view.

Learning a Logistic Regression model involves finding a weight vector that makes the predicted output as close to the correct output as possible. To achieve this, we use a loss function. The smaller the loss function value, the better our model performs. Specifically, the loss function used here is **Cross Entropy**, also known as **Log Loss**. The process of constructing this function is outlined below:

Start from:

$$P(y_i|x_i, w) = P_{(y=1)}{}^{y_i} * \left(1 - P_{(y=1)}\right)^{1-y_i}$$

(by Bernoulli Distribution)

with for y_i in {0,1}; x_i, w represents the output conditional probability given corresponding x_i and weight vector.

We then replace $P_{(y=1)} = \frac{1}{1+e^{-(w_0+w_1x_1+\cdots+w_nx_n)}}$, or we can write the $\frac{1}{1+e^{-(w_0+w_1x_1+\cdots+w_nx_n)}}$ for short is $f(w^T x_i)$

That is for an x observation, the function on whole observation in dataset X = $\{x_{1,}\, x_{2}\, \ldots, x_n\}$ will be (assume that each observation is independent to each other):

$$P(y|X\,, w) = \prod_{i=1}^{n} P(y_i|x_i, w) \;=\; \prod_{i=1}^{n} f(w^T x_i)^{yi} \;*\; \left(1 - f(w^T x_i)\right)^{1-y_i}$$

This is the likelihood function. We expect the value of this function to be as large as possible since that means data instances belonging to class 1 will have the probability near 1 and data instances belonging to class 0 will have the probability near 0. So, our goal now is to find an optimal w^* such that it maximizes $P(y|X\,, w)$:

$$\boldsymbol{w^* \;=\; arg\; \max_{w}\; P(y|X\,, w)}$$

Directly optimizing this function is not a good approach due to complexity, so we take the log () to the left-hand side to covert from $\prod$ to $\sum$.

$$\omega^* = arg\; \max_{w} log\left(\prod_{i=1}^{n} f(w^T x_i)^{yi} \;*\; \left(1 - f(w^T x_i)\right)^{1-y_i}\right)$$

$$= arg \max_{w} \sum_{i=1}^{n} y_i\; log(f(w^T x_i)) \;+\; (1 \;-\; y_i)\; log(1 \;-\; f(w^T x_i)$$

$$= \; arg \min_{w} \sum_{i=1}^{n} -\left[\, y_i log\big(f(w^T x_i)\big) + (1 - y_i) log(1 - f(w^T x_i)\right]$$

So, the Cross Entropy is:

$$\boldsymbol{J\,(w, x_i, y_i) \;=\; \sum_{i=1}^{n} -\left[\, y_i\; log(f(w^T x_i)) \;+\; (1 \;-\; y_i)\; log(1 \;-\; f(w^T x_i)\right]}$$

This is a metric density function that measures the correlation between the probability distribution of the prediction ($P_{(yi=1)}$ and 1-$P_{(yi=1)}$) and the actual probability distribution ($y_i\; and\; 1 - \;y_i$). The value of Cross Entropy will be smaller if the two probability distributions are closer to each other, meaning the predicted value is more like the actual value.

To continue solving the Cross Entropy, we there are several ways, like using Lagrange multiplier, Gradient Descent, … Among these, we solve using the simplest algorithm which is Gradient Descent.

We take the derivative of the function:

$$\nabla J(w, x_i, y_i)_w = -[\, y_i \frac{\partial log(f(w^T x_i))}{\partial w} + (1 - y_i) \frac{\partial log(1 - f(w^T x_i))}{\partial w} \,]$$

$$= -[\, y_i \frac{\partial log(f(w^T x_i))}{\partial f(w^T x_i)} \frac{\partial f(w^T x_i)}{\partial w} + (1 - y_i) \frac{\partial log(1 - f(w^T x_i))}{\partial f(w^T x_i)} \frac{\partial f(w^T x_i)}{\partial w} \,]$$ (Chain Rule Derivative)

$$= -[\, y_i \frac{1}{f(w^T x_i)} + (1 - y_i) \frac{1}{1 - f(w^T x_i)} \,] \frac{\partial f(w^T x_i)}{\partial w}$$

$$= [\frac{y_i - f(w^T x_i)}{f(w^T x_i)(1 - f(w^T x_i))}] \frac{\partial f(w^T x_i)}{\partial w}$$

We analysis

$$\frac{\partial f(w^T x_i)}{\partial w} = \frac{\partial \frac{1}{1 + e^{-w^T x}}}{\partial w} = \frac{\partial \frac{1}{1 + e^{-w^T x}}}{\partial e^{-w^T x}} \frac{\partial e^{-w^T x}}{\partial w} = \frac{-1}{(1 + e^{-w^T x})^2} \cdot (e^{-w^T x} x_i)$$

$$= x_i \frac{-e^{-w^T x}}{(1 + e^{-w^T x})^2} = x_i f(w^T x_i)(1 - f(w^T x_i))$$

Then $[\frac{y_i - f(w^T x_i)}{f(w^T x_i)(1 - f(w^T x_i))}] \frac{\partial f(w^T x_i)}{\partial w} = x_i (y_i - f(w^T x_i))$

Finally, we have the Gradient Descent formula:

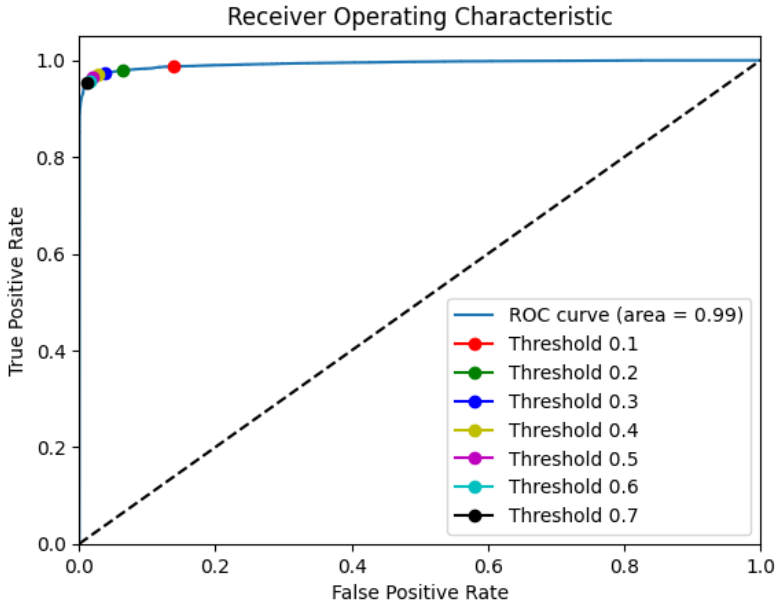
$$\boldsymbol{w_{update} = w - \alpha * x_i (y_i - f(w^T x_i))}$$

To demonstrate this algorithm, we implement the code in the next section.

1.2 Implementation

Hyperparameter Tuning

In this model, we first choose the threshold value by using the ROC curve. Below is the ROC curve plotted from dataset1 after applying SMOTE for imbalance preprocessing:

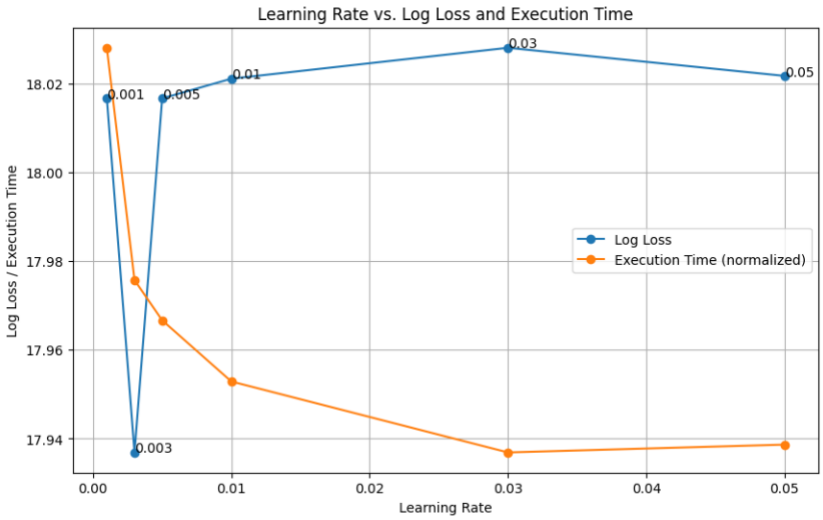


ROC from Dataset1

The method for selecting a good threshold value is detailed in the Theory section. In this instance, we choose a threshold of 0.2.

Another important parameter to evaluate is the learning rate. However, this parameter is only relevant if our Logistic Regression model uses Gradient Descent to optimize. By default, the LogisticRegression () function in the Scikit-learn library uses solvers like 'liblinear', 'newton-cg', 'sag', 'saga', and 'lbfgs', which do not allow for specifying the use of Gradient Descent.

Here is the plot of the training times for various learning rates we evaluated: *learning_rates = [0.001, 0.003, 0.005, 0.01, 0.03, 0.05]* with their corresponding training time:



The higher the learning rate, the quicker the algorithm converges, but it might not get as close to the optimal point. Conversely, a lower learning rate results in a more precise solution but increases the training time. However, this is just theoretical. In our case, the training time graph can be ignored since Logistic Regression executes in less than 1 minute for all our datasets. Therefore, we choose the learning rate that gives the best loss function value: 0.003.

Run Model

Our group has several approaches to this model:

a. Building a self-implemented Logistic Regression model using Gradient Descent:

We implemented our own Logistic Regression model using Gradient Descent. This method is beneficial for educational purposes and for understanding the underlying mechanics of the algorithm.

```
class LogisticRegression_shopee:
    def __init__(self, learning_rate=0.005, num_iterations=1000):
        self.learning_rate = learning_rate
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
      # Clip input values to be between -250 and 250
        z = np.clip(z, -500, 500)
        return 1 / (1 + np.exp(-z))

    def train(self, X, y):
        num_samples, num_features = X.shape

        # 1. Initialize weights and bias
```

```
        self.weights = np.zeros(num_features)
        self.bias = 0

        # 2. Gradient descent loop
        for i in range(self.num_iterations):
            hyperplane = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(hyperplane)

            # 3. Compute gradients
            self.weights -= self.learning_rate * np.dot(X.T, (y_predicted - y))

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(linear_model)
        y_predicted_cls = [1 if i >= 0.2 else 0 for i in y_predicted]
        return np.array(y_predicted_cls)

# Usage:
lr_shopee_model = LogisticRegression_shopee()
lr_shopee_model.train(np.concatenate((x_train, x_val)), np.concatenate((y_train, y_val)))
```

b. Build in Logistic Regression model using Gradient Descent:

As mentioned previously, LogisticRegression in Scikit-learn does not provide a way to use Gradient Descent directly. Therefore, we must use the SGDClassifier model from Scikit-learn with the parameter SGDClassifier(loss='log_loss') to implement Logistic Regression with Gradient Descent. This method leverages the Stochastic Gradient Descent (SGD) optimization, which can be more efficient for large datasets. Detailed instructions can be found here.

c. Default Logistic Regression in Scikit-learn

The default LogisticRegression in Scikit-learn uses various solvers such as '*liblinear*', '*newton-cg*', '*sag*', '*saga*', and '*lbfgs*', which are optimized for several types of datasets and use cases. Documentation for this model and its parameters can be found here.

1.3 Result

All tests are performed on Dataset1:

Model in part a:

	precision	recall	f1-score	support
Not fraud	0.50	1.00	0.67	56863
Fraud	0.35	0.00	0.00	56863

accuracy			0.50	113726
macro avg	0.42	0.50	0.33	113726
weighted avg	0.42	0.50	0.33	113726

Model in part b:

	precision	recall	f1-score	support
Not fraud	0.54	0.99	0.69	56863
Fraud	0.91	0.15	0.26	56863
accuracy			0.57	113726
macro avg	0.72	0.57	0.48	113726
weighted avg	0.72	0.57	0.48	113726

Model in part c:

	precision	recall	f1-score	support
Not fraud	0.96	0.98	0.97	56863
Fraud	0.98	0.96	0.97	56863
accuracy			0.97	113726
macro avg	0.97	0.97	0.97	113726
weighted avg	0.97	0.97	0.97	113726

→ Gradient descent is a fundamental algorithm that allows for grasping the basic concepts of how Logistic Regression operates. However, it may not be sufficiently robust to tackle complex problems effectively on its own.

2. Support Vector Machine

In this part, we will explain our procedure of applying Support Vector Machine model on the classification of fraud credit transactions. Support Vector Machine (SVM) is a powerful machine learning algorithm used for linear or nonlinear classification, regression, and even outlier detection tasks. SVMs can be used for a variety of tasks, such as text classification, image classification, spam detection, handwriting identification, gene expression analysis, face detection, and anomaly detection. SVMs are adaptable and efficient in a variety of applications because they can manage high-dimensional data and nonlinear relationships. In this project, we divided SVM into 2 types: linear SVM and non-linear SVM.

