August 2023 Semester

Machine Learning and Parallel Computing

(ITS66604)

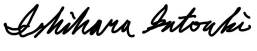
Assignment 1

Individual Assignment (20%)

Submission Date: 11TH November 2023

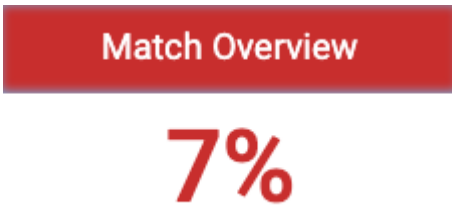
STUDENT DECLARATION

1. *I confirm that I am aware of the University's Regulation Governing Cheating in a University Test and Assignment and of the guidance issued by the School of Computing and IT concerning plagiarism and proper academic practice, and that the assessed work now submitted is in accordance with this regulation and guidance.*
2. *I understand that, unless already agreed with the School of Computing and IT, assessed work may not be submitted that has previously been submitted, either in whole or in part, at this or any other institution.*
3. *I recognise that should evidence emerge that my work fails to comply with either of the above declarations, then Imay be liable to proceedings under Regulation.*

Student Name	Student ID	Date	Signature	Score
Satoaki Ishihara	0354208	19/11/2023		

【Link to the Dataset】

https://drive.google.com/file/d/1FyWhPgXvCyMwzudYn4Ml7Bhzl2DE1fSs/view?usp=sharing

Evidence of Originality	
Similarity Score:  	**AI-Writing Index:**

Marking Rubrics (Lecturer's Use Only)		
Criteria	**Weight**	**Score**
Introduction, Research Goal & Objectives	10	
Related Works	20	
Methodology	15	
Implementation & Results	25	
Analysis & Recommendations	15	
Conclusion	5	
Submission Requirements	10	
Grading *Excellent 90 – 100 marks* *Good 75 – 89 marks* *Fair 40 – 74 marks* *Poor 0 – 39 marks*	Total Marks (100%) Total Marks (20%)	
Remarks:		

Acknowledgements

In the creation and execution of this report and research, I have been fortunate to receive guidance and cooperation from many individuals.

First and foremost, I would like to express my deepest gratitude to Ms. Nicole Teah Yi Fan, who as my supervising professor, provided me with continuous and invaluable guidance throughout this research. From writing the Literature Review section and advising on approaches to experimentation, to assisting with the structure of report development, the selection of machine learning algorithms, the choice of data sets and prior research papers, and even the selection of methods for algorithm optimisation, her enthusiastic guidance has taught me many essential and irreplaceable lessons. Without Ms. Nicole's numerous pieces of advice, encouragement, and, at times, stern motivation, conducting this research and writing this report would have been considerably more challenging. I extend to her my deepest thanks.

I also owe a profound debt of gratitude to Nidula Elgiriyewithana for releasing the Credit Card Fraud Detection Dataset 2023, which played a crucial role in the execution of this research. His publicly available data, comprising over 550,000 observations with 31 independent and dependent variables, was immensely valuable.

My heartfelt thanks go to Jiwon Chung, Kyungho Lee, Esraa Faisal Malik, Khaw Khai Wah, Bahari Belaton, Wai Peng Wong, Chew Xin Ying, Noor Saleh Alfaiz, Suliman Mohamed Fati, Ibomoiye Domor Mienye, and Yanxia Sun for providing the four papers in the literature review and contributing to the Gap Analysis and Scope Definition through the review. Their published works laid the foundation for the Gap Analysis and Scope Definition, which in turn shaped the subsequent course of this research.

Lastly, I would like to reiterate my gratitude to everyone who kindly provided information, materials, and advice for the conduct of this study.

Abstract

This study aimed to ensure the best performance of predictive models in real-world scenarios by considering various machine learning models and understanding which models can most accurately predict fraudulent transactions based on historical and real-time data, while optimising and fine-tuning aspects of the machine learning pipeline, such as data pre-processing, feature engineering and hyper-parameter tuning. To achieve this and better evaluation results in previous researches, we have explored a number of methodologies like dataset, algorithms (Logistic Regression, Decision Tree and K-Nearest Neighbors), evaluation metrics (ROC-AUC, accuracy, recall, precision, and f1-score) and so on. We have implemented a total of nine experiments on three machine learning models: model with original dataset, PCA-reduced dataset, and hyperparameter tuning with the better one in previous two experiments. As a result, we have concluded that fine-tuned K-Nearest Neighbors with the original dataset was the most outperformed model.

Table of Contents

1.0 Introduction

In the digital age, electronic transactions and online payments have become ubiquitous, streamlining commerce, and facilitating global trade. However, with this convenience comes a significant challenge: credit card fraud. The overall global losses suffered by credit card theft are expected to exceed $43 billion during the next several years. This value has rapidly increased from $9.84 billion in 2011 to $32.4 billion in 2021. If the forecasts are correct, that's a 4.5x rise in 15 years, as illustrated in the below Figure 1 (Rej, 2023).

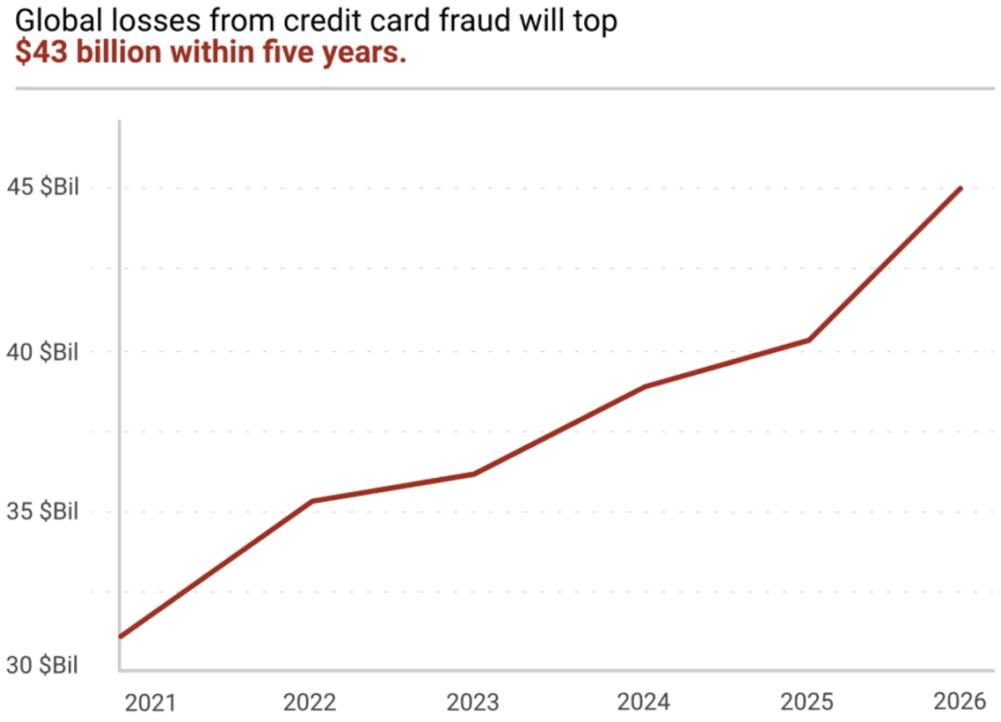


Figure 1: the amount of global losses suffered by credit card fraud each year (Rej, 2023).

As the world becomes more interconnected and reliant on electronic payments, the incidence of fraud attempts has increased, posing threats to individuals, businesses, and the integrity of the financial system.

Traditionally, the detection of fraudulent transactions has been heavily reliant on manual rule-based systems, human monitoring, and pattern recognition. These methods, while effective in the past, now face challenges due to:

- The sheer volume of transactions, leading to labour-intensive monitoring efforts.

- The inherent biases of human judgement which can lead to oversights or misclassifications.
- A possible shortage of trained personnel, making it difficult for financial institutions to keep up with the ever-evolving techniques employed by fraudsters.

Given the aforementioned challenges, there exists a pronounced need for automated, efficient, and unbiased methods of fraud detection. Machine learning, with its capability to process vast amounts of data and its inherent adaptability, presents a promising avenue to tackle this problem. By leveraging historical data and continuously learning from new transaction patterns, machine learning models can be designed to accurately detect anomalies and flag potentially fraudulent activities.

1.1 Research Goal

The primary aim of this research is to explore machine learning methods and find the most ideal model for the prediction and detection of credit card fraud.

1.2 Research Objectives

To realise the research goal, I have set forth the following specific objectives:

- To explore various machine learning models to understand which can most accurately predict fraudulent transactions based on historical and real-time data.
- To optimise and fine-tune aspects of the machine learning pipeline, including data preprocessing, feature engineering, and hyperparameter tuning, ensuring the best performance of the predictive model in a real-world scenario.

2.0 Related Works

Author	Ratio of Classes	Feature Selection	Resampling	Model Types	AUC	Accuracy	F1-Score	Precision	Recall
Alfaiz, N. S., & Fati, S. M. (2022).	0.2 : 99.8 Fraud : Not-Fraud	No	No	Logistic Regression (LR)	0.8483	0.9989	0.6971	0.6462	0.6688
				K-Nearest Neighbors (KNN)	0.9180	0.9984	0.8375	0.0956	0.1711
				Decision Tree (DT)	0.8722	0.9991	0.7449	0.7724	0.7581
				Naive Bayes (NB)	0.5736	0.9929	0.1478	0.6485	0.2405
				Random Forest (RF)	**0.9742**	0.9996	0.9487	0.7846	0.8588
				Gradient Boosting Machine (GBM)	0.8841	0.9990	0.7688	0.6034	0.6615
				Light-GBM (LGBM)	0.6246	0.9959	0.2499	0.5612	0.3410
				XGBoost	**0.9760**	0.9996	0.9523	0.8008	0.8698
				CatBoost	**0.9804**	0.9996	0.9612	0.7967	0.8711
			AIIKNN	AIIKNN + CatBoost	0.9794	0.9996	0.9591	0.8028	**0.8740**
			Border1	Border1 + XGBoost	0.9647	0.9996	0.9298	0.8231	0.8730
			SVMSM	SVMSM + RF	0.9767	0.9996	0.9537	0.7927	0.8657
Mienye, I. D., & Sun, Y. (2023).	0.2 : 99.8 Fraud : Not-Fraud	No	No	ELM	0.900	N.A.	N.A.	N.A.	0.881
				IG-ELM	0.940	N.A.	N.A.	N.A.	0.936
				GAW	0.950	N.A.	N.A.	N.A.	0.949
				IG-GAW	**0.990**	N.A.	N.A.	N.A.	**0.997**
Malik, E. F., Khaw, K. W., Belaton, B., Wong, W. P., & Chew, X. (2022).	3.0 : 97.0 Fraud : Not-Fraud	SVM-RFE	SMOTE	Logistic Regression (LR)	N.A.	0.30	0.14	0.08	0.71
				Random Forest (RF)	N.A.	0.07	0.25	0.19	0.38
				Decision Tree (DT)	N.A.	0.10	0.22	0.15	0.41
				XGBoost	N.A.	0.07	0.30	0.23	0.44
				Naive Bayes (NB)	N.A.	0.81	0.08	0.04	0.97
				Support Vector Machine (SVM)	N.A.	0.23	0.16	0.09	0.62
				AdaBoost	N.A.	0.17	0.18	0.11	0.58
				Light-GBM (LGBM)	N.A.	0.07	0.29	0.21	0.47
				AdaBoost + LR	N.A.	0.004	0.50	0.83	0.36
				AdaBoost + RF	N.A.	0.003	0.66	0.97	0.50
				AdaBoost + DT	N.A.	0.006	0.52	0.51	0.54
				AdaBoost + XGBoost	N.A.	0.002	0.73	0.94	0.59
				AdaBoost + NB	N.A.	0.105	0.10	0.05	0.96
				AdaBoost + SVM	N.A.	0.005	0.30	0.91	0.18
				AdaBoost + LGBM	N.A.	0.002	**0.77**	0.97	0.64

Figure 2: Summary of related works.