2.1 Theory

Linear SVM

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression. The main objective of the SVM algorithm is to find the optimal hyperplane in an N-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries to ensure that the margin between the closest points of different classes should be as maximum as possible. The dimension of the hyperplane depends upon the number of features. If the number of input features is two, then the hyperplane is just a line. If the number of input features is three, then the hyperplane becomes a 2-D plane.

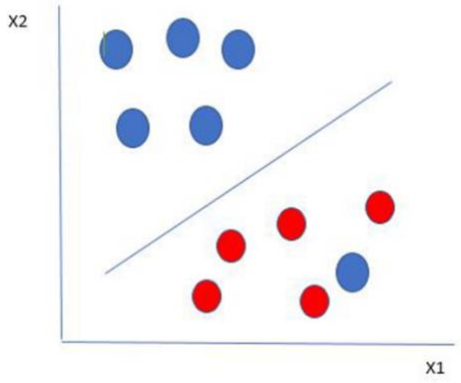


Figure 1: A SVM hyperplane of 2-dimensional space

One reasonable choice as the best hyperplane is the one that represents the largest separation or margin between the two classes. Therefore, the hyperplane whose distance from it to the nearest data point on each side is maximized is chosen. If such a hyperplane exists it is known as the maximum-margin hyperplane/hard margin.

The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin. SVM is robust to outliers.

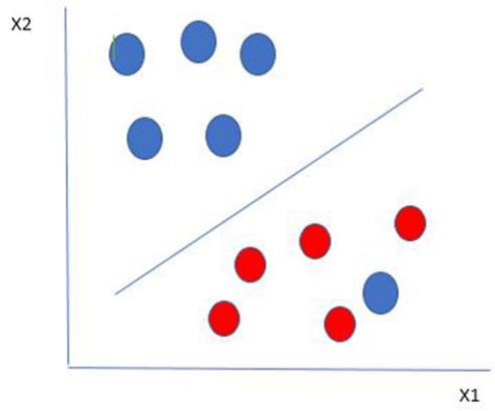
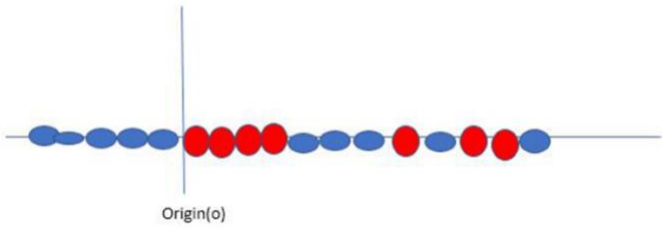


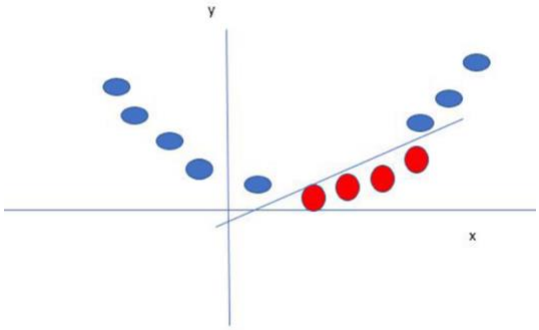
Figure 2: SVM algorithm ignores outliers.

In this type of data point, what SVM does is to find the maximum margin as done with previous data sets along with that it adds a penalty each time a point crosses the margin. The margins in these types of cases are called soft margins. When there is a soft margin to the data set, the SVM tries to minimize $\frac{1}{m \arg in} + C \sum penalty$. Hinge loss is a commonly used penalty. If no violations, no hinge loss. If violations hinge, loss proportional to the distance of violation.

Non-linear SVM



SVM solves the problem of non-linearly separable data by creating a new variable using a kernel. We call a point x_i on the line, and we create a new variable y_i as a function of distance from origin O. If we plot this, we get the graph as shown below.



In this case, the new variable y is created as a function of distance from the origin. A non-linear function that creates a new variable is referred to as a kernel.

b) *Support Vector Machine Terminology*

- ***Hyperplane***: Hyperplane is the decision boundary that is used to separate the data points of different classes in a feature space. In the case of linear classifications, it will be a linear equation i.e., wx+b = 0.
- ***Support Vectors***: Support vectors are the closest data points to the hyperplane, which plays a critical role in deciding the hyperplane and margin.
- ***Margin***: Margin is the distance between the support vector and hyperplane. The main objective of the support vector machine algorithm is to maximize the margin. The wider margin indicates better classification performance.
- ***Kernel***: Kernel is the mathematical function, which is used in SVM to map the original input data points into high-dimensional feature spaces, so, that the hyperplane can be easily found out even if the data points are not linearly separable in the original input space. Some of the common kernel functions are linear, polynomial, radial basis function (RBF), and sigmoid.
- ***Hard Margin:*** The maximum-margin hyperplane or the hard margin hyperplane is a hyperplane that properly separates the data points of distinct categories without any misclassifications.
- ***Soft Margin:*** When the data is not perfectly separable or contains outliers, SVM permits a soft margin technique. Each data point has a slack variable introduced by the soft-margin SVM formulation, which softens the strict margin requirement and permits certain misclassifications or violations. It discovers a compromise between increasing the margin and reducing violations.
- ***C:*** Margin maximization and misclassification fines are balanced by the regularization parameter C in SVM. The penalty for going over it decides the margin or misclassifying data items. A stricter penalty is imposed with a greater value of C, which results in a smaller margin and perhaps fewer misclassifications.
- ***Hinge Loss:*** A typical loss function in SVMs is hinge loss. It punishes incorrect classifications or margin violations. The objective function in SVM is frequently formed by combining it with the regularization term.
- ***Dual Problem:*** A dual Problem of the optimization problem that requires locating the Lagrange multipliers related to the support vectors can be used to solve SVM. The dual formulation enables the use of kernel tricks and more effective computing.

c) Mathematical intuition of Support Vector Machine

Consider a binary classification problem with two classes, labeled as 1 and 0.

The equation for the linear hyperplane can be written as:

$$w^T x + b = 0$$

where w is the weight vector, x is the input vector, and b is the bias term.

The distance between a data point x_i and the decision boundary can be calculated as:

$$d_i = \frac{w^T x_i + b}{||w||}$$

where ||w|| represents the Euclidean norm of the weight vector w.

For Linear SVM classifier:

$$\hat{y} = \begin{cases} 1 & : w^T x + b \geq 0 \\ 0 & : w^T x + b < 0 \end{cases}$$

Optimization:

- For Hard margin linear SVM classifier:

$$\underset{w,b}{\text{minimize}} \frac{1}{2} w^T w = \underset{W,b}{\text{minimize}} \frac{1}{2} \|w\|^2$$
$$\text{subject to } y_i(w^T x_i + b) \geq 1 \text{ for } i = 1, 2, 3, \cdots, m$$

The target variable or label for the training instance is denoted by the symbol t_i in this statement. And t_i=-1 for negative occurrences (when y= 0) and t_i=1positive instances (when y_i = 1) respectively. Because we require the decision boundary that satisfy the constraint:

- For Soft margin linear SVM classifier:

$$\underset{w,b}{\text{minimize}} \ \frac{1}{2} w^T w + C \sum_{i=1}^{m} \zeta_i$$
$$\text{subject to } y_i(w^T x_i + b) \geq 1 - \zeta_i \text{ and } \zeta_i \geq 0 \text{ for } i = 1, 2, 3, \cdots, m$$

Dual Problem: A dual Problem of the optimization problem that requires locating the Lagrange multipliers related to the support vectors can be used to solve SVM. The optimal Lagrange multipliers α_i that maximize the following dual objective function.

$$\underset{\alpha}{\text{maximize}} : \frac{1}{2} \sum_{i \to m} \sum_{j \to m} \alpha_i \alpha_j t_i t_j K(x_i, x_j) - \sum_{i \to m} \alpha_i$$

where,

- α_i is the Lagrange multiplier associated with the training sample.
- $K(x_i, x_j)$ is the kernel function that computes the similarity between two samples x_i and x_j. It allows SVM to handle nonlinear classification problems by implicitly mapping the samples into a higher-dimensional feature space.

The term $\sum\alpha_i$ represents the sum of all Lagrange multipliers.

The SVM decision boundary can be described in terms of these optimal Lagrange multipliers and the support vectors once the dual issue has been solved and the optimal Lagrange multipliers have been discovered. The training samples that have i > 0 are the support vectors, while the decision boundary is supplied by:

$$w = \sum_{i \to m} \alpha_i t_i K(x_i, x) + b$$
$$t_i(w^T x_i - b) = 1 \Longleftrightarrow b = w^T x_i - t_i$$

d) Popular kernel functions in SVM

The SVM kernel is a function that takes low-dimensional input space and transforms it into higher-dimensional space, i.e., it converts non separable problems to separable problems. It is mostly useful in non-linear separation problems. Simply put the kernel, does some extremely complex data transformations, and then finds out the process to separate the data based on the labels or outputs defined.

$$\text{Linear: } K(w, b) = w^T x + b$$

$$Polynomial: \ K(w, b) \ = \ (\gamma w^T \ + \ b)^N$$

$$Gaussian\ RBF: K(w, b) = \ exp(\ -\gamma \ ||x_i \ - \ x_j||^n)$$

$$\text{Sigmoid}: \ K(x_i, x_j) \ = \ tanh(\alpha {x_i}^T x_j \ + \ b)$$

2.2 Implementation

Model Setup

In this part of the project, we apply SVC function from Scikit-learn library to train the model with hyperparameters we got by Grid Search Method.

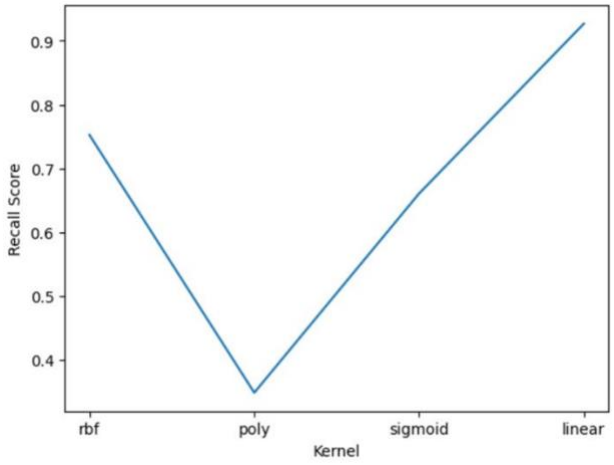
Hyperparameter tuning

We use GridSearchCV function to train and evaluate our machine learning model using a combination of three parameters: Kernel, Regularization and Gamma. As our dataset at this stage is now balanced, we will use the positive recall metric as our main evaluation metric.

- **Kernel**

The learning of the hyperplane in SVM is done by transforming the problem using a parameter called kernel to specify which type of SVM kernel function will be used.

Below is the graph indicating the recall score for each kernel:



Look at the line graph, we can see that the linear kernel is the most appropriate choice for the model.

- **Regularization**

  The Regularization parameter (often termed as C parameter in python's Scikit-learn library) tells the SVM optimization how much you want to avoid misclassifying each training example.

  For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a ridiculously small value of C will cause the optimizer to look for a larger margin separating hyperplane, even if that hyperplane misclassifies more points.