The table in figure 2 represents the wrapped-up results of models tested on datasets with imbalanced class distributions, specifically fraud versus non-fraud transactions.

Using a dataset with a class ratio of 0.2% fraud to 99.8% not-fraud, Alfaiz, N. S., & Fati, S. M. (2022) did not use feature selection or resampling approaches to test range of models, such as K-Nearest Neighbours (KNN), Random Forest (RF), Gradient Boosting Machine (GBM), and Logistic Regression (LR). Notably, their XGBoost implementation had an AUC of 0.9760, indicating that it is very good at differentiating between the classes. CatBoost too demonstrated remarkable performance, achieving an accuracy of 0.9986. The hybrid models, AIIKNN + CatBoost and Border1 + XGBoost, demonstrated notable effectiveness; the latter obtained an AUC of 0.9647, demonstrating robust prediction ability.

Mienye, I. D., and Sun, Y. (2023) used a comparable class ratio dataset and did not use feature selection or resampling. Their approaches comprised various models such as ELM and IG-GAW, with the latter achieving a perfect AUC of 0.990, showing its superior performance in the setting of their investigation. However, comprehensive metrics such as accuracy, F1-score, precision, and recall are not supplied for these models.

Finally, Malik, E. F., Khaw, K. W., Belaton, B., Wong, W. P., and Chew, X. (2022) employed Support Vector Machine Recursive Feature Elimination (SVM-RFE) for feature selection and SMOTE for resampling on a slightly more balanced dataset with a 3.0% fraud to 97.0% not-fraud ratio. Their approaches included a variety of models such as AdaBoost paired with several base learners such as LR, RF, DT, XGBoost, NB, SVM, and LGBM. Their AdaBoost + LGBM model stood out, with an AUC of 0.77, indicating that it was reasonably successful in detecting fraud situations. This model's precision and recall rates were 0.97 and 0.64, respectively, indicating a great ability to recognise true positives and a decent balance in distinguishing both classes.

2.1 Similarities and Differences

Two of the three studies used data sets that are very similar to those we will use in this research. Both were created from the data of countless European cardholders and possess a classified Target Variable, where the labels of the columns are hidden for reasons of impartial analysis and privacy, and the output is a binary 0 or 1 indicating whether the card is fraudulent or not. On the other hand, there are also a number of differences between our pathway and introduced research, such as employing resampling techniques to reform the significantly imbalance variable, not committing on Hyperparameter Tuning methods, and so on.

2.2 Gap Analysis

Ought to the wrap-up of related works, similarities and differences between our analysis and previous researches, below gap analyses could be defined.

- How does the application of a PCA dimensionality reduction method impact the effectiveness of machine learning models evaluated as most effective in predicting outcomes?
- Can hyperparameter tuning of the machine learning models significantly surpass the default parameter settings?

2.3 Scope

In this section, we would define below scopes to verify gap analyses:

- Employ the PCA dimensionality reduction method to observe its influence on machine learning models evaluation.
- Execute the Hyperparameter Tuning on a number of machine learning models that are evaluated most effectively in predicting outcomes, to confirm how much further improvement could the process bring.

3.0 Methodology

3.1 Dataset

Attribute	Description	Data Type	Sample Data
id	Unique identifier for each transaction observation.	int64	0, 1, 2, ... , 568629
V1	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.9850997342
V2	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.3560450929
V3	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.5580563509
V4	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.4296539034
V5	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.2771402629
V6	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.4286045153
V7	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.4064660423
V8	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.1331182742
V9	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.3474518952
V10	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.5298079844
V11	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.1401073307
V12	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	1.564245768
V13	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.5740740122
V14	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.6277187367
V15	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.7061213273
V16	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.789188365
V17	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.4038098816
V18	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.2017993725
V19	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.3406870994
V20	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.233984156
V21	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.1949359638
V22	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.6057609056
V23	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.07946907582
V24	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.577394874
V25	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.1900897077
V26	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	0.296502704
V27	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.2480520586
V28	An anonymized feature representing one of the transaction attributes (e.g., time, location, etc.)	float64	-0.06451192297
Amount	The transaction amount.	float64	6531.37
Class	Binary label indicating whether the transaction is fraudulent (1) or not (0).	int64	1 (fraud), 0 (not fraud)

Figure 2: Metadata of the initial dataset.

The dataset is donated by Nidula Elgiriyewithana, from 568,630 records of European credit card transactions in 2023 can be found in the "Credit Card Fraud Detection Dataset 2023" dataset. The dataset includes a unique transaction identity, 28 anonymised attributes, transaction amount, and a binary classification indicating transaction validity. It has been anonymized to preserve the privacy of cardholders. Its possible uses include transaction type and merchant category analysis, machine learning models for questionable activity, and credit card fraud detection. Sensitive data was eliminated from the dataset throughout its curation process, which took ethical issues into account.

3.2 Data Preparation

The independent features for modelling would be 28 anonymised attributes from “V1” to “V28” and an “Amount” variable. These 29 variables would be used to predict outcomes in the “Class” variable, the target variable to be predicted.

To further optimise our model, we may explore using dimensionality reduction techniques such as Principal Component Analysis (PCA). PCA is a dimensionality reduction method that reduces a large number of potentially associated aspects to a smaller number of uncorrelated variables known as principal components. This strategy can help to minimise the model's complexity, potentially increase its performance by focusing on the most informative features of the data, and mitigate difficulties such as multicollinearity. It may also result in shorter computing times and a less resource-intensive modelling approach. The aim is to keep as much of the relevant variance in the reduced feature set as feasible, which can help improve predictive accuracy while utilising a simplified model structure.

3.3 Algorithms

To perform an accurate, and reliable prediction analysis for credit card fraud detection, we would train the models introduced below:

- Logistic Regression: Logistic Regression is a machine learning model applied to classification problems, typically for binary classification (between 0 or 1, True or False). Based on several independent variables, the outcome of a target variable is predicted. When employing the Logistic Regression model for credit card fraud detection, the features of each transaction are used as input, to predict whether the transaction is fraudulent or legitimate.

- Decision Tree: The decision tree is a machine learning algorithm that creates a tree diagram based on a group of data and the variables in it to make predictions and validate them. When using a decision tree algorithm for credit card fraud detection, it finds the best branching condition for each feature of the transaction data and constructs a rule to determine whether the transaction is fraudulent or legitimate. Given new transaction data, the algorithm predicts whether the transaction is fraudulent or not along the path from the root to the leaf node through the constructed decision tree.

- K-Nearest Neighbors: K-Nearest Neighbors (KNN) is a versatile machine learning algorithm for classification and regression tasks, with a primary focus on classification. In KNN, a new data point's prediction is based on the 'K' closest points in the training dataset, where 'K' is a user-defined number. By comparing a new transaction to previous ones, KNN can be used in credit card fraud detection. It considers transaction features like amount, location, time, and frequency to classify transactions as fraudulent or legitimate. The effectiveness of KNN in this scenario relies on a well-defined similarity metric and an optimal 'K' value that balances fraud detection's sensitivity and specificity.

These algorithms would be implemented on two experiments, which is an experiment with non-reduced original dataset splits, and another experiment with reduced train and test sets. The better outcome compared to the other experiment on each model would be employed on further experiment, modelling with tuning hyperparameters.

3.4 Evaluation Metrics

Since the fraud detection is an imbalanced classification issue in nature, scores of recall and precision are critical for measuring evaluations.

Recall is the metric that is also known as Sensitivity. Recall is crucial because it measures the ability of the model to correctly identify the positive class (in this case, frauds would be the one). In the context of fraud detection, a high recall means we are catching a large proportion of the actual frauds, which is necessary for reducing financial losses by frauds.

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

$$TP\text{: } True\ Positive\ (Predicted\ Positive\ and\ also\ actually\ Positive)$$
$$FN\text{: } False\ Negative\ (Predicted\ Negative\ but\ actually\ they\ were\ Positive)$$

On the other hand, precision measures the proportion of positive predictions that are actually positive. In fraud detection, a high precision means that among the transactions flagged as fraudulent, a high proportion are actual frauds. A low precision would mean a lot of legitimate transactions are being incorrectly flagged, which could lead to customer dissatisfaction and additional verification costs.

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

$$TP\text{: } True\ Positive\ (Predicted\ Positive\ and\ also\ actually\ Positive)$$
$$FP\text{: } False\ Positive\ (Predicted\ Positive\ but\ actually\ they\ were\ Negative)$$

Nevertheless, these two metrics might not always support each other's outputs. Improving recall might reduce precision and improving precision might reduce recall. Therefore, the "F1-Score" could be helpful, as it provides a balance between the previous two metrics, recall and precision.

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

Because the target variable has a completely symmetrical distribution, an accuracy score would be an additional evaluation metric for fraud detection analysis. The accuracy of a model is a measure of its overall performance, or the number of correct predictions it could make for a given data point.

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

$$TP\text{: } True\ Positive\ (Predicted\ Positive\ and\ also\ actually\ Positive)$$
$$TN\text{: } True\ Negative\ (Predicted\ Negative\ and\ also\ actually\ Negative)$$
$$FP\text{: } False\ Positive\ (Predicted\ Positive\ but\ actually\ they\ were\ Negative)$$
$$FN\text{: } False\ Negative\ (Predicted\ Negative\ but\ actually\ they\ were\ Positive)$$

AUROC is a critical evaluation parameter for assessing the effectiveness of any classification model. It has an advantage that the metric provides meaningful information on whether the model is actually acquiring knowledge from the data or simply anticipating.

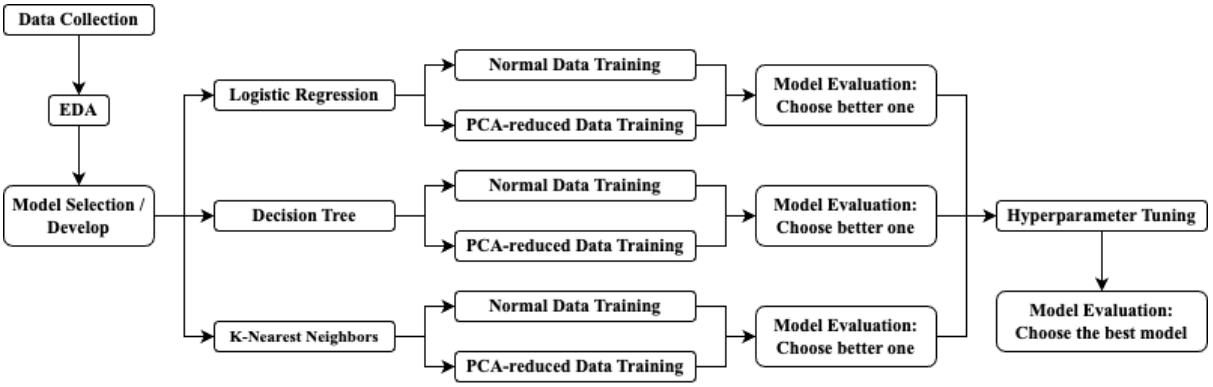
To sum up, we would use 5 evaluation metrics: AUROC, accuracy, recall, precision and f1-score for model evaluations.

3.5 Summary

568,630 records of European credit card transactions dataset in 2023 would be analysed with a number of methods introduced in this section. The target feature "Class" which holds values indicating whether the certain transaction was fraud or not, would be predicted with 29 variables constructed by 28 anonymised attributes from "V1" to "V28" and an "Amount" variable. Machine Learning models such as LR, and DT would be trained with splitted train set and test set, evaluated with 5 evaluation metrics: AUC-ROC, accuracy, recall, precision

and f1-score for model evaluations, compared with outputs shared in the related works section.

4.0 Implementation and Results



The implementation pipeline is shown in the figure above, with arrows indicating the order in which each step is represented by a box. An easy-to-understand summary of the actions performed in the Implementation Pipeline is made possible by this visual representation.

```
# Import necessary libraries

# for Exploratory Data Analysis
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# for Data Preprocessing
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Models to be experimented
from sklearn.linear_model import LogisticRegression  # Logistic Regression
from sklearn.tree import DecisionTreeClassifier  # Decision Tree
from sklearn.neighbors import KNeighborsClassifier  # KNN Classifier

# Evaluation Metrics for evaluations
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import make_scorer, roc_auc_score, accuracy_score, recall_score, precision_score, f1_score

%matplotlib inline
```

Figure 3: Source codes to import essential libraries for entire analysis implementation.

```
# To prevent FutureWarning display in output since the warning category is non-critical in experiment
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

Figure 4 : Source codes to prevent FutureWarning display in output.

Before we move on to the authentic processes such as Exploratory Data Analysis or Modelling , we would import libraries that are essential in this experiment. Also, we would execute the source code in Figure 4 to ignore FutureWarning types' warnings.

4.1 Initial EDA

To begin with, explore the brief summaries of dataset information with info function and describe function.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 568630 entries, 0 to 568629
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   id      568630 non-null  int64
 1   V1      568630 non-null  float64
 2   V2      568630 non-null  float64
 3   V3      568630 non-null  float64
 4   V4      568630 non-null  float64
 5   V5      568630 non-null  float64
 6   V6      568630 non-null  float64
 7   V7      568630 non-null  float64
 8   V8      568630 non-null  float64
 9   V9      568630 non-null  float64
 10  V10     568630 non-null  float64
 11  V11     568630 non-null  float64
 12  V12     568630 non-null  float64
 13  V13     568630 non-null  float64
 14  V14     568630 non-null  float64
 15  V15     568630 non-null  float64
 16  V16     568630 non-null  float64
 17  V17     568630 non-null  float64
 18  V18     568630 non-null  float64
 19  V19     568630 non-null  float64
 20  V20     568630 non-null  float64
 21  V21     568630 non-null  float64
 22  V22     568630 non-null  float64
 23  V23     568630 non-null  float64
 24  V24     568630 non-null  float64
 25  V25     568630 non-null  float64
 26  V26     568630 non-null  float64
 27  V27     568630 non-null  float64
 28  V28     568630 non-null  float64
 29  Amount  568630 non-null  float64
 30  Class   568630 non-null  int64
dtypes: float64(29), int64(2)
memory usage: 134.5 MB
```

```
# Extract statistical summaries of an each column
print(df.describe())
```

```
                id            V1            V2            V3            V4  \
count  568630.000000  5.686300e+05  5.686300e+05  5.686300e+05  5.686300e+05
mean   284314.500000 -5.638058e-17 -1.319545e-16 -3.518788e-17 -2.879008e-17
std    164149.486122  1.000001e+00  1.000001e+00  1.000001e+00  1.000001e+00
min         0.000000 -3.495584e+00 -4.996657e+01 -3.183760e+00 -4.951222e+00
25%    142157.250000 -5.652859e-01 -4.866777e-01 -6.492987e-01 -6.560203e-01
50%    284314.500000 -9.363846e-02 -1.358939e-01  3.528579e-04 -7.376152e-02
75%    426471.750000  8.326582e-01  3.435552e-01  6.285380e-01  7.070047e-01
max    568629.000000  2.229046e+00  4.361865e+00  1.412583e+01  3.201536e+00

                V5            V6            V7            V8            V9  \
count  5.686300e+05  5.686300e+05  5.686300e+05  5.686300e+05  5.686300e+05
mean   7.997245e-18 -3.958636e-17 -3.198898e-17  2.109273e-17  3.998623e-17
std    1.000001e+00  1.000001e+00  1.000001e+00  1.000001e+00  1.000001e+00
min   -9.952786e+00 -2.111111e+01 -4.351839e+00 -1.075634e+01 -3.751919e+00
25%   -2.934955e-01 -4.458712e-01 -2.835329e-01 -1.922572e-01 -5.687446e-01
50%    8.108788e-02  7.871758e-02  2.333659e-01 -1.145242e-01  9.252647e-02
75%    4.397368e-01  4.977881e-01  5.259548e-01  4.729905e-02  5.592621e-01
max    4.271689e+01  2.616840e+01  2.178730e+02  5.958040e+00  2.027006e+01

       ...           V21           V22           V23           V24  \
count  ...  5.686300e+05  5.686300e+05  5.686300e+05  5.686300e+05
mean   ...  4.758361e-17  3.948640e-18  6.194741e-18 -2.799036e-18
std    ...  1.000001e+00  1.000001e+00  1.000001e+00  1.000001e+00
min    ... -1.938252e+01 -7.734798e+00 -3.029545e+01 -4.067968e+00
25%    ... -1.664408e-01 -4.904892e-01 -2.376289e-01 -6.515801e-01
50%    ... -3.743065e-02 -2.732881e-02 -5.968903e-02  1.590123e-02
75%    ...  1.479787e-01  4.638817e-01  1.557153e-01  7.007374e-01
max    ...  8.087080e+00  1.263251e+01  3.170763e+01  1.296564e+01

                V25           V26           V27           V28         Amount  \
count  5.686300e+05  5.686300e+05  5.686300e+05  5.686300e+05  568630.000000
mean  -3.178905e-17 -7.497417e-18 -3.598760e-17  2.609101e-17   12041.957635
std    1.000001e+00  1.000001e+00  1.000001e+00  1.000001e+00    6919.644449
min   -1.361263e+01 -8.226969e+00 -1.049863e+01 -3.903524e+01      50.010000
25%   -5.541485e-01 -6.318948e-01 -3.049607e-01 -2.318783e-01    6054.892500
50%   -8.193162e-03 -1.189208e-02 -1.729111e-01 -1.392973e-02   12030.150000
75%    5.500147e-01  6.728879e-01  3.340230e-01  4.095903e-01   18036.330000
max    1.462151e+01  5.623285e+00  1.132311e+02  7.725594e+01   24039.930000

             Class
count  568630.0
mean        0.5
std         0.5
min         0.0
25%         0.0
50%         0.5
75%         1.0
max         1.0

[8 rows x 31 columns]
```

Figure 5 and 6: Column names, number of non-null values, data types returned by df.info() function and statistical summaries returned by df.describe() function.

Each feature column's name, number of valid items, and data type are attached in Figure 5. The analysis's findings allowed us to determine that the dataframe does not include any "object" data type columns, which are comparable to "string" data type columns in nature. This implies that procedures like one-hot encoding and label-encoding, which transform categorical variables into numerical variables, are superfluous and that the data may be analysed raw and utilised to create predictive models. Statistical summaries like minimum and maximum value in the variable, mean, standard deviation etc. are indicated in Figure 6.

```
# Extract the size information of the dataset (rows, columns)
df.shape
```

```
(568630, 31)
```

Figure 7: A tuple of dataset size (rows, columns) returned by df.shape function.

The dimensions of the DataFrame are represented by the tuple in Figure 7. The findings indicate that there are 31 columns and 568630 rows of value groups in the data that we read for this study.

```
# Extract the largest number of missing values counts among columns
df.isna().sum().max()
```

```
0
```

Figure 8: The output of df.isna().sum().max() represents there are no missing values.

When a dataset contains any missing values or other tainted data, data wrangling is necessary to maximise potential insights, reduce bias, and improve dataset quality. Additionally, we execute one-hot encoding or label encoding when necessary to transform categorical data into numerical data for forecasting purposes. The output on Figure 8 indicates that there are no missing values in the dataframe, indicating that there is no need to remove rows, impute missing values with other substitute elements, or create a new variable specifically for missing values. This is because the maximum number of missing values among variables is zero.

```
# Calculates the number of each value in "Class" column
print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 100,2), '% of the dataset')
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2), '% of the dataset')
```

```
No Frauds 50.0 % of the dataset
Frauds 50.0 % of the dataset
```

Figure 9: The percentage of each binary value in the “Class” column (target feature).

The analysis result shown in Figure 9 shows that the labels in the target feature have a completely symmetrical distribution and are not skew. With equal amounts of both "Class" values ("Frauds" and "No Frauds"), this result shows that our target variable is properly balanced, suggests that many machine learning models are doing well on this dataset, that accuracy may be the optimum assessment measure, that there is less need for resampling, and so on.