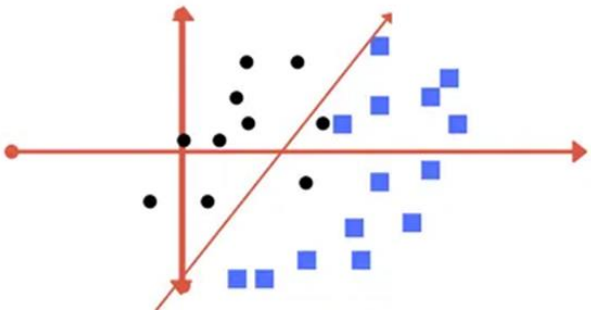


Figure 3: Lower regularization value

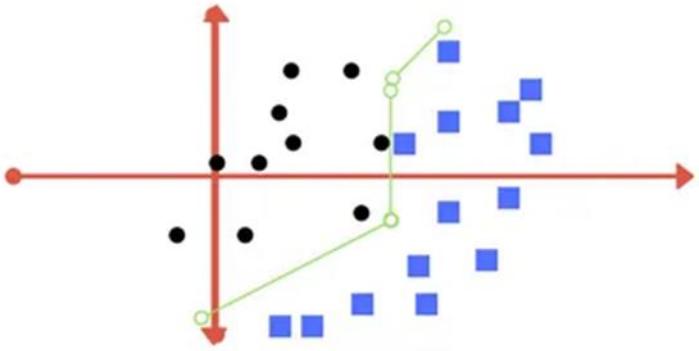
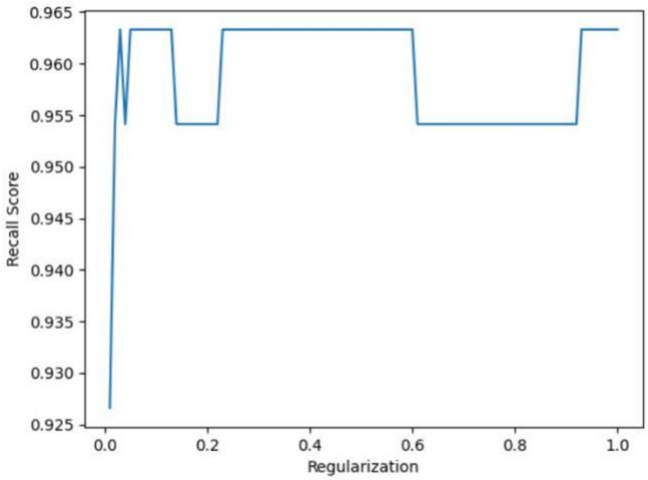


Figure 4: Higher regularization value

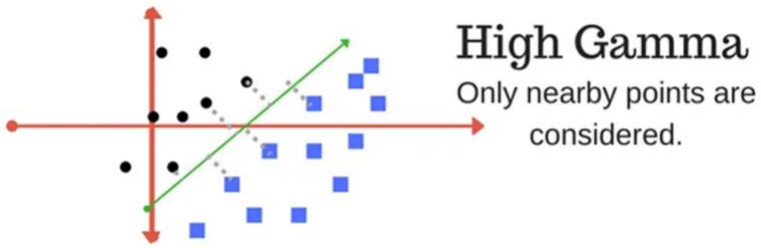
Below is the graph indicating the recall score for a range of regularization:

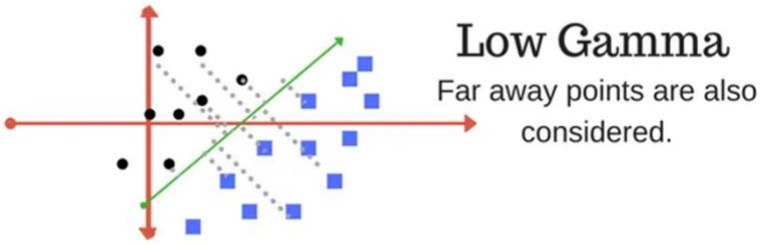


The chart shows how regularization affects the recall score. We can see that the recall score comes to the highest at the point C = 0.02 and then becomes unstable due to overfitting.

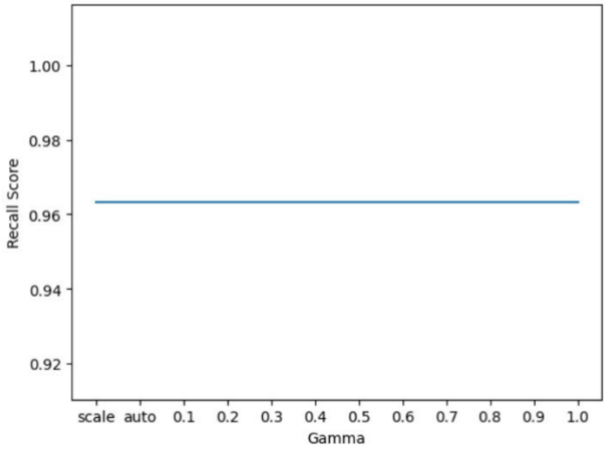
- **Gamma**

The gamma parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close.’ In other words, with low gamma, points far away from plausible separation line are considered in calculation for the separation line. Whereas high gamma means the points close to plausible line are considered in calculation.





Here is the graph that shows the relation between recall score and gamma:



From the above line chart, we can see that the performance of the model does not clearly rely on gamma.

2.3 Result

Undersampling:

	precision	recall	f1-score	support
Not fraud	0.92	0.98	0.95	88
Fraud	0.98	0.92	0.95	92
accuracy			0.95	180
macro avg	0.95	0.95	0.95	180
weighted avg	0.95	0.95	0.95	180

Oversampling:

	precision	recall	f1-score	support
Not fraud	0.91	0.99	0.95	272236
Fraud	0.99	0.89	0.94	241788
accuracy			0.95	514024

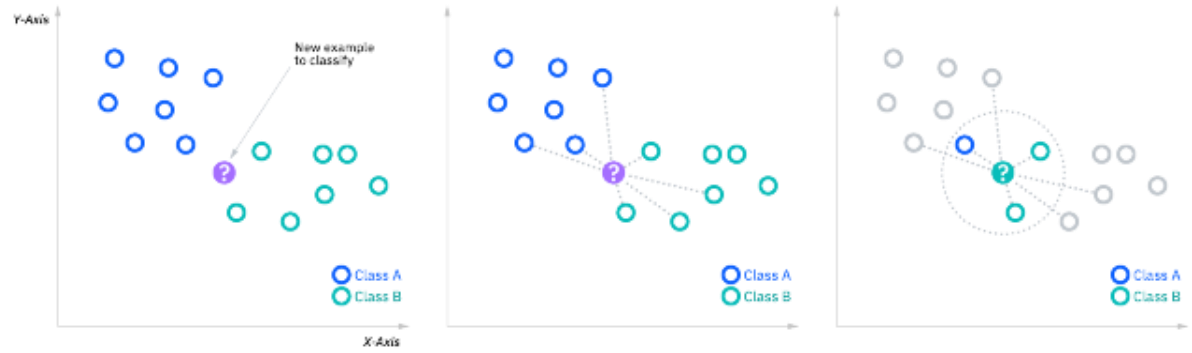
macro avg	0.95	0.94	0.94	514024
weighted avg	0.95	0.95	0.95	514024

3. K-Nearest Neighbors

3.1. Theory

The K nearest neighbors (KNN) algorithm is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. It is one of the popular and simplest classification and regression classifiers used in machine learning today. It is widely disposable in real-life scenarios since it is non-parametric, meaning it does not make any underlying assumptions about the distribution of data.

The KNN algorithm works by finding the K nearest neighbors to a given data point based on a distance metric, such as Euclidean distance. The class or value of the data point is then determined by the majority vote or average of the K neighbors. This approach allows the algorithm to adapt to different patterns and make predictions based on the local structure of the data.



3.2. Implementation

There were two problems we had to deal with when working with KNN algorithm, that were, how do we calculate the distances between datapoints and what is the optimal value of k.

To solve the distance problem, we listed some popular distance functions and selected between them through trial and error:

- Euclidean Distance:

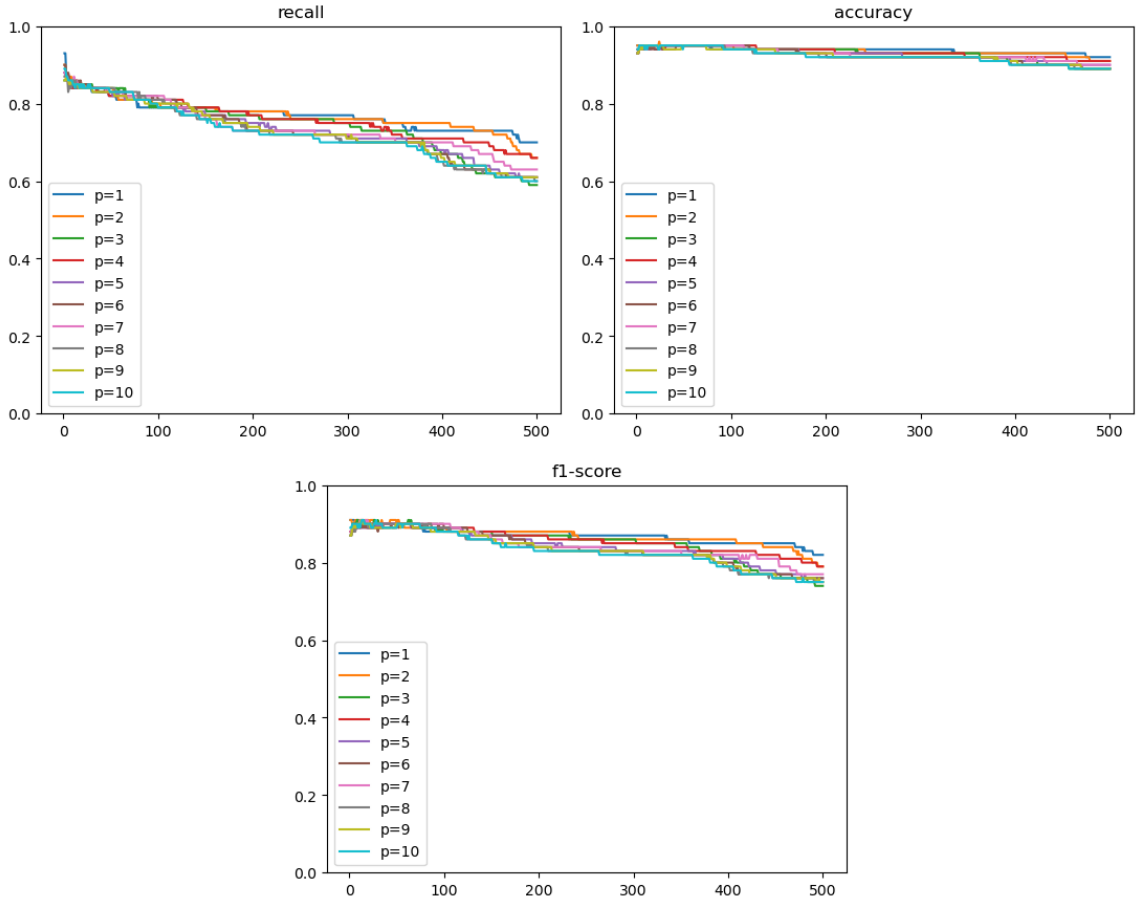
$$d(x, y) = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2}$$

- Manhattan Distance:

$$d(x,y) = \sum_{i=1}^{n} |x_i - y_i|$$

- Minkowski Distance: We can say that the Euclidean (with p = 2), as well as the Manhattan distance (with p = 1), are special cases of the Minkowski distance.

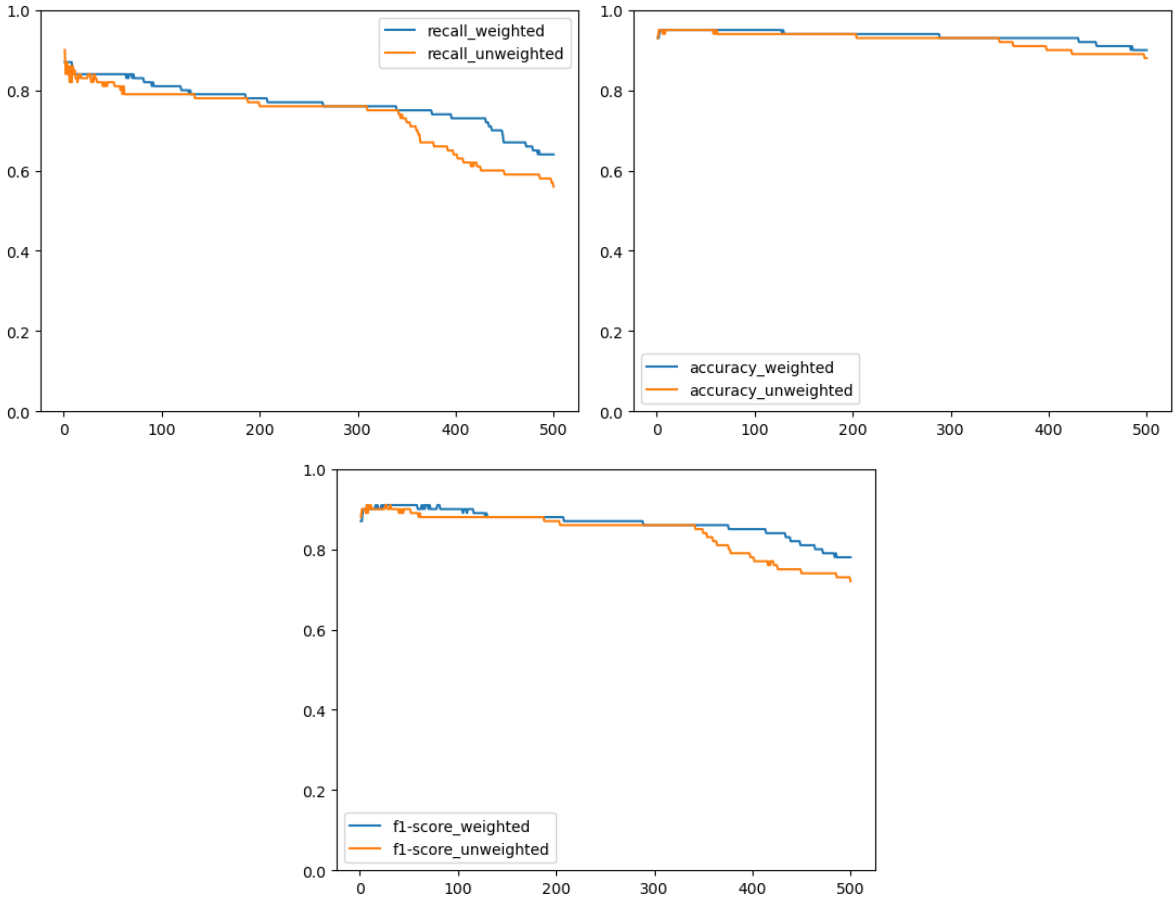
$$d(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$



After testing we saw that while lower values of p in the Minkowski distance function gave better results, the difference was neglectable given a reasonable value of k. For the results we chose p = 2, corresponding to the Euclidean distance function, as the final value.

For the problem of choosing k, we also can try many values to see which one yells the best result. However, instead of treating all nearest neighbors of a data point equally, we can weight them using a function that is inversely proportional to the distance between them, making the vote of further neighbors less important and thus reducing the error of k. The simplest way to obtain a weight function like that is to take the inverse of the distance between two data points:

$$w = \frac{1}{d(x,y)}$$



The graphs above show that while the quality of both sides starts high and drops off as k grows, the one with weighted votes remains more accurate especially when k is big, thus allowing more freedom when choosing k. For the results we chose k = 50 as the final value.

3.3. Results

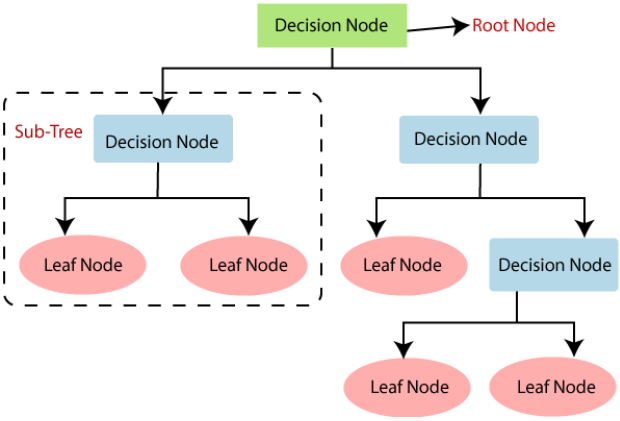
KNN algorithm with 50 nearest neighbors and Euclidean distance:

	precision	recall	f1-score	support
Not fraud	0.95	0.99	0.97	56863
Fraud	0.98	0.84	0.91	56863
accuracy			0.95	113726
macro avg	0.96	0.91	0.94	113726
weighted avg	0.96	0.95	0.95	113726

4. Decision Tree

4.1 Theory

Decision Tree is a Supervised learning technique that can be used for both classification and regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes, also called Decision nodes, represent the features of a dataset, multi branches represent the decision rules and each leaf node represents the outcomes of those decisions and do not contain any further branches.



Decision Trees are easy to understand, interpret, and visualize, making them a powerful tool for data analysis and prediction.

How do decision tree algorithms work?

Decision Tree algorithm works in some steps:

1. Starting at the Root: The algorithm begins at the top, called the “root node,” representing the entire dataset.
2. Asking the Best Questions: It looks for the most important feature or question that splits the data into the most distinct groups. This is like asking a question at a fork in the tree. But how can we know that is the “most important feature”? We can obtain using Attribute Selection Measure (ASM), making comparison between each attribute we choose and then selecting the best one.
3. Branching Out: Based on the answer to that question, it divides the data into smaller subsets, creating new branches. Each branch represents a possible route through the tree.
4. Repeating the Process: The algorithm continues asking questions and splitting the data at each branch until it reaches the final “leaf nodes,” representing the predicted outcomes or classifications.

Now, let's dive deeper into Attribute Selection Measures (ASM), which are crucial when implementing a Decision Tree. These measures help us choose the best attribute for the nodes of the tree. The two most popular techniques for ASM are:

1. **Information Gain**
2. **Gini Index**

Information Gain

Information Gain is a key component of the **ID3** (Iterative Dichotomiser 3) algorithm. It uses a terminology called **Entropy**:

Entropy measures the impurity or inhomogeneity of a set. For a set SS with cc classes or labels, the entropy is defined as:

$$Entropy(S) = -\sum_{i=1}^{c} p_i \log_2 p_i$$

where p_i is the proportion of instances with class label i in S. The sum of all p_i values equal 1: $p_1 + p_2 + \ldots + p_c = 1$

For example, if we have a set S = $\{ c_1{:}21, c_2{:}9\}$, $then$ $Entropy(S) = \frac{-21}{30} \log_2 \frac{21}{30} + \frac{-9}{30} \log_2 \frac{9}{30} = 0.881$

Entropy can measure the impurity of a set due to it represent value:

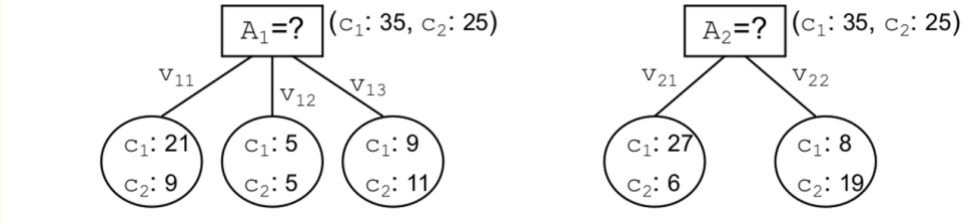
$$\text{If } Entropy(S) = \begin{cases} 0 \; if \; all \; sample \; in \; S \; have \; \; the \; same \; label \\ 1 \; if \; all \; classes \; in \; S \; have \; the \; same \; number \; of \; sampless/size \\ in \; range \; (0,1) \; otherwise \end{cases}$$

Information Gain: measures the reduction in entropy when a set S is divided into subsets based on an attribute A. The formula for Information Gain is:

$$Gain(S, A) = Entropy(S) - \sum_{v \, \varepsilon \, Values(A)} \frac{|\, S_v|}{|\, S \,|} \, Entropy(S_v)$$

Where values(A) are the set of all values of A and $S_v = \{ x | \; x \; in \; S \; and \; x_a = V\}$

The larger the Information Gain value, the more useful the attribute is for splitting the data, as it provides a greater reduction in entropy. Let us illustrate this with an example:



$$Entropy(A1) = \frac{-35}{60}\log_2\frac{35}{60} + \frac{-25}{60}\log_2\frac{25}{60} = 0.97$$

$$Entropy(A2) = 0.97$$

Now, we calculate $Gain(A_1, A_{1i}) = 0.97 - \sum_{\{v_{11},v_{12},v_{13}} \frac{|A_{1i}|}{|A_1|} Entropy(A_{1i}) = 0.97 + \frac{-30}{60} Entropy(A_{11}) + \frac{-10}{60} Entropy(A_{12}) + \frac{-20}{60} Entropy(A_{13}) = \ldots. = 0.03284$

Using the equivalent way, we have $Gain(A_2, A_{2i}) = 0.2045$

Since $Gain(A_1, A_{1i})$ is smaller than $Gain(A_2, A_{2i})$, so in this case, dividing it by attribute A_2 will be a better option.

Applying Entropy and Information Gain with the ID3 Algorithm

Now that we have a measure for selecting the best attribute, we can use the ID3 algorithm to build our decision tree. The ID3 algorithm is a greedy method that operates as follows:

1. **At each node N**, select the test attribute A that provides the highest Information Gain. This attribute will best classify the data at node N.
2. **Generate a branch** for each value of A.
3. **Repeat the process** for each branch until:
    - All training data is correctly classified (i.e., all instances in each leaf node belong to the same class).
    - All attributes have been used (since in the ID3 algorithm, an attribute can only be used once along a path from the root).

Quinlan's ID3 algorithm is the simplest and fundamental form of decision tree algorithms. However, the algorithm uses all the features that are given to it, even if some of them are not necessary. This obviously runs the risk of overfitting; indeed, it makes it highly likely. Several updated variants and extensions to improve this disadvantage, such as C4.5 and CART, are widely used instead. By default, the DecisionTree in Scikit-Learn uses the **CART** algorithm.

**Since ID3 prioritizes selecting attributes with high classification power near the root, these attributes tend to have many values. As a result, the tree often has a short and wide shape because it branches out significantly at the early levels. This can lead to high computational costs later*

and makes the model prone to overfitting. For example, when categorizing students in a class, if we start by splitting based on their birthdate, the data might be well classified after this step, but the first level of the tree could have up to 365 branches and this feature also does not useful for future data.

Gini Index

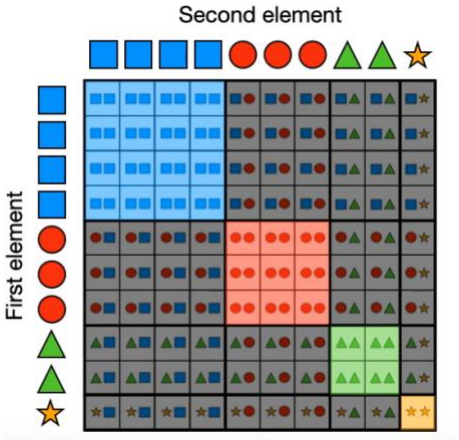
The Gini Index measures the diversity of a set, representing the probability of picking two distinct elements from the set. It is used by the CART (Classification and Regression Tree) algorithm to decide how to split the data at each node of the tree. The Gini impurity of a set S with c classes is calculated using the following formula:

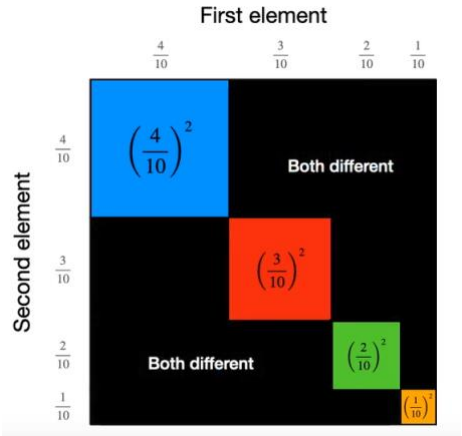
$$G(S) = 1 - \sum_{i=1}^{c} (p_i)^2$$

where p_i is the proportion of class i in the set S.

The higher the Gini index, the greater number of classes that all instances in that set belong to.

Here's why we use this formula: Since the Gini index represents the probability of picking two distinct elements in the set, this is expressed as: P(both different) = P(Everything) – P(both equal) = 1 - P(both equal), and P(both equal) is the sum of P(both equal for each class), which is the power two of proportion of that class i^{th} in set S. This can be simple to explain when we represent every possible way, we choose any two elements arbitrary from the set, in form of a matrix. For example:





Applying Gini index with the CART Algorithm

Cart algorithm works quite like the ID3, so we will not go in that detail anymore, here we just mention some of key different:

	ID3	CART
Splitting Criterion	Information Gain	Gini impurity (For classification), MSE (for Regression)
Splitting type	Multi-way splits	Binary splits
Tree pruning	Typically, not built-in, requires separate post-pruning techniques	Built-in through cost-complexity pruning (CCP), which we will evaluate in parameter tuning section
Attribute types	Handles categorical attributes directly	Requires conversion of categorical attributes to numerical

→ We see some advantages of CART like it is simpler, more interpretable model, and with built-in pruning mechanism, it is more likely to avoid Overfitting.

4.2 Implementation

Hyperparameter turning

To avoid Overfitting of this decision tree model, we can use two methods:

- **stop learning early (Pre-prune)**
- **prune the fill tree (post-prune)**

1. Pre-prune

The hyperparameters that can be tuned for early stopping and preventing overfitting are criterion, max_depth, min_samples_leaf, and min_samples_split.

These same parameters can also be used to tune to get a robust model. However, you should be cautious as early stopping can also lead to underfitting.

- ***criterion***: parameter in the DecisionTreeClassifier determines the function to measure the quality of a split. Scikit-learn supports two criteria: "gini" for the Gini impurity and "entropy" for the information gain.

- ***max_depth***: The maximum depth of the tree. This is a way to control over-fitting. The deeper the tree, the more splits it has, and it captures more information about the data. None means that nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

- ***min_samples_split***: The minimum number of samples required to split an internal node. This can vary between considering at least one sample at each node to considering all the samples at each node. When we increase this parameter, the tree becomes more constrained as it considers more samples at each node.
- **min_samples_leaf**: The minimum number of samples required to be at a leaf node. This parameter is like min_samples_splits, however, this describes the minimum number of samples of samples at the Leaves, the base of the tree.

- ***max_features***: The number of features to consider when looking for the best split. If None, then max_features=n_features, set to 'sqrt', then sqrt(n_features) features are considered at each split. This is often used in Random Forests also, where it helps to make the individual trees more independent of each other and thus increases the diversity of the forest.