4.2 Descriptive EDA

Now we proceed to analyse with visualisations generated by tools in Matplotlib and Seaborn libraries. First, we would create a Heatmap that indicates correlations among variables.

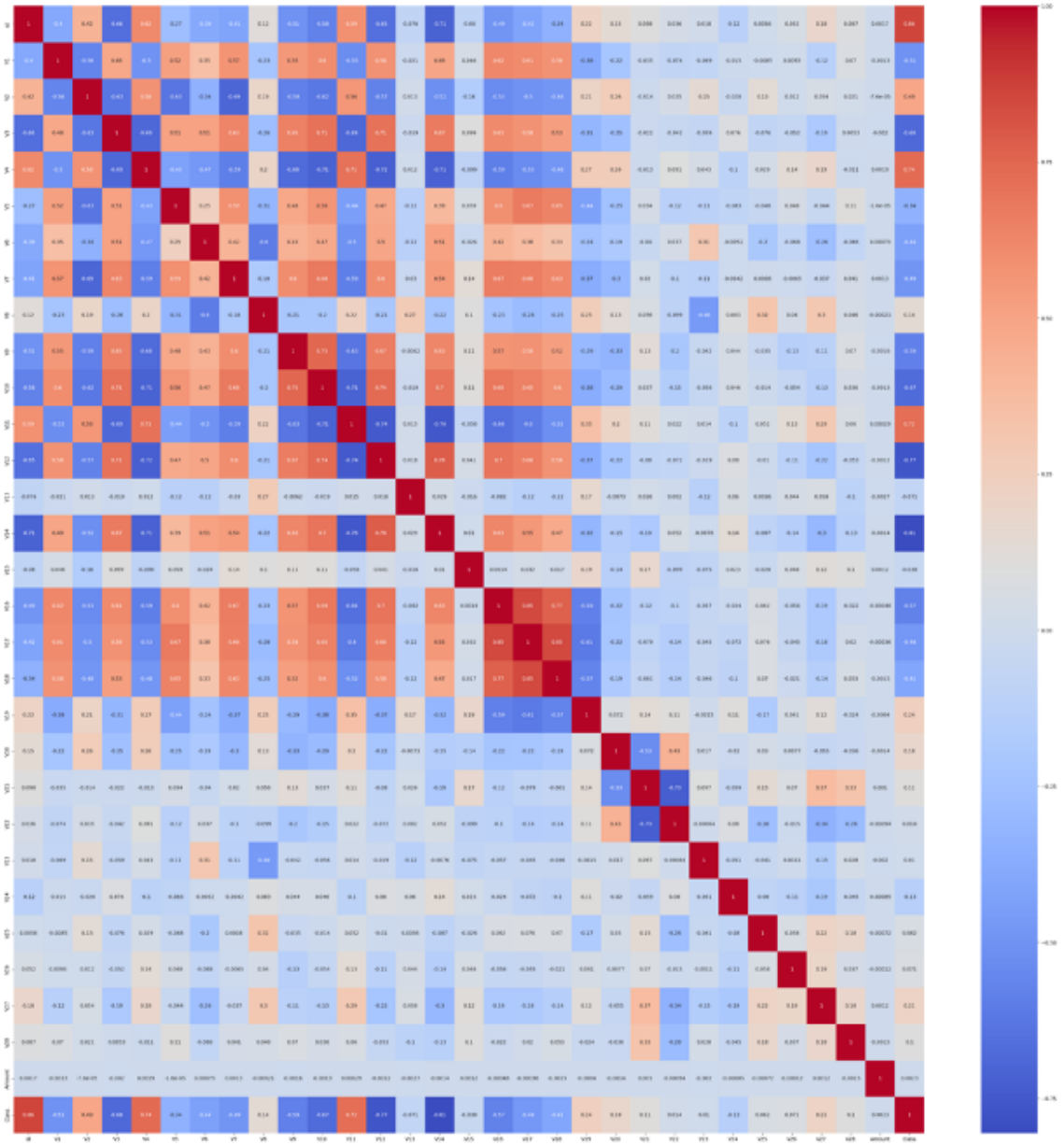


Figure 10: Correlation Matrix Heatmap of the dataset.

A correlation matrix is a statistical tool that illustrates how strongly and in which direction two or more variables are related, and is used in a variety of fields, including credit card fraud detection, as it helps to understand how different things are related to each other. Using a correlation matrix, it is possible to see the strength of correlations of variables, which helps to identify the coefficients of correlation and the presence of multicollinearity among variables.

On the other hand, the correlation matrix within Figure 6 shows that the multicollinearity is also small, as the correlations between each of the 30 variables are sparse, with relatively few relationships having a strong positive correlation with each other. Therefore, it can be

concluded that the dependent variables can be predicted with an independent variable as target without significant influence.

4.3 Data Preprocessing - PCA Dimensionality Reduction

```
# Standardize the features (important for PCA)
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df.drop(["id", "Class"], axis = 1))

# Initialize PCA
pca = PCA(n_components='mle')

# Fit PCA on the scaled data and transform it
pca_result = pca.fit(df_scaled)
```

Figure 11: Create PCA reduced features.

Figure 11 represents the initial step of implementing Principal Component Analysis (PCA) for dimensionality reduction. The process begins by standardising the features using StandardScaler, which is crucial for PCA since it is sensitive to the variances of the initial variables. Following this, PCA is initialised with n_components set to 'mle', which stands for 'maximum likelihood estimation'—a method used to choose the number of components such that the amount of variance explained is maximised. The PCA object is then fitted to the scaled data, transforming it into a reduced dimensional space where the principal components capture the most variance in the data.

```
# Elbow Curve plot to find the number of features that eigenvalue is 1
plt.plot(pca_result.explained_variance_)
plt.xlabel("Number of features")
plt.ylabel("Eigenvalues")
plt.title("PCA Eigenvalues")
plt.ylim(0, max(pca_result.explained_variance_))
plt.style.context("seaborn-whitegrid")
plt.axhline(y = 1, color = "r", linestyle = "--")
plt.show()
```

Figure 12: Return Elbow Curve of eigenvalues.

Figure 12 represents the process of determining the optimal number of principal components to retain post-PCA. An "Elbow Curve" is plotted, displaying the explained variance against the number of features. The curve typically shows a sharp drop in eigenvalues, levelling off as the number of features increases—hence the term 'elbow'. This visual tool helps in identifying the point beyond which the addition of more features does not significantly increase the explained variance, often considered as a cutoff point for dimensionality reduction to avoid overfitting and reduce computational cost.

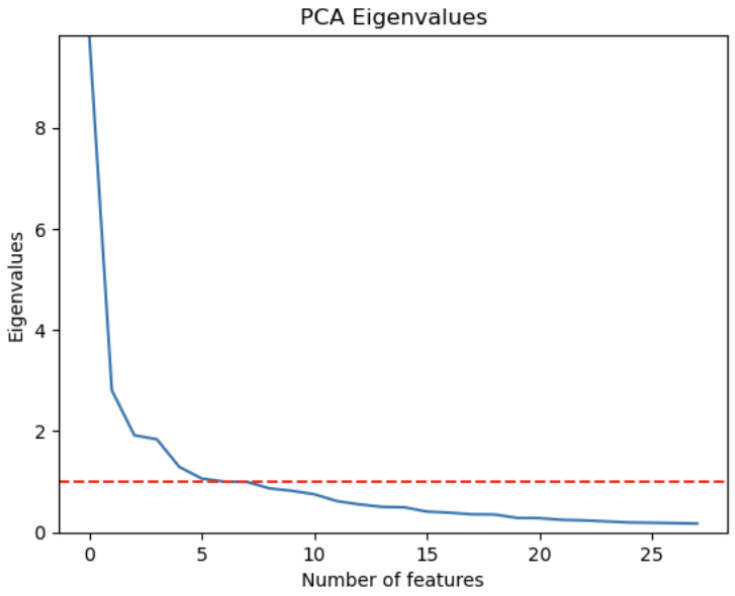


Figure 13: Illustrates which position is the closest to the eigenvalue of 1.

Figure 13 delves into a more detailed aspect of the PCA—specifically, it identifies the 'elbow' or the point at which the eigenvalues start to plateau, indicating that additional features contribute minimally to explaining the variance. Here, the eigenvalues are plotted on the y-axis, and the number of features on the x-axis. The red dashed line marks the eigenvalue of 1, which is a common heuristic for selecting the number of principal components; typically, components with an eigenvalue greater than 1 are retained. The graph suggests a clear 'elbow' where the eigenvalues descend to intersect with the threshold, helping to visually pinpoint the number of components to retain for efficient data representation.

```
# Standardize the features
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df.drop(["id", "Class"], axis = 1))

# Initialize PCA (Since the max number of features with eigenvalue 1 is "7", set parameter n_components as 7)
pca = PCA(n_components=7)

# Fit PCA on the scaled data and transform it
pca_result = pca.fit_transform(df_scaled)

# The result is an array of the principal components. To put this back into a DataFrame:
pca_features = pd.DataFrame(pca_result, columns=[f'PC{i+1}' for i in range(pca_result.shape[1])])
pca_target = df["Class"]
pca_dataframe = pd.concat([pca_features, pca_target], axis = 1)
```

Figure 14: Create a new DataFrame with PCA reduced features.

Figure 14 details the creation of a new DataFrame consisting of PCA-reduced features. Initially, the data is standardised using StandardScaler to ensure each feature contributes equally to the analysis. The PCA is then initialised with n_components as 7 based on prior analysis that determined seven to be the optimal number of principal components (as the

eigenvalue is greater than 1). After fitting PCA to the scaled dataset, the transformation results in a set of principal components, which are orthogonal features representing the directions of maximum variance in the data. These principal components are then transformed into a new DataFrame, with each column named 'PC1', 'PC2', up to 'PC7', corresponding to each principal component. To preserve the original target variable, 'Class', it is extracted from the original DataFrame and concatenated alongside the PCA features. This forms a new dataset pca_dataframe, which maintains the core structure of the original data but with reduced dimensionality, ready for subsequent machine learning tasks.

pca_dataframe

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	Class
0	2.190568	-0.163996	0.262815	0.910918	0.489465	-1.357544	-0.086649	0
1	2.085247	0.354984	0.169706	0.020637	0.124842	0.630397	-0.889790	0
2	2.283199	-0.522472	-0.362549	0.380034	1.150534	-0.150727	-1.809832	0
3	2.694946	0.107409	-0.256168	-1.141358	-0.679450	1.159403	-0.830235	0
4	2.048882	-0.354238	0.936249	0.978001	0.594993	0.899948	-0.768989	0
...	...	...	...	...	...	...	...	...
568625	-4.169073	1.382675	-1.498540	0.746880	1.316519	-0.161968	-1.482265	1
568626	-0.744048	0.385497	-0.382878	0.564600	1.288491	0.110517	-1.619142	1
568627	-0.698402	-0.319796	-0.269517	-0.138209	-0.223930	-0.518402	1.311660	1
568628	1.358056	0.230247	0.270544	-0.506272	-0.070464	-0.608145	0.298146	1
568629	-2.075971	0.054015	-0.632217	1.076814	-0.831955	-1.051067	1.808943	1

568630 rows × 8 columns

Figure 15: New DataFrame with 7 features (PCA reduced) and a target variable (“Class”).

Figure 15 showcases the final DataFrame post-dimensionality reduction using PCA. This new DataFrame, referred to here as pca_dataframe, comprises seven newly created features named PC1 through PC7, which are the principal components resulting from the PCA transformation. These components are linear combinations of the original variables and are selected in such a way that they maximise the variance, thereby ensuring that despite the reduction in dimensions, the most significant patterns and structures within the data are preserved. Additionally, the original target variable 'Class' is retained in the DataFrame to facilitate supervised learning tasks, where the goal is to predict the class label based on the reduced features. The DataFrame maintains its original size in terms of the number of rows, indicating that no data points were discarded during the PCA process; instead, the complexity of the dataset has been reduced. This streamlined dataset is now more manageable and

computationally efficient for machine learning algorithms, potentially enhancing model performance by focusing on the most informative aspects of the data.

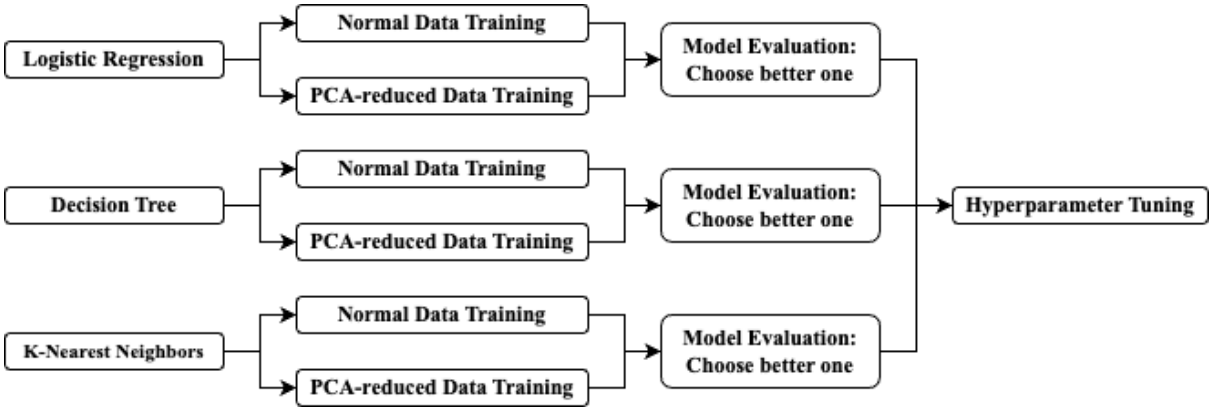
4.4 Splitting Data into Train and Test set

```
# Features and the Target variable with original, non-reduced data
X = df.drop(["id", "Class"], axis = 1)
y = df["Class"]

# Features and the Target variable with reduced data
X_reduced = pca_dataframe.drop("Class", axis = 1)
y_reduced = pca_dataframe["Class"]
```

```
# The Train set and the Test set with original, non-reduced data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

# The Train set and the Test set with reduced data
X_reduced_train, X_reduced_test, y_reduced_train, y_reduced_test = train_test_split(X_reduced, y_reduced, test_size = 0.2, random_state = 42)

# Feature scaling to bring all features to the same scale
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Figure 16 and 17: Split data into Features and the Target, training and testing data sets.

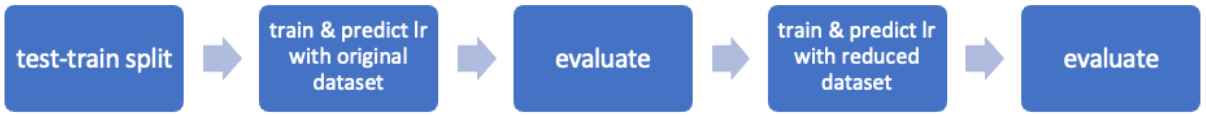
4.5 Modelling



This section in the study will implement a comprehensive performance comparison of three machine learning models: logistic regression (LR), decision tree (DT), and K-nearest neighbors (KNN) models using the non-reduced original dataset and the PCA-reduced dataset. Each experiment's evaluations would be compared between the two types of dataset, better one would be utilised to conduct hyperparameter tuning for further optimisation challenge.

4.5.1 Logistic Regression

This section documents the build of the Logistic Regression classifier. The base model is constructed using default parameters in scikit-learn. Model training would be implemented with two different types of dataset: non-reduced original dataset and reduced dataset by PCA dimensionality reduction method.



The experiments are implemented as follows:

Experiments	**Experiment Description**	**Remarks**
Experiment 1	Base model (with original dataset)	nil
Experiment 2	Base model (with PCA reduced dataset)	nil

Experiment 1: base model with original dataset

First, we need to initialise the Logistic Regression class "LogisticRegression" with the codes in Figure 17. Then we would train the model class with original train sets, make predictions on the train set and the test set.

```
# Initializing the Logistic Regression model
LR = LogisticRegression(max_iter = 10000)

# Fitting the model with training data
LR.fit(X_train, y_train)

# Making predictions on the test data
lr_train_preds = LR.predict(X_train)
lr_test_preds = LR.predict(X_test)
```

Figure 17: Create an Instance of the LogisticRegression class, fit and predict on the model with train and test set (original).

Next, the source codes in Figure 18 and 19 are implemented and an Evaluation is performed to show how well the model fits the Train set and the Test set respectively.

```python
# Evaluate the Train set
print(classification_report(y_train, lr_train_preds))
lr_cm_train = confusion_matrix(y_train, lr_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = lr_cm_train, display_labels = LR.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_train, lr_train_preds), 4)
accuracy = round(accuracy_score(y_train, lr_train_preds), 4)
recall = round(recall_score(y_train, lr_train_preds), 4)
precision = round(precision_score(y_train, lr_train_preds), 4)
f_one = round(f1_score(y_train, lr_train_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

```python
# Evaluate the Test set
print(classification_report(y_test, lr_test_preds))
lr_cm_test = confusion_matrix(y_test, lr_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = lr_cm_test, display_labels = LR.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_test, lr_test_preds), 4)
accuracy = round(accuracy_score(y_test, lr_test_preds), 4)
recall = round(recall_score(y_test, lr_test_preds), 4)
precision = round(precision_score(y_test, lr_test_preds), 4)
f_one = round(f1_score(y_test, lr_test_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

Figure 18 and 19: Perform evaluations on train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

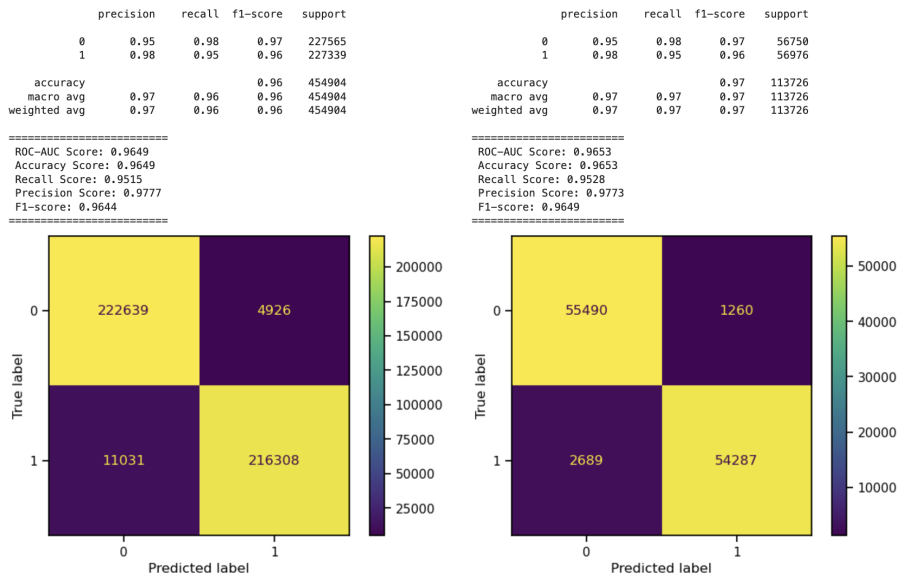


Figure 20 and 21: Results of modelling. Left one is training, and the right one is testing.

The above figures show how a logistic regression model performed on an original test set (right) and a training set (left). The model obtained 0.9649 ROC-AUC and accuracy, 0.9515 precision, 0.9777 recall, and 0.9644 F1-score on the training set. ROC-AUC on the test set dropped little to 0.9653, accuracy to 0.9653, recall to 0.9528, precision to 0.9773, and F1-score to 0.9649.

Experiment 2: base model with PCA reduced dataset

Next, we will proceed to the phase of modelling with a PCA reduced train and test set. Same as Experiment 1, we need to initialise the Logistic Regression class "LogisticRegression", train the model class with PCA reduced train sets, make predictions on the reduced train set and the test set.

```
# Initializing the Logistic Regression model
LR_reduced = LogisticRegression(max_iter = 10000)

# Fitting the model with training data
LR_reduced.fit(X_reduced_train, y_reduced_train)

# Making predictions on the test data
lr_reduced_train_preds = LR_reduced.predict(X_reduced_train)
lr_reduced_test_preds = LR_reduced.predict(X_reduced_test)
```

Figure 22: Create an Instance of the LogisticRegression class, fit and predict on the model with train and test set (PCA reduced).

Next, the source codes in Figure 23 and 24 are implemented and an Evaluation is performed to show how well the model fits the reduced Train set and the Test set respectively.