Here are the parameters list we want to choose the best one:

```
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [10, 15,20],
    'min_samples_split': [2, 5, 10, 15],
    'min_samples_leaf': [1, 2, 5],
    'max_features': ['sqrt', 'log2']
}
```

Using GridSerachCV and measures on the best F1 score, we get the best result as follow:

Best parameters: {'criterion': 'entropy', 'max_depth': 15, 'max_features': 'sqrt', 'min_samples_leaf': 2, 'min_samples_split': 5}
Best score: 0.9958130411936044

In Scikit-learn, GridSearchCV is a tool that automates find the best combination of hyperparameters for a machine learning model, performing an exhaustive search over a specified parameter grid, and evaluating the performance of each combination using **cross-validation.*

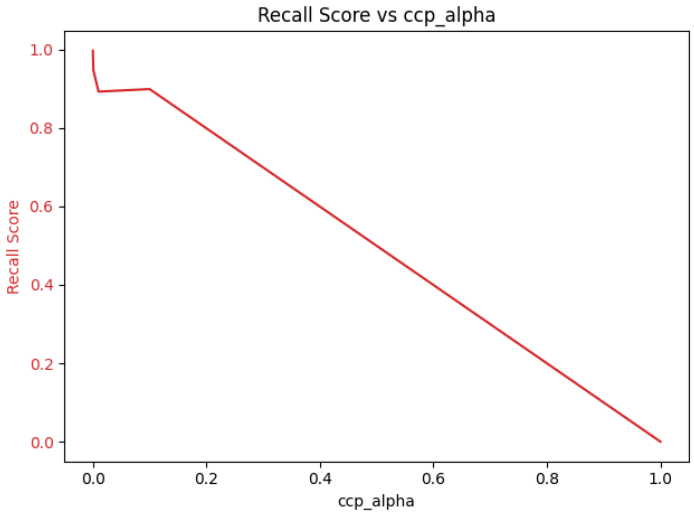
2. Post-prune

Pruning is a technique used to reduce the size of decision trees by removing sections that provide little power in classifying instances. Pruning results in a smaller, simpler, and more interpretable tree and helps reduce overfitting, making it generalize better to unseen data.

After the tree is fully grown, post-pruning involves removing branches or nodes to improve the model's generalization ability. Common post-pruning techniques include:

- ***Cost-Complexity Pruning (CCP)***: Assigns a cost to each subtree based on its accuracy and complexity, then selects the subtree with the lowest cost.
- *Reduced Error Pruning:* Removes branches that do not significantly affect the overall accuracy.
- *Minimum Impurity Decrease*: Prunes nodes if the decrease in impurity (Gini impurity or entropy) is below a certain threshold.
- *Minimum Leaf Size:* Removes leaf nodes with fewer samples than a specified threshold.

Scikit-learn provides the DecisionTreeClassifier and DecisionTreeRegressor classes with built-in support for post-pruning using the **ccp_alpha** parameter (cost complexity pruning). We try some value of ccp_alpha values and plot this graph:



A higher value of ccp_alpha makes the model run faster, but at the cost of prediction quality. In our project, the dataset1 is small and, also as demonstrated in the last section of this report, our model does not seem to face overfitting problems. Therefore, post-pruning the tree in our case is not necessary.

Run Model

a. Decision Tree using ID3 algorithm

Scikit-learn DecisionTreeClassifier does not directly support the ID3 algorithm. It uses an optimized version of the CART (Classification and Regression Trees) algorithm, which is like ID3 but uses a different metric (Gini impurity or entropy) for splitting the nodes. To use the ID3 algorithm, we need to use a different library, such as the **decision-tree-id3** package.

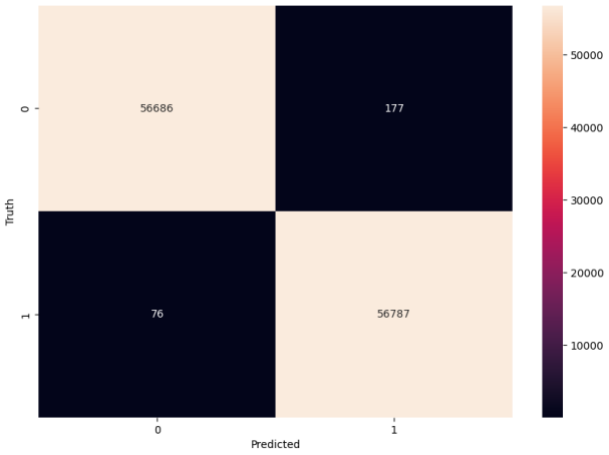
However, our team has encountered several challenges while trying to run this code due to the incompatible version of this library with `six` module in Scikit-learn and NumPy. More details about this can be read in the source code file.

b. Default Decision Tree using CART algorithm.

Detail about this model and its parameters can be found here

4.3 Result

	precision	recall	f1-score	support
Not fraud	1.00	1.00	1.00	56863
Fraud	1.00	1.00	1.00	56863
accuracy			1.00	113726
macro avg	1.00	1.00	1.00	113726
weighted avg	1.00	1.00	1.00	113726



Confusion matrix from Decision Tree model

5. Random Forest

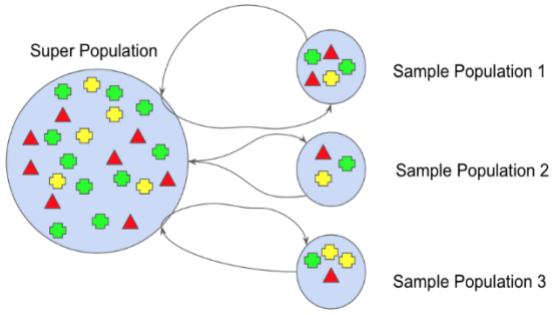
5.1 Theory

Random Forest is an ensemble learning method used for classification, regression, and other tasks that operates by constructing multiple decision trees during training and outputting the mode of the classes (classification) or mean prediction (regression) of the individual trees. It has become one of the most popular machine learning algorithms due to its simplicity and effectiveness.

The primary difference between Random Forest and a single decision tree lies in the ensemble approach. Random Forest combines multiple decision trees, each trained on a random subset of data and features, to reduce overfitting and improve prediction accuracy. Decision tree, on the other hand, builds a single decision tree based on the entire dataset, which can be more prone to overfitting. We will explain this mechanism in more detail through the work of Random Forest.

Here is how Random Forest work:

1. ***Bootstrap Sampling:***



Random Forest uses a technique called bootstrap aggregating (or bagging) to create multiple subsets of the training data. Each subset is created by randomly sampling the data points with replacement. This means some data points may appear multiple times in a subset, while others may not appear at all. Each subset then will be used to train an individual decision tree, resulting in more accurate and less overfitting-prone model.

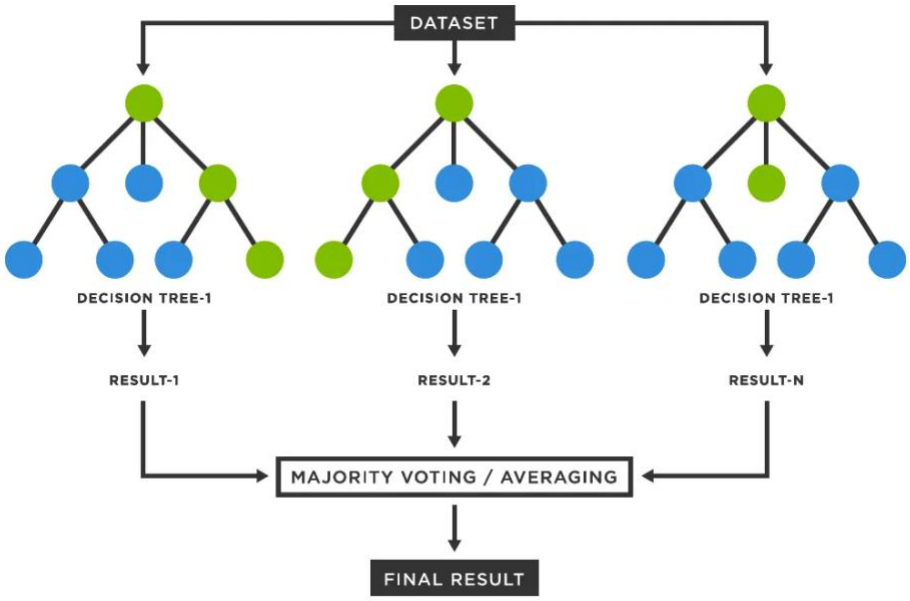
2. ***Building trees with random subspace:***

For each subset from bootstrap sampling, a decision tree is built. However, unlike traditional decision trees, Random Forest introduces further randomness by selecting a random subset of features at each split. By allowing each tree to focus on a different subset of features, this process helps to reduce the correlation between the individual trees and increases the diversity of the model, which improves overall performance.

3. ***Aggregation:***

For classification tasks, the final output is determined by the majority vote of the individual trees. For regression tasks, the final output is the average of the predictions from all the trees.

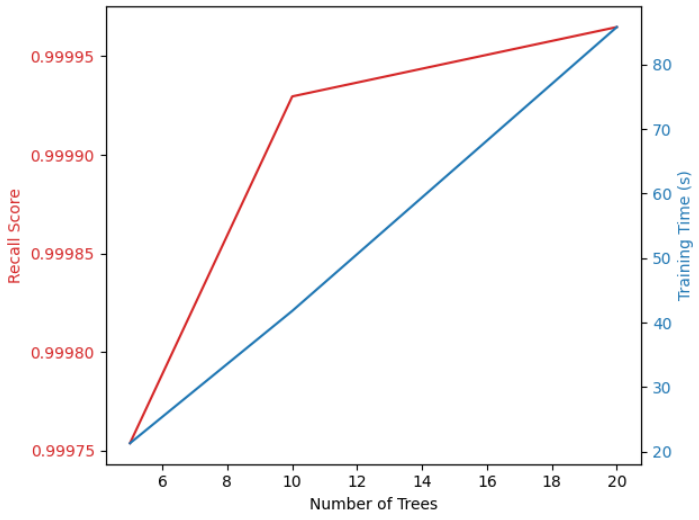
The overall process can be described by this graph:



5.2 Implementation

Hyperparameter tuning

We plot the graph to evaluate number of tree *tree_counts = [5, 10, 20]* with the corresponding training time. The greater number of trees used, the more accuracy (maybe) but trade of the training time



Based on the graph, it is evident that the Recall score exhibits a narrow range, hovering around 1.0, with variations of less than 0.00001 across different measures. Therefore, we will opt for a modest number of trees, specifically 5, to have shortest training times and potentially mitigate the risk of overfitting.

Run Model

In this section, we will implement two models of Random Forest.

- The first model will be based on the theoretical concepts discussed earlier to help us understand how it works, which we will manually perform bootstrap sampling, build decision trees, and aggregate their predictions.
- The second model will use the default Random Forest implementation provided by scikit-learn.

a. Manually Random Forest

First, we create three bootstrap sample form the dataset:

```
bootstrap_samples = []
for i in range(3):
    bootstrap_sample = df.sample(n=len(df), replace=True)
    bootstrap_samples.append(bootstrap_sample)
```

Then we train three individual decision trees with corresponding subsets. We use *max_features* for randomly subspace (it determines the maximum number of features to consider when looking for the best split at each node):

```
from Scikit-learn.tree import DecisionTreeClassifier
for sample in bootstrap_samples:
    dt_model = DecisionTreeClassifier(max_features='sqrt') # Random subspace
    # Extract X,y from sample ...
    dt_model.fit(X, y)
    dt_models.append(dt_model)
```

After training, we use majority voting to decide the final prediction:

- If vote1 + vote2 + vote3 = 1 means, there are two 0 and only one 1 -> final result is 0
- If vote1 + vote2 + vote3 = 2 means, there are two 1 and only one 0 -> final result is 1

```
for j in range(len(predictions[0])): #take length of the first tree prediction list
  n_ones = predictions[0][j] +  predictions[1][j]   + predictions[2][j]
  if n_ones > 1:
     final_result = 1
  else:
     final_result = 0
```

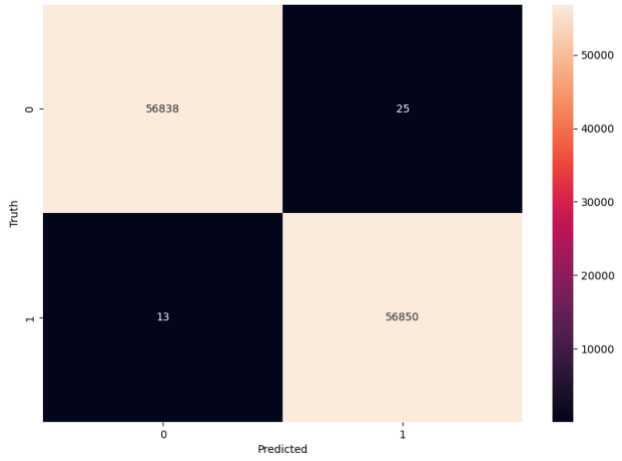
b. Default Random Forest in Scikit-learn

The Random Forest implementation in scikit-learn operates similarly to the theoretical approach but employs advanced techniques for enhanced computational and memory efficiency. It often leverages compiled code (e.g., Cython), parallel processing, and automatic selection of optimal hyperparameters. In this model, we just tune the number of decision trees since the default setting in scikit-learn (version 0.22 and above) is 100 trees, which we believe exceeds our needs.