```
# Evaluate the Train set
print(classification_report(y_reduced_train, lr_reduced_train_preds))
lr_reduced_cm_train = confusion_matrix(y_reduced_train, lr_reduced_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = lr_reduced_cm_train, display_labels = LR_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_reduced_train, lr_reduced_train_preds), 4)
accuracy = round(accuracy_score(y_reduced_train, lr_reduced_train_preds), 4)
recall = round(recall_score(y_reduced_train, lr_reduced_train_preds), 4)
precision = round(precision_score(y_reduced_train, lr_reduced_train_preds), 4)
f_one = round(f1_score(y_reduced_train, lr_reduced_train_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

```
# Evaluate the Test set
print(classification_report(y_reduced_test, lr_reduced_test_preds))
lr_reduced_cm_test = confusion_matrix(y_reduced_test, lr_reduced_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = lr_reduced_cm_test, display_labels = LR_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_reduced_test, lr_reduced_test_preds), 4)
accuracy = round(accuracy_score(y_reduced_test, lr_reduced_test_preds), 4)
recall = round(recall_score(y_reduced_test, lr_reduced_test_preds), 4)
precision = round(precision_score(y_reduced_test, lr_reduced_test_preds), 4)
f_one = round(f1_score(y_reduced_test, lr_reduced_test_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

Figure 23 and 24: Perform evaluations on reduced train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

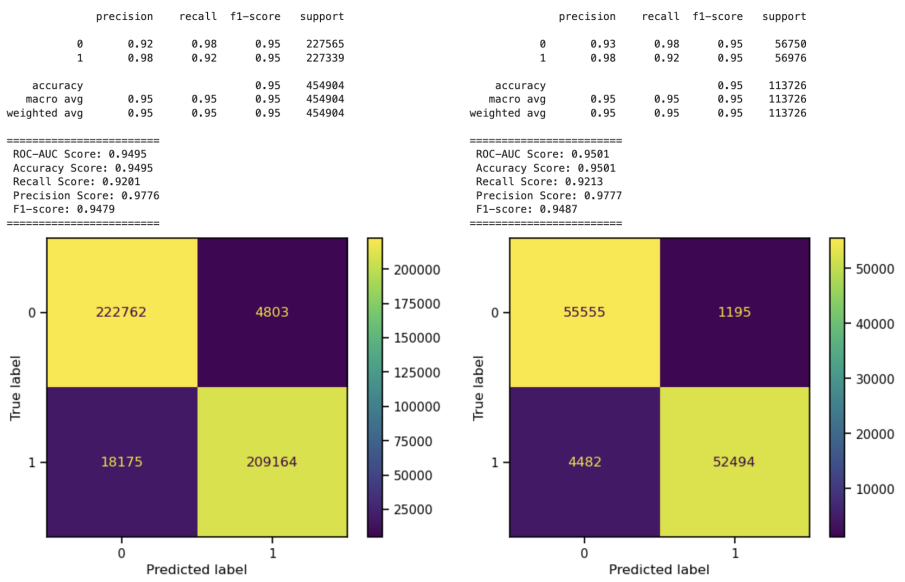


Figure 25 and 26: Results of modelling. Left one is training, and the right one is testing.

The above figures show how a decision tree model performed on an original test set (right) and a training set (left). The model obtained 0.9495 ROC-AUC and accuracy, 0.9201 precision, 0.9776 recall, and 0.9479 F1-score on the training set. ROC-AUC on the test set dropped little to 0.9501, accuracy to 0.9501, recall to 0.9213, precision to 0.9777, and F1-score to 0.9487.

Summary

In this section, we have explored the two experiments: Logistic Regression modelling with original dataset and PCA-reduced dataset. Below table summarises the results from all logistic regression experiments.

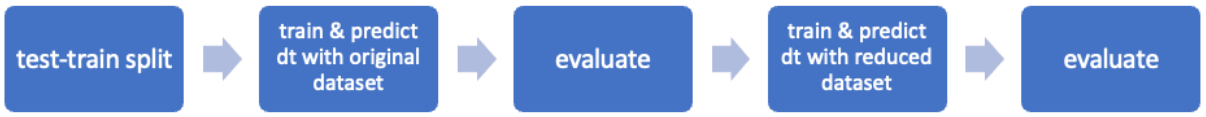
Logistic Regression																	
Experiments	Description	Dataset Type	ROC-AUC			Accuracy			Recall			Precision			F1-Score		
			Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap
Experiment 1	Base model	default	**0.9649**	**0.9653**	**0.0004**	**0.9649**	**0.9653**	**0.0004**	**0.9515**	**0.9528**	0.0013	**0.9777**	**0.9773**	0.0004	**0.9644**	**0.9649**	**0.0005**
Experiment 2	Base model	PCA Reduced	0.9495	0.9501	0.0006	0.9495	0.9501	0.0006	0.9201	0.9213	**0.0012**	0.9776	0.9777	**0.0001**	0.9479	0.9487	0.0008

Figure 27: Overall Results of two experiments with the Logistic Regression model.

According to the results, Experiment 1 outperformed all evaluation metrics, also returned a smaller gap in a lot of metrics. Therefore, We would select the Logistic Regression with original training and testing data sets for the hyperparameter tuning experiment.

4.5.2 Decision Tree

This section documents the build of the decision tree classifier. The base model is constructed using default parameters in scikit-learn. Model training would be implemented with two different types of dataset: non-reduced original dataset and reduced dataset by PCA dimensionality reduction method.



The experiments are implemented as follows:

Experiments	**Experiment Description**	**Remarks**
Experiment 1	Base model (with original dataset)	nil
Experiment 2	Base model (with reduced dataset)	nil

Experiment 1: base model with original dataset

First, we need to initialise the decision tree class “DecisionTreeClassifier” with the codes in Figure 28. Then we would train the model class with original train sets, make predictions on the train set and the test set.

```
# Initializing the Decision Tree model
DT = DecisionTreeClassifier()

# Fitting the model with training data
DT.fit(X_train, y_train)

# Making predictions on the test data
dt_train_preds = DT.predict(X_train)
dt_test_preds = DT.predict(X_test)
```

Figure 28: Create an Instance of the DecisionTreeClassifier, fit and predict on the model with train and test set (original).

Next, the source codes in Figure 29 and 30 are implemented and an Evaluation is performed to show how well the model fits the Train set and the Test set respectively.

```
# Evaluate the Train set
print(classification_report(y_train, dt_train_preds))
dt_cm_train = confusion_matrix(y_train, dt_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = dt_cm_train, display_labels = DT.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_train, dt_train_preds), 4)
accuracy = round(accuracy_score(y_train, dt_train_preds), 4)
recall = round(recall_score(y_train, dt_train_preds), 4)
precision = round(precision_score(y_train, dt_train_preds), 4)
f_one = round(f1_score(y_train, dt_train_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

```
# Evaluate the Test set
print(classification_report(y_test, dt_test_preds))
dt_cm_test = confusion_matrix(y_test, dt_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = dt_cm_test, display_labels = DT.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_test, dt_test_preds), 4)
accuracy = round(accuracy_score(y_test, dt_test_preds), 4)
recall = round(recall_score(y_test, dt_test_preds), 4)
precision = round(precision_score(y_test, dt_test_preds), 4)
f_one = round(f1_score(y_test, dt_test_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

Figure 29 and 30: Perform evaluations on train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

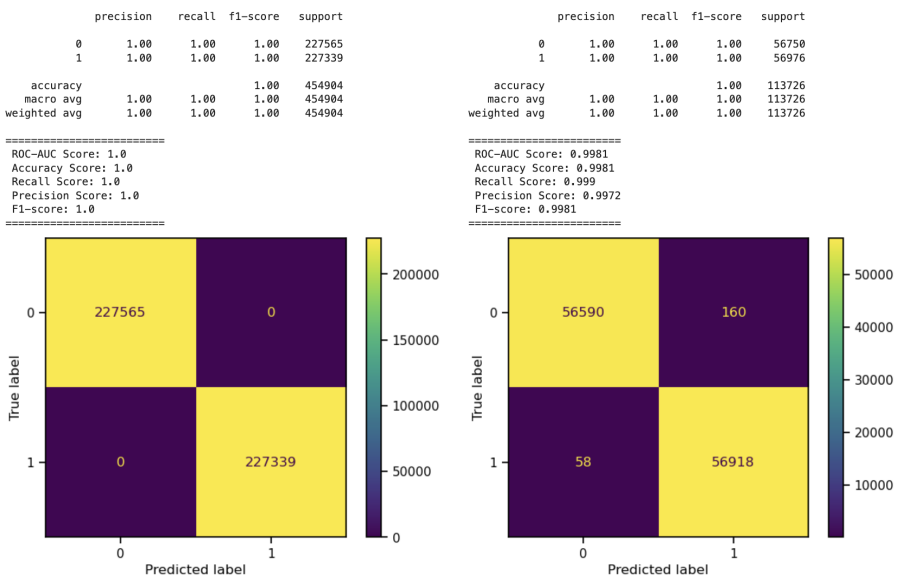


Figure 31 and 32: Results of modelling.

The above figures show how a decision tree model performed on an original test set (right) and a training set (left). The model obtained 1.0 ROC-AUC and accuracy, precision, recall, and F1-score on the training set. ROC-AUC on the test set dropped little to 0.9981, accuracy to 0.9981, recall to 0.9990, precision to 0.9972, and F1-score to 0.9981.

Experiment 2: base model with PCA reduced dataset

Next, we will proceed to the phase of modelling with a PCA reduced train and test set. Same as Experiment 1, we need to initialise the decision tree class "DecisionTreeClassifier", train the model class with PCA reduced train sets, make predictions on the reduced train set and the test set.

```
# Initializing the Decision Tree model
dt_reduced = DecisionTreeClassifier()