Details about this model and its parameter can be found here.

5.3 Result:

	precision	recall	f1-score	support
Not fraud	1.00	1.00	1.00	56863
Fraud	1.00	1.00	1.00	56863
accuracy			1.00	113726
macro avg	1.00	1.00	1.00	113726
weighted avg	1.00	1.00	1.00	113726



Confusion matrix of Random Forest model

6. Artificial Neural Network (MLP)

In this part, we shall discuss our approach of applying Artificial Neural Network (ANNs) model on the problem of classifying fraudulent and legitimate transactions. ANNs is a powerful and flexible machine learning technique that can model complex relationships in a dataset, dealing with a large amount of input. In the context of this project, we focus on 2 certain types of ANN, the Multi-layer Perception (MLP) model and the Deep Neural Network (DNN).

6.1 Theory

Artificial Neural Networks is a machine learning model inspired by the human brain's activities. A typical ANN architecture comprises of 3 main components:

- Neurons (nodes): A basic unit that receives the input, weights, and passes the results through an activation function to give out the result.
- Layers: ANNs are composed of neurons layers. The output from a single neuron will not be able to model complex tasks. So, to handle more complex structures, neurons layers are implemented so the class of functions can become more complicated. A ANNs structure usually consists of three layers: an input layer, hidden layers, and an output layer.

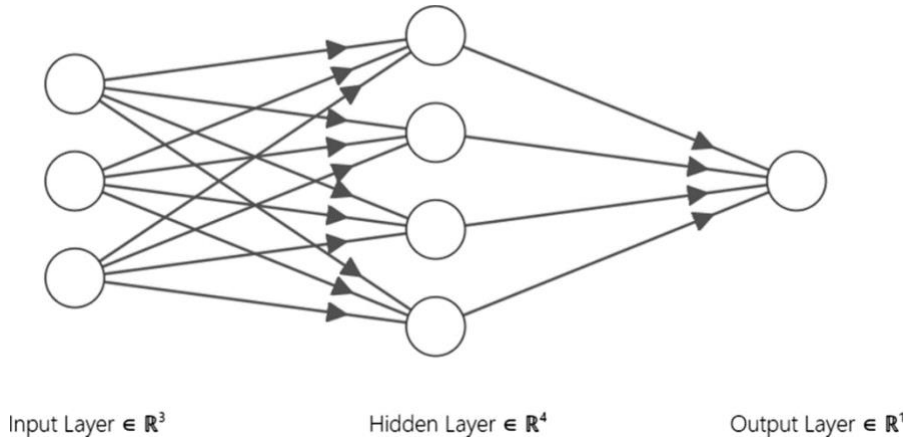


Figure 6.1 A neural network structure

- The input layer takes input from the dataset, passing the input value through to the next layer.
- Hidden layers: layers between the input and output layer. It takes in a set of weighted input and produces an output through an activation function.
- Output layer: the final layer, responsible for outputting the required values.
- Weights: The value assigned to each connection between neurons represents the strength of the connection between units to minimize error.

Assume that a neural network model has m number of neurons, we can calculate the total weights arrive at a certain neuron, add up to see if there is enough strength to make the neuron fire:

$$h = \sum_{i=1}^{m} w_i.x_i$$

The following is a summary of how the algorithm works:

1. **Initialization:** an input vector is put into the input nodes, set all the network's weights(W) and biases (B) to their initial value.

2. **Forward propagation:** the inputs are fed forward through the network repeatedly. The inputs and the first-layer weights are used to decide whether the hidden nodes fire or not. The outputs of these neurons and the second-layer weights are used to decide if the output neurons fire or not.
3. **Loss calculation:** the error (loss) is computed as the sum-of-squares difference between the networks and the targets.
4. **Backpropagation:** the error is fed backwards through the network to reduce the loss by modifying its biases and weights.

Hyperparameters are the variables set before the training process and cannot be learned during training. ANNs have plenty of hyperparameters, which make them quite flexible. However, it also makes the model tuning process more difficult. Some notable hyperparameters are:

- Number of hidden layers and nodes: The number of hidden layers and nodes primarily depends on the complexity of the task. Too few nodes and layers might lead to high errors while too many nodes and layers result in overfitting to the training data.
- Learning rate: The learning rate determines how quickly we want to update our weight values. It should be high enough to be able to converge in a reasonable amount of time, while low enough to find the minimum value of the loss function.
- Activation functions: the function used over the weighted sum of inputs in the ANNs to get the desired output. Activation functions allow the network to combine the inputs in more complex ways.
- Cost (loss) functions: measure how well the ANN fits to the empirical data.
  Because the problem is a binary classification problem, we apply Binary Cross Entropy Loss function as it is a default function for this type of problem:

$$L\left(y,\hat{y}\right) = -\frac{1}{N}\sum_{i}^{N}\left[y_i log\left(\hat{y}_i\right) + \left(1 - y_i\right) log\left(1 - \hat{y}_i\right)\right]$$

  Here, y_i is the true output for the i^{th} sample, which can be either 0 or 1, and $\hat{y}$ is the predicted output for the i^{th} sample, which is a probability value between 0 and 1. N is the number of samples.

- Optimizers: Optimizers update the weight parameters to minimize the loss function. There are many optimizers that can be used for different purposes. Some optimizers are widely used and famous for certain types of problems. For example, Stochastic Gradient Descant (SGD) is as follows:

Algorithm 1: Stochastic Gradient Descent (SGD)

Given: L the loss function,
Input: Training sample $(x_{(train)}, y_{(train)})$, regularization parameters $\Omega(\theta)$, learning rate α, λ is a parameter to controlate the penalization importance, initialize θ
Output: Model parameters $\hat{\theta} = (\mathbf{b}, \mathbf{W})$
Initialize $\theta = \{W^{(1)}, b^{(1)}, ..., W^{(L+1)}, b^{(L+1)}\}$;
repeat
  for $i \in n_{epochs}$ **do**
    Given a training set $(x_{(train)}, y_{(train)})$
    Compute: $\Delta = -\frac{\partial}{\partial\theta} l(f((x_{(train)}; \theta), y_{(train)})) - \lambda \frac{\partial}{\partial\theta} l(\Omega(\theta))$
    $\theta_i \leftarrow \theta_i + \alpha\Delta$
  end
until *stopping criterion/ convergence*;
where: $\frac{\partial}{\partial\theta} l(f((x_{(train)}; \theta), y_{(train)}))$ is the function gradient

The algorithm for Stochastic Gradient Descent

Multi-layer Perceptron is a certain type of Artificial Neural Networks, with some special modified features. MLP can be classified as a fully connected neurons with a non-linear kind of activation function.

6.2 Implementation

Model Setup

For this part of the project, we define an MLPClassfier library, specifying the architecture of the neural network, including all the hyperparameters.

Hyperparameter Tuning

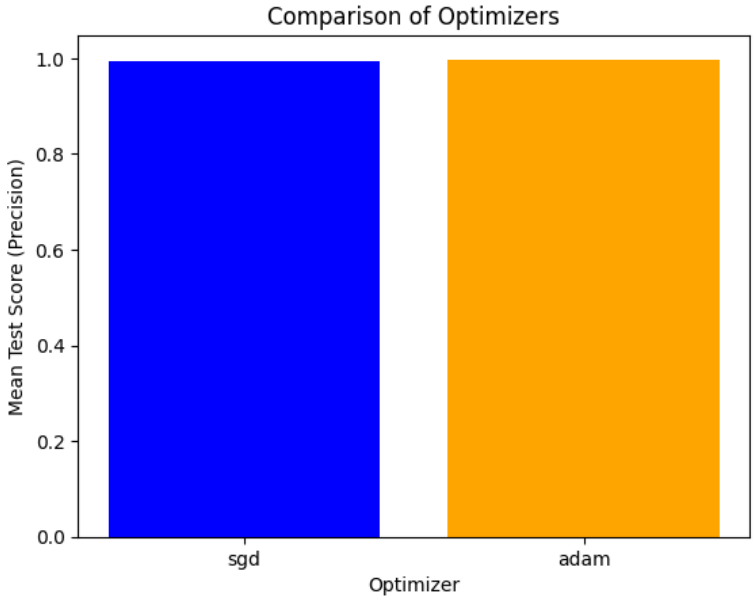
To proceed with the hyperparameter tuning process, we used 'GridSearchCV,' a scikit-learn function that performs tuning by training and evaluating a machine learning model using different combinations of parameters. We will evaluate 3 hyperparameters: learning rate, size, and number of hidden layers. As our dataset at this stage is now balanced, we will use the positive recall metric as our main evaluation measurement metric.

```
parameter_space = {
    'hidden_layer_sizes': [(50,), (40, 20, 10, 5)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant', 'adaptive'],
    'learning_rate_init': [0.001,0.005, 0.05, 0.065,0.1,0.5],}
```

Picture: different hyperparameters for ANN

- **Optimizer**

Optimizers are one of the fundamentals of neural networks with an aim to study the features of the input data, then find a couple of weight-bias suitable for the dataset given. In our model, we evaluate 2 popular optimizers: 'Adaptive Moment Estimation (Adam)' and 'Stochastic Gradient Descent (SGD)'. The results of two optimizers with their corresponding accuracy with different learning rate is as follows:



Graph: normalized Mean Test Score for different optimizers

As the graph and the code shows, 'Adam' performs slightly better than 'Sgd' with different learning rates. 'Sgd' focuses more on generalizing the data at the cost of low computation speed, while 'adam' pays more attention to faster computation time.

So, we chose 'Adam' as our optimizer.

- **Learning rate**

As mentioned above, the learning rate is a hyperparameter that controls the step size for a model to reach the minimum loss function. There is a tradeoff between lower learning rate needs higher epochs, more time or more memory capacity resources. The issue with tuning the learning rate is that they all depend on hyperparameters that must be manually chosen for each given learning session and may vary greatly.

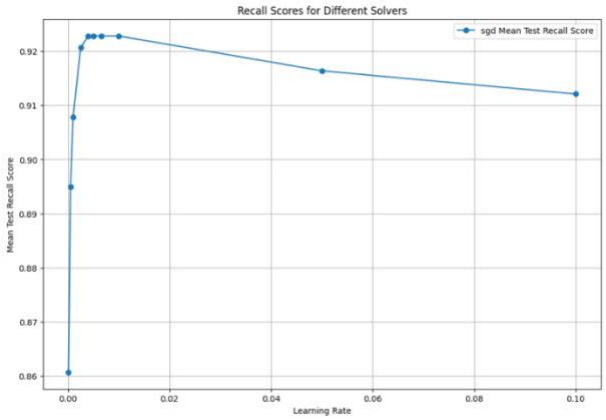


Figure 6.2 Learning rate of models with accuracy metric

We run our model on different learning rates: low learning rates of (0.001,0.005), medium learning rates of (0.05) and high learning rate of (0.1) along with the mean test accuracy of the model. As we can see from the graph, the medium learning rate of 0.001 proves to be the best choice, balancing the exploration and exploitation during training.

Number of hidden layers

We run out model using two choices of numbers of hidden layers: the first contains 4 hidden layers with 40 nodes on the 1st layer, 20,10,5 on the next 3 layers, while the second contains 2 hidden layers containing 100 nodes and 50 nodes.

Number of hidden layers		(40,20,10,5)	(40)
Training Set	Accuracy	0.99988	0.99966
	Precision	0.99976	0.99933
	F1 Score	0.99988	0.99966
	Recall	1.00000	1.0000
Test Set	Accuracy	0.99906	0.99306
	Precision	0.64285	0.64285
	F1 Score	0.69230	0.69230
	Recall	0.75000	0.7500

The first choice of (40) proves to perform better in our given condition. Because our dataset is quite simple and not complex, the differences in accuracy and precision are not too high. Therefore, we chose to use the four hidden layers option.