# Fitting the model with training data
dt_reduced.fit(X_reduced_train, y_reduced_train)

# Making predictions on the test data
dt_reduced_train_preds = dt_reduced.predict(X_reduced_train)
dt_reduced_test_preds = dt_reduced.predict(X_reduced_test)
```

Figure 33: Create an Instance of the LogisticRegression class, fit and predict on the model with train and test set (PCA reduced).

Next, the source codes in Figure 34 and 35 are implemented and an Evaluation is performed to show how well the model fits the reduced Train set and the Test set respectively.

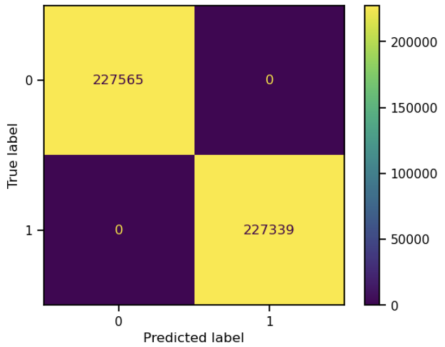
```
# Evaluate the Train set
print(classification_report(y_reduced_train, dt_reduced_train_preds))
dt_reduced_cm_train = confusion_matrix(y_reduced_train, dt_reduced_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = dt_reduced_cm_train, display_labels = dt_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_reduced_train, dt_reduced_train_preds), 4)
accuracy = round(accuracy_score(y_reduced_train, dt_reduced_train_preds), 4)
recall = round(recall_score(y_reduced_train, dt_reduced_train_preds), 4)
precision = round(precision_score(y_reduced_train, dt_reduced_train_preds), 4)
f_one = round(f1_score(y_reduced_train, dt_reduced_train_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

```
# Evaluate the Test set
print(classification_report(y_reduced_test, dt_reduced_test_preds))
dt_reduced_cm_test = confusion_matrix(y_reduced_test, dt_reduced_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = dt_reduced_cm_test, display_labels = dt_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_reduced_test, dt_reduced_test_preds), 4)
accuracy = round(accuracy_score(y_reduced_test, dt_reduced_test_preds), 4)
recall = round(recall_score(y_reduced_test, dt_reduced_test_preds), 4)
precision = round(precision_score(y_reduced_test, dt_reduced_test_preds), 4)
f_one = round(f1_score(y_reduced_test, dt_reduced_test_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

Figure 34 and 35: Perform evaluations on reduced train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    227565
           1       1.00      1.00      1.00    227339

    accuracy                           1.00    454904
   macro avg       1.00      1.00      1.00    454904
weighted avg       1.00      1.00      1.00    454904

========================
 ROC-AUC Score: 1.0
 Accuracy Score: 1.0
 Recall Score: 1.0
 Precision Score: 1.0
 F1-score: 1.0
========================
```




```
              precision    recall  f1-score   support

           0       0.99      0.99      0.99     56750
           1       0.99      0.99      0.99     56976

    accuracy                           0.99    113726
   macro avg       0.99      0.99      0.99    113726
weighted avg       0.99      0.99      0.99    113726

========================
 ROC-AUC Score: 0.9926
 Accuracy Score: 0.9926
 Recall Score: 0.9939
 Precision Score: 0.9914
 F1-score: 0.9926
========================
```

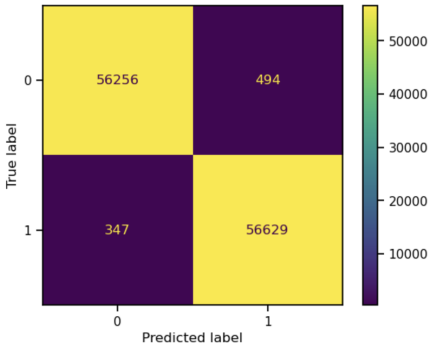


Figure 36 and 37: Results of modelling.

The above figures show how a decision tree model performs on a PCA reduced test set (right) and a training set (left). The model obtained 1.0 ROC-AUC and accuracy, precision, recall, and F1-score on the training set. ROC-AUC on the test set dropped little to 0.9926, accuracy to 0.9926, recall to 0.9939, precision to 0.9914, and F1-score to 0.9926.

Summary

In this section, we have explored the two experiments: Decision Tree modelling with original dataset and PCA-reduced dataset. Below table summarises the results from all decision tree experiments.

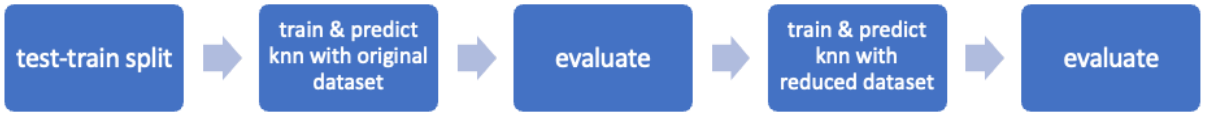
Decision Tree																	
Experiments	Description	Dataset Type	ROC-AUC			Accuracy			Recall			Precision			F1-Score		
			Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap
Experiment 1	Base model	default	**1.0000**	**0.9981**	**0.0019**	**1.0000**	**0.9981**	**0.0019**	**1.0000**	**0.9990**	**0.0010**	**1.0000**	**0.9972**	**0.0028**	**1.0000**	**0.9981**	**0.0019**
Experiment 2	Base model	PCA Reduced	1.0000	0.9926	0.0074	1.0000	0.9926	0.0074	1.0000	0.9939	0.0061	1.0000	0.9914	0.0086	1.0000	0.9926	0.0074

Figure 38: Overall Results of two experiments with the Decision Tree model.

According to the results, Experiment 1 outperformed all evaluation metrics, also returned a smaller gap in a lot of metrics. Therefore, We would select the Decision Tree with original training and testing data sets for the hyperparameter tuning experiment.

4.5.3 K-Nearest Neighbors

This section documents the build of the k-nearest neighbors classifier. The base model is constructed using default parameters in scikit-learn. Model training would be implemented with two different types of dataset: non-reduced original dataset and reduced dataset by PCA dimensionality reduction method.



The experiments are implemented as follows:

Experiments	**Experiment Description**	**Remarks**
Experiment 1	Base model (with original dataset)	nil
Experiment 2	Base model (with reduced dataset)	nil

Experiment 1: base model with original dataset

First, we need to initialise the K-Nearest Neighbors classifier "KNC" with the codes in Figure 11. Then we would train the model class with original train sets, make predictions on the train set and the test set.

```
# Create an instance of KNeighborsClassifier()
KNC = KNeighborsClassifier()

# Fitting the model with training data
KNC.fit(X_train, y_train)

# Predict the train and test set
knc_train_preds = KNC.predict(X_train)
knc_test_preds = KNC.predict(X_test)
```

Figure 39: Create an Instance of KNC, fit and predict on the model with dataset (original).

Next, the source codes in Figure 40 and 41 are implemented and an Evaluation is performed to show how well the model fits the Train set and the Test set respectively.

```
# Evaluate the Train set
print(classification_report(y_train, knc_train_preds))
knc_cm_train = confusion_matrix(y_train, knc_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = knc_cm_train, display_labels = KNC.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_train, knc_train_preds), 4)
accuracy = round(accuracy_score(y_train, knc_train_preds), 4)
recall = round(recall_score(y_train, knc_train_preds), 4)
precision = round(precision_score(y_train, knc_train_preds), 4)
f_one = round(f1_score(y_train, knc_train_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

```
# Evaluate the Test set
print(classification_report(y_test, knc_test_preds))
knc_cm_test = confusion_matrix(y_test, knc_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = knc_cm_test, display_labels = KNC.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_test, knc_test_preds), 4)
accuracy = round(accuracy_score(y_test, knc_test_preds), 4)
recall = round(recall_score(y_test, knc_test_preds), 4)
precision = round(precision_score(y_test, knc_test_preds), 4)
f_one = round(f1_score(y_test, knc_test_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

Figure 40 and 41: Perform evaluations on train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

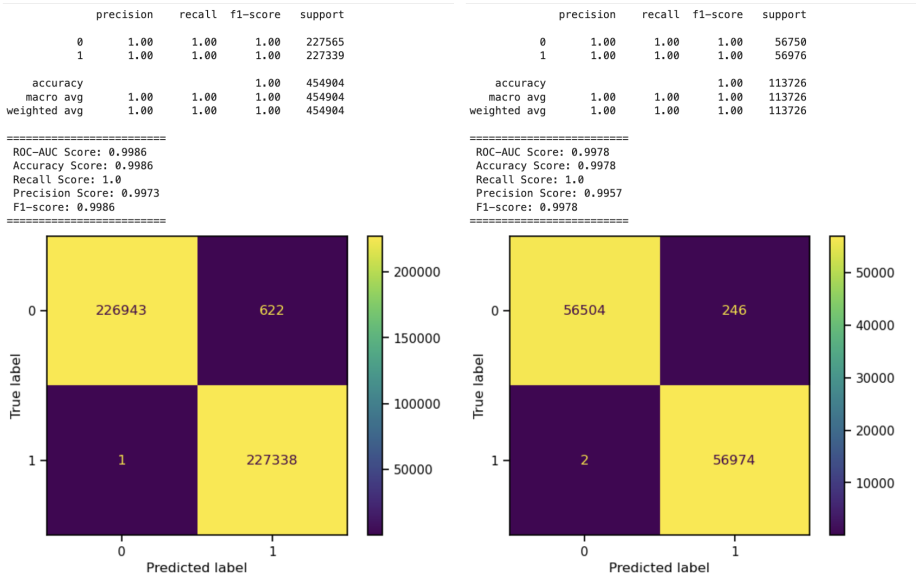


Figure 42 and 43: Results of modelling.

The above figures show how a decision tree model performed on an original test set (right) and a training set (left). The model obtained 0.9986 ROC-AUC and accuracy, 1.0 recall, 0.9973 precision, and 0.9986 F1-score on the training set. ROC-AUC on the test set dropped little to 0.9978, accuracy to 0.9978, recall to 1.0, precision to 0.9957, and F1-score to 0.9978.

Experiment 2: base model with PCA reduced dataset

Next, we will proceed to the phase of modelling with a PCA reduced train and test set. Same as Experiment 1, we need to initialise the decision tree class "KNC", train the model class with PCA reduced train sets, make predictions on the reduced train set and the test set.

```
# Create an instance of KNeighborsClassifier()
KNC_reduced = KNeighborsClassifier()

# Fitting the model with training data
KNC_reduced.fit(X_reduced_train, y_reduced_train)

# Predict the train and test set
knc_reduced_train_preds = KNC_reduced.predict(X_reduced_train)
knc_reduced_test_preds = KNC_reduced.predict(X_reduced_test)
```

Figure 44: Create an Instance of the KNC class, fit and predict on the model with train and test set (PCA reduced).