6.3 Results

Our final Artificial Neural Network model has the following parameters:

```
parameter_space = {
    'hidden_layer_sizes': [(40)],# the size of the networks
    'activation': ['relu'], # The activation function
    'solver': ['adam'], # The optimizer
    'alpha': [0.01], #L2 regularization
    'learning_rate': ['adaptive'], #leraning rate
    'learning_rate_init': [0.001],}
```

The following is the results of the ANN model:

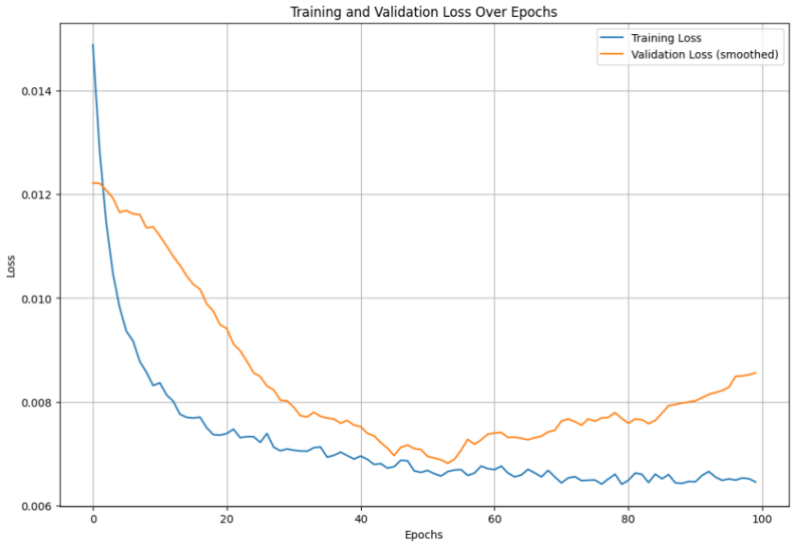
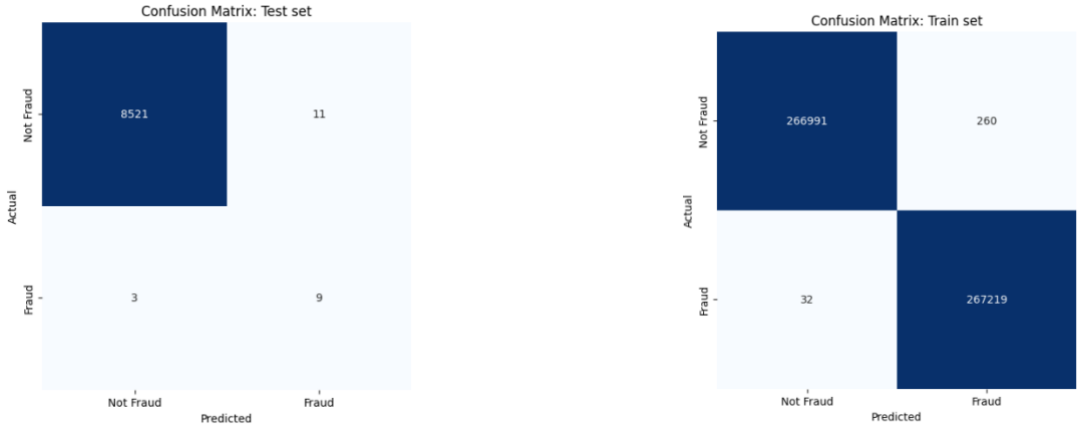


Figure 6.3 The training and validation loss over epochs



Picture: the confusion matrix for Testing data and training data

	Validation results	Testing Results
Accuracy	0.9987	0.9989
Precision	0.5555	0.64
Recall	0.7692	0.75
F1 Score	0.6451	0.6669
ROC AUC	0.9116	0.9218

Table: The measurement metrics for artificial neural network model on test data

It is clear that Artificial Neural Network is sensitive to the scale of input features, especially with those who have many layers and scaling to unit variance. So, it is important to apply standardization techniques.

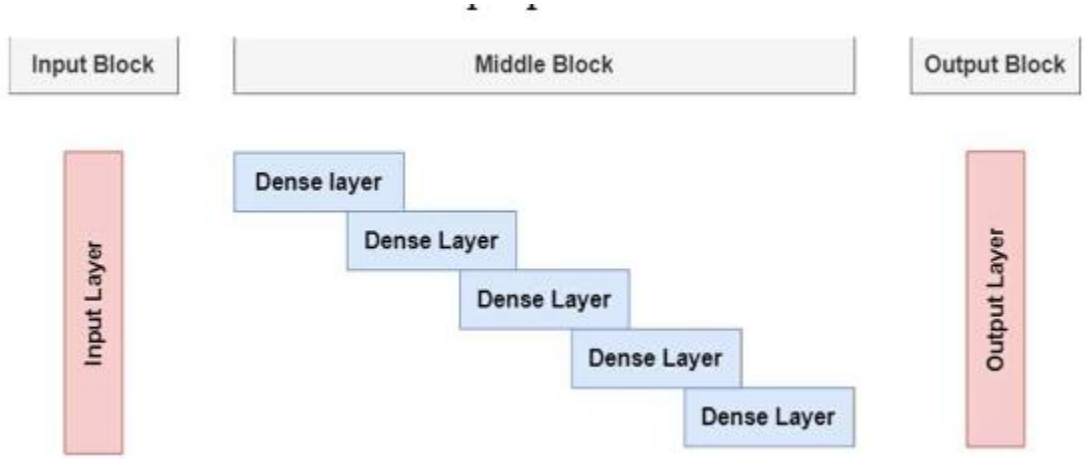
8. Deep Neural Networks (DNNs)

In this part of the project, we examine our approach of Deep Neural Network.

7.1 Theory

A deep learning neural network is an artificial neural network with several hidden layers, at least three layers, including the input and output ones. Each layer can have its own number of neurons. As mentioned, DNN is a different type of artificial neural network, so they share the same components of neurons, synapses, weights, biases, and functions. DNNs are typically feedforward networks in which data flows from the input layer to the output layer without looping back. At first, the DNN creates a map of virtual neurons and assigns random numerical values, or "weights", to connections between them. The weights and inputs are multiplied and return an output between 0 and 1. If the network did not accurately recognize a particular pattern, an algorithm would adjust

the weights. That way the algorithm can make certain parameters more influential, until it determines the correct mathematical manipulation to fully process the data.



Picture: The architecture of deep neural network classifier.

The main difference between ANN and DNN is that ANNs are relatively shallow and are used for simpler tasks, while DNNs are characterized by their depth and are employed for complex pattern recognition tasks where hierarchical features are essential. By adding more layers and density into the code, we hope that the neural network model can perform better.

7.2 Implementation

We first employ the required libraries for creating a neural network model alongside hyperparameters that are needed for the hyperparameter tuning process.

- **The size** (number of hidden layers and number of units per layer): this parameter defines how large we want the model to be, the size may depend on the difficulty level of the task, or the complexity of the input value.
- **The learning rate** decides how quickly the code adapts to and modifies itself for the input value.
- **Initial weights**: is an important consideration in the design of a neural network model. The nodes in neural networks are composed of parameters referred to as weights used to calculate a weighted sum of the inputs.

```
def create_model(learning_rate=0.01):
    model = Sequential()
    model.add(Dense(X_train.shape[1], input_dim=X_train.shape[1], activation='relu'))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))

    optimizer = SGD(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

Picture: an example of DNN model with learning rate of 0.01

We first created the Deep neural network model, for example, with a learning rate of 0.01. This parameter specifies the learning rate used by the optimizer during training. If no learning rate is provided when calling this function, it defaults to 0.01. We create a sequential model object, representing a linear stack of layers. The first layer added to the model is a dense (fully connected) layer with "X_train. shape[1]" units. "X_train.shape[1]" is the number of features in the input data. The "input_dim" parameter specifies the number of input features. The activation function used in this layer is ReLU (Rectified Linear Unit), specified by activation='relu'. The second layer added is another dense layer with 32 units, also using ReLU activation. The third and final layer is a dense layer with one unit and a sigmoid activation function. This layer is typically used for binary classification tasks, such as credit card fraud classification, as it produces output probabilities between 0 and 1.

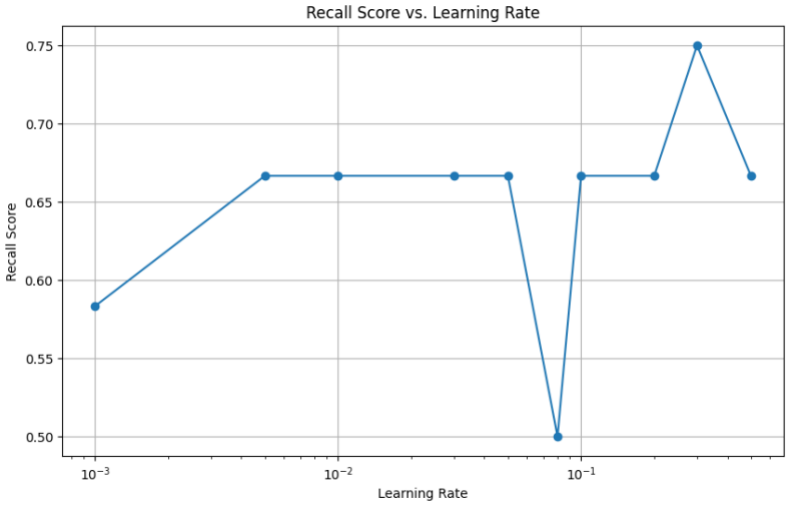
Hyperparameter Tuning

As mentioned above, we will evaluate the different values of the network size, the learning rate of the model, and the initial weights of neurons.

- The learning rate

The learning rate to govern the pace at which an algorithm updates or learns the values of a parameter estimate. A too high learning rate will make the learning jump over minima, while a too low learning rate will either take too long to converge or get stuck in an undesirable local minimum. We evaluated our DNN model with 3 different learning rates: low ones of (0.001,0.005), medium ones (0.01,0.03,0.05) and a high learning rate (0.1,0.2,0.3,0.5). Here are the results:

Learning rate	0.001	0.005	0.01	0.03	0.05	0.08	0.1	0.2	0.3	0.5
Precision	0.88	0.89	0.89	0.89	0.89	0.86	0.89	0.89	0.9	0.89



As the learning rate 0.3 has a slightly better recall score than others, we take that as our main learning rate for the DNN model. Although it is a little higher than the average learning rate, we have experimented and think that this is still a usable value.

- The network size

The network size depends a lot on the complexity of the task, a more challenging task would be required. We test the code using varied sizes of the neural network.

Size	[64,]	[32,]	[64,32]	[64,32,16]	[64,32,16,1]
Recall score	0.97	0.96	0.96	0.97	0.98

As the examination shows, the difference between varied sizes of the networks does not fluctuate too massively, as the problem is not too hard of a problem to increase a higher number of layers. We chose the number 4 layers with [64,32,16,1] as our choice.

- Initial weight

We examined on three different weights initialization techniques:

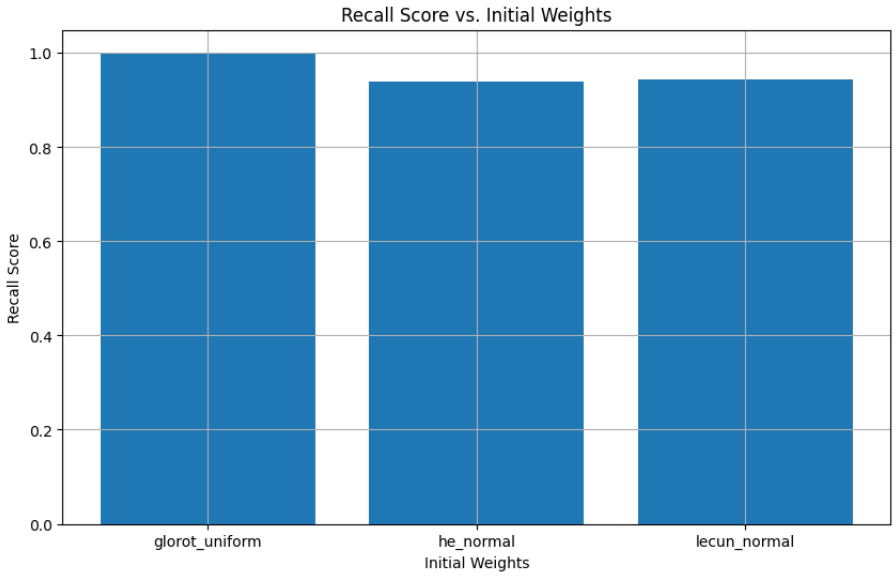
+) Glorot uniform initialization technique is the weights initialization technique that tries to make the variance of the outputs of a layer to be equal to the variance of its inputs. Naturally, this initialization depends on the layer activation function.

+) he_normal (Kaiming) initialization technique is an initialization method for neural networks that considers the non-linearity of activation functions, such as ReLU activations. A proper initialization method should avoid reducing or magnifying the magnitudes of input signals exponentially. The formula is as follows:

$$w_l \sim N(0, \frac{2}{n_l})$$

That is, a zero-centered Gaussian with standard deviation of $\sqrt{\frac{2}{n_l}}$ (variance shown in equation above). Biases are initialized at 0.

+) lecun initialization technique:

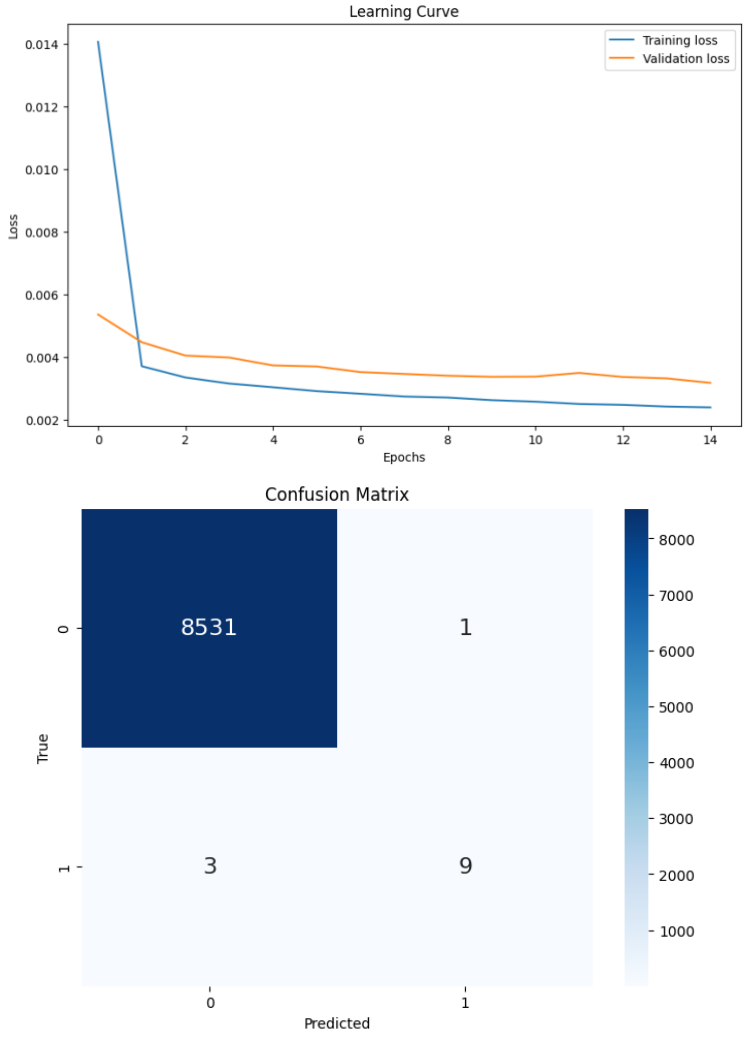


7.3 Results

Given the best parameters after the hyperparameter tuning process. We have the following results for our DNN model, testing on the dataset:

+) network size: (64,32,16,1), learning_rate = 0.0, inital weights = Glorot Uniform.

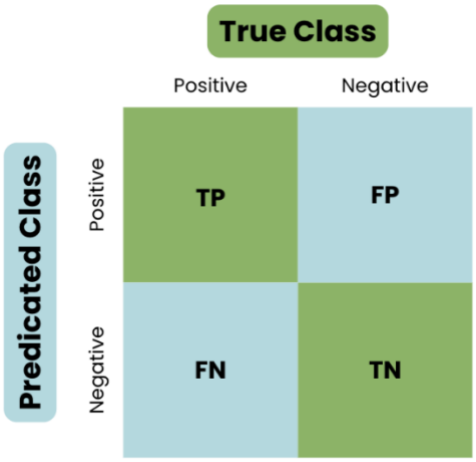
Measurements	Results
precision	0.90
recall	0.75
F1-score	0.82
Accuracy	0.96



The confusion matrix and the learning curve of DNN

Part IV: Results and Comparisons

In this part, we provide the output of all the models and compare them to find out the model that is most suitable for this problem. It is mandatory that we use base metrics of evaluation such as “Accuracy”, “Precision”, “Recall” and “F1 score”. The formulas for those metrics are as follows:



$$Accurancy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2\ Precision * Recall}{Precision + Recall}$$

TP – True Positive *TN – True Negative* *FP – False Positive* *FN – False Negative*

In this project context, we consider fraud (Class=1) as our positive class. Here are the key features of each metric:

- ***Accuracy***: This metric measures the overall correct prediction rate of the model. It shows how many predictions are correct, but it does not specify the accuracy for each class separately. For example, in our case, if the model predicts the negative class correctly most of the time but performs poorly on the positive class, the overall accuracy might still be high. This makes accuracy less suitable for our needs, as it does not reflect the model's effectiveness in detecting fraud transactions.
- ***Precision***: measures the proportion of true positive predictions out of all instances predicted as positive. This metric is crucial when the cost of false positives (incorrectly classifying a negative instance as positive) is high. For example, in medical diagnosis, high precision is important to avoid unnecessary treatments for patients who are not actually sick. In other words, precision answers the question: "Of all the instances predicted as positive, how many are actually positive?"
- ***Recall (Sensitivity)***: In our project context, recall seems to be the most suitable metric. Recall measures the proportion of true positive predictions out of all actual positive instances. High recall means a low possibility of missing a fraud transaction, which is critical for our goal. Therefore, ***recall should be prioritized*** when evaluating models for fraud detection.
- ***F1 Score:*** The F1 score is the harmonic mean of precision and recall. A high F1 score indicates that both precision and recall are high. However, in our project context, the F1 score may not be as suitable because we are willing to trade off precision for higher recall to ensure we detect as many fraud transactions as possible.

Based on these metrics, here is our detail experimental results for seven models:

Methods	Accuracy	Precision	F1-Score	Recall
LR	0.96	0.93	0.96	0.98
SVM	0.95	0.95	0.94	0.95
DT	1.0	1.0	1.0	1.0
RF	1.0	1.0	1.0	1.0
KNN	0.95	0.98	0.91	0.84
ANN	0.99	0.64	0.69	0.75
DNN	0.89	0.90	0.82	0.75

To further visualize the performance of various models, we created a box plot displaying the distribution of error scores for each model. This box plot allows us to compare the models in terms of their central tendency, variability, and presence of outliers.

Box Plot Explanation

- X-Axis (Models): The x-axis represents the different models being evaluated. Each model is labeled accordingly, allowing for easy comparison across multiple approaches. In our graph: there are eight models.

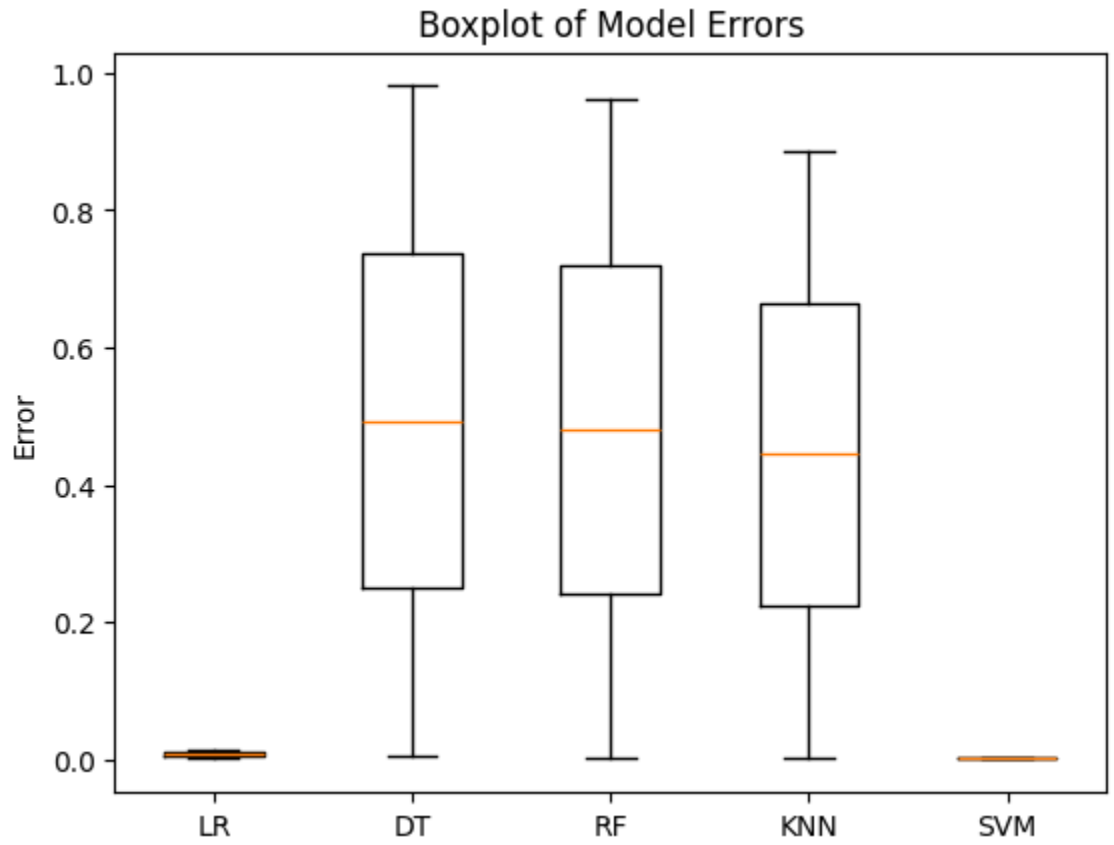
- Y-Axis (Error Scores): The y-axis represents the error scores associated with each model. Lower values indicate better performance, with scores derived from an appropriate error metric, in this case the Mean error squared.

- Boxes: Each box in the plot represents the interquartile range (IQR) of the error scores for a model, which is the range between the 25th percentile (Q1) and the 75th percentile (Q3). This shows the spread of the middle 50% of the data. The line inside each box represents the median error score, providing insight into the central tendency of the model's performance.

- Whiskers: The whiskers extend from the boxes to the smallest and largest values within 1.5 times the IQR from Q1 and Q3, respectively. These lines help identify the range within which most of the data points lie.

- Outliers: Any data points outside the whiskers are considered outliers and are plotted individually. These points highlight exceptional cases where the model's error scores were significantly higher or lower than the rest of the data.

Here is the boxplot of different models.



The box plot graph of 5 different models

We can make comparisons by examining the medians, we can identify which model has the lowest typical error score, indicating better overall performance. The spread of boxes and the presence of outliers give insights into the consistency of each model. The outliers can highlight potential issues with specific models or instances where the model performed exceptionally poorly or well. Due to different libraries used, we could not plot the graph including ANN and DNN models.

From the graph, it can be seen that Logistic Regression (LR) and the Support Vector Machine (SVM), the errors are very low and tightly clustered, indicating high consistency and low error score. The Decision Tree (DT) and Random Forest (RF) have the errors spread over a wider range, with the interquartile range (IQR) spanning from about 0.2 to 0.7. The median error stands at around 0.45, indicating a moderate performance with higher variability. The K-Nearest-Neighbors (KNN) has an IQR slightly narrower than Decision Tree and Random Forest, but the range and median are approximately similar, indicating comparable performance to Decision Tree and Random Forest.

Overall, to compare our seven ways of approaching this credit card fraud classification method, we use comparisons, and it is proven that the Decision Tree, Random Forest perform better than

the other models while Artificial Neural Network perform worse than other models. This can be explained, the Artificial Neural Network is likely to be overfitted while models such as Decision Tree and Random Forest are good at handling imbalance data and classification problem.

Future Implementation

Although our project of Credit Card Fraud Detection has produced some satisfactory results on classifying the fraudulent and legitimate transactions in a dataset, it is still quite simple and can still be modified for enhancing the model for better performance and robustness:

The project currently relies on a data set with comprehensive information about customers and their transactions. In real-life scenarios, obtaining such detailed information can be challenging due to privacy concerns and data availability. To address this, the model can be adapted to work with more limited and anonymized data, making it more applicable to real-world conditions.

Artificial Neural Network model performed very badly and needs to be rebuilt.

Despite the models' s acceptable scores, some techniques can still be applied to enhance the model's performance. With techniques such as the Bayes Search for hyperparameter tuning process, Adaptive Synthetic Sampling to handle the imbalanced dataset, there might be some improvements to our current approach to the problem.

The project is still not user-friendly. A better user interface system can be built to resolve the problem and help the project to be able to be used more effectively.

In conclusion, the capstone project was able to perform and give out satisfactory results but there are still many improvements that can be made to further improve the performance of the project.

References

1. *Machine Learning & Data Science Blueprints for Finance by Hariom Tatsat, Sahil Puri& Brad Lookabaugh*
2. *MACHINE LEARNING An Algorithmic Perspective by Stephen Marsland*
3. *Credit Card Fraud Detection using Deep Learning Techniques*. Available from: https://www.researchgate.net/publication/350829171_Credit_Card_Fraud_Detection_using_Deep_Learning_Techniques [accessed May 31 2024].
4. *K-nearest neighbors algorithm for credit card fraud detection*. https://pyfi.com/blogs/articles/k-nearest-neighbors-algorithm-for-credit-card-fraud-detection
5. *K-Nearest Neighbors (KNN) Algorithm.* https://www.geeksforgeeks.org/k-nearest-neighbours/
6. *How Logistic Regression Works: The Sigmoid Function and Maximum Likelihood* https://medium.com/@karan.kamat1406/how-logistic-regression-works-the-sigmoid-function-and-maximum-likelihood-36cf7cec1f46
7. *Logistic Regression* https://machinelearningcoban.com/2017/01/27/logisticregression/
8. *Understanding logistic regression* https://www.geeksforgeeks.org/understanding-logistic-regression/
9. *Decision Tree Classification Algorithm* https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm
10. *Gini Index* https://www.youtube.com/watch?v=u4IxOk2ijSs
11. *Decision tree algorithm* https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/
12. *Decision tree* https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/
13. *Random forest* https://medium.com/@denizgunay/random-forest-af5bde5d7e1e
14. *Bootstrap in machine learning* https://www.analyticsvidhya.com/blog/2020/02/what-is-bootstrap-sampling-in-statistics-and-machine-learning/