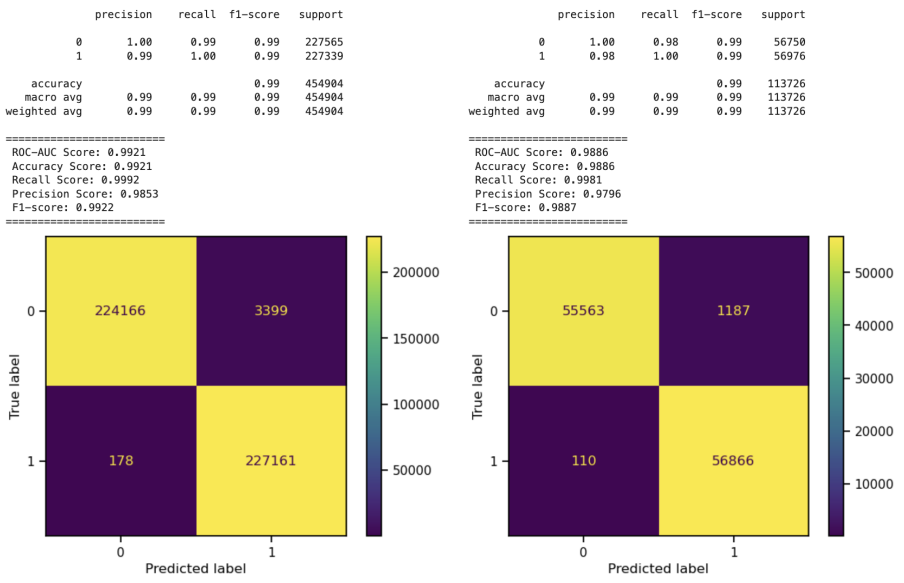
Next, the source codes in Figure 45 and 46 are implemented and an Evaluation is performed to show how well the model fits the reduced Train set and the Test set respectively.

```
# Evaluate the Train set
print(classification_report(y_reduced_train, knc_reduced_train_preds))
knc_reduced_cm_train = confusion_matrix(y_reduced_train, knc_reduced_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = knc_reduced_cm_train, display_labels = KNC_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_reduced_train, knc_reduced_train_preds), 4)
accuracy = round(accuracy_score(y_reduced_train, knc_reduced_train_preds), 4)
recall = round(recall_score(y_reduced_train, knc_reduced_train_preds), 4)
precision = round(precision_score(y_reduced_train, knc_reduced_train_preds), 4)
f_one = round(f1_score(y_reduced_train, knc_reduced_train_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

```
# Evaluate the Test set
print(classification_report(y_reduced_test, knc_reduced_test_preds))
knc_reduced_cm_test = confusion_matrix(y_reduced_test, knc_reduced_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = knc_reduced_cm_test, display_labels = KNC_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_reduced_test, knc_reduced_test_preds), 4)
accuracy = round(accuracy_score(y_reduced_test, knc_reduced_test_preds), 4)
recall = round(recall_score(y_reduced_test, knc_reduced_test_preds), 4)
precision = round(precision_score(y_reduced_test, knc_reduced_test_preds), 4)
f_one = round(f1_score(y_reduced_test, knc_reduced_test_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

Figure 45 and 46: Perform evaluations on reduced train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.



The results for both sets were obtained and the returned evaluation scores were respectively as follows.

Figure 47 and 48: Results of modelling.

The above figures show how a decision tree model performed on an original test set (right) and a training set (left). The model obtained 0.9921 ROC-AUC and accuracy, 0.9992 precision, 0.9853 recall, and 0.9922 F1-score on the training set. ROC-AUC on the test set dropped little to 0.9886, accuracy to 0.9886, precision to 0.9853, recall to 0.9992, and F1-score to 0.9922.

Summary

In this section, we have explored the two experiments: K-Nearest Neighbors modelling with original dataset and PCA-reduced dataset. Below table summarises the results from all k-nearest neighbors experiments.

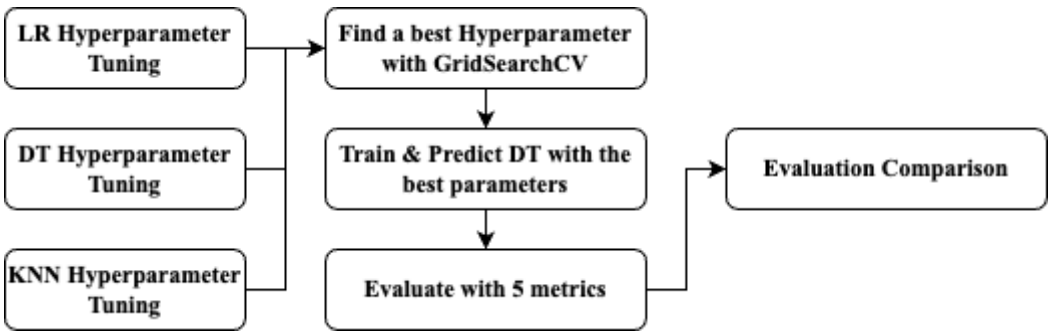
K-Nearest Neighbors																	
Experiments	Description	Dataset Type	ROC-AUC			Accuracy			Recall			Precision			F1-Score		
			Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap
Experiment 1	Base model	default	**0.9986**	**0.9978**	**0.0008**	**0.9986**	**0.9978**	**0.0008**	**1.0000**	**1.0000**	**0.0000**	**0.9973**	**0.9957**	**0.0016**	**0.9986**	**0.9978**	**0.0008**
Experiment 2	Base model	PCA Reduced	0.9921	0.9886	0.0035	0.9921	0.9886	0.0035	0.9992	0.9981	0.0011	0.9853	0.9796	0.0057	0.9922	0.9887	0.0035

Figure 49: Overall Results of two experiments with the K-Nearest Neighbors model.

According to the results, Experiment 1 outperformed all evaluation metrics, also returned a smaller gap in a lot of metrics. Therefore, We would select the K-Nearest Neighbors with original training and testing data sets for the hyperparameter tuning experiment.

4.5.4 Hyperparameter Tuning

Now we move on to the Hyperparameter Tuning Experiment to experiment whether the method optimises results in previous modelling implementations. The workflow pipeline is shown in the figure below.



Experiment 1: Logistic Regression Hyperparameter Tuning

This section documents the build of the Logistic Regression Hyperparameter Tuning. First, we would use GridSearchCV to find the best hyperparameter. Model training would be then implemented with non-reduced original dataset and best parameters.

```
# Define the hyperparameters and their possible values
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear', 'saga']
}

# Define multiple scoring metrics
scoring_metrics = {
    "ROC-AUC": make_scorer(roc_auc_score),
    'accuracy': make_scorer(accuracy_score),
    "recall": make_scorer(recall_score),
    "precision": make_scorer(precision_score),
    'f1': make_scorer(f1_score, average='weighted')
}

# Instantiate the grid search with the model and the hyperparameters
grid_search = GridSearchCV(estimator=LR, param_grid=param_grid,
                           cv=5, n_jobs=-1, verbose=2, scoring=scoring_metrics, refit="ROC-AUC")

# Fit the grid search to the reduced data
grid_search.fit(X_train, y_train)

# Search the best parameters with training data
lr_best_params = grid_search.best_params_
print("Best Hyperparameters:", lr_best_params)
```

```
Fitting 5 folds for each of 28 candidates, totalling 140 fits
Best Hyperparameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
```

Figure 50: Tune Hyperparameters with GridSearchCV and return the results.

The code in Figure 50 demonstrated instructions to fine-tune a logistic regression model using GridSearchCV. First, we have defined potential hyperparameters, assessed their performance using metrics like ROC-AUC, accuracy, recall, precision, and F1 score, and instantiated the grid search object. The grid search fit the model to the training data set, cycling through hyperparameter combinations to maximise the ROC-AUC score. Finally, each optimal parameter is identified and returned.

Then we would train the model class with original train sets and previously defined best parameters, make predictions on the train set and the test set.

```
# Define the best hyperparameters
C = lr_best_params["C"]
penalty = lr_best_params["penalty"]
solver = lr_best_params["solver"]

# Re-create the Logistic Regression class instance with the defined hyperparameters
lr_tuned = LogisticRegression(C = C, penalty = penalty, solver = solver)

# Fitting the model with training data
lr_tuned.fit(X_train, y_train)

# Re-predict the train and test set
lr_tuned_train_preds = lr_tuned.predict(X_train)
lr_tuned_test_preds = lr_tuned.predict(X_test)
```

Figure 51: Create an Instance of the LogisticRegression class with discovered parameters, fit and predict on the model with train and test set (original).

Next, the source codes in Figure 52 and 53 are implemented and an Evaluation is performed to show how well the model fits the Train set and the Test set respectively.

```
# Evaluate the Train set
print(classification_report(y_train, lr_tuned_train_preds))
lr_tuned_cm_train = confusion_matrix(y_train, lr_tuned_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = lr_tuned_cm_train, display_labels = lr_tuned.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_train, lr_tuned_train_preds), 4)
accuracy = round(accuracy_score(y_train, lr_tuned_train_preds), 4)
recall = round(recall_score(y_train, lr_tuned_train_preds), 4)
precision = round(precision_score(y_train, lr_tuned_train_preds), 4)
f_one = round(f1_score(y_train, lr_tuned_train_preds), 4)

# Return evaluation scores
print(f"=======================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=======================")
```

```
# Evaluate the Test set
print(classification_report(y_test, lr_tuned_test_preds))
lr_tuned_cm_test = confusion_matrix(y_test, lr_tuned_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = lr_tuned_cm_test, display_labels = lr_tuned.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_test, lr_tuned_test_preds), 4)
accuracy = round(accuracy_score(y_test, lr_tuned_test_preds), 4)
recall = round(recall_score(y_test, lr_tuned_test_preds), 4)
precision = round(precision_score(y_test, lr_tuned_test_preds), 4)
f_one = round(f1_score(y_test, lr_tuned_test_preds), 4)

# Return evaluation scores
print(f"=======================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=======================")
```

Figure 52 and 53: Perform evaluations on reduced train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

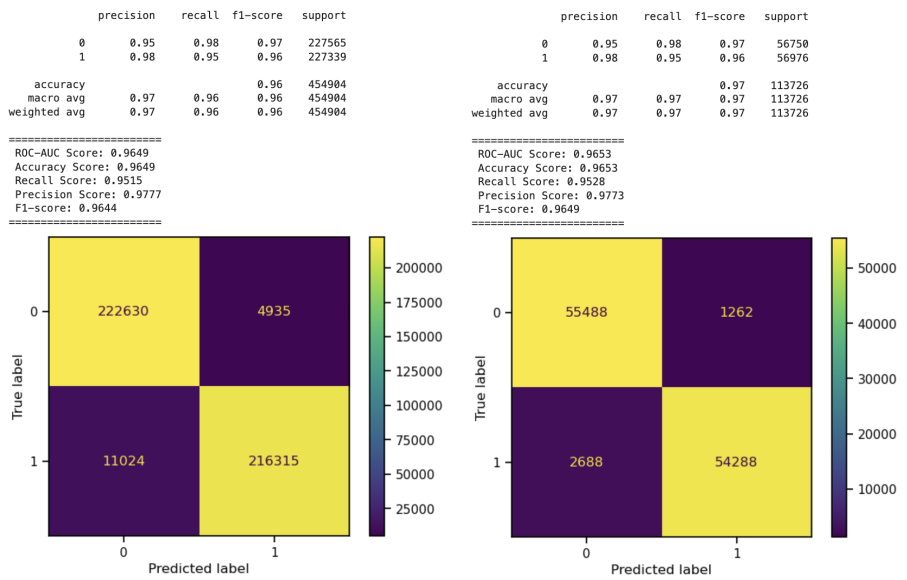


Figure 54 and 55: Results of modelling.

The fine-tuned logistic regression model obtained 0.9649 ROC-AUC and accuracy, 0.9515 precision, 0.9777 recall, and 0.9644 F1-score on the training set. ROC-AUC on the test set dropped little to 0.9653, accuracy to 0.9653, recall to 0.9528, precision to 0.9773, and F1-score to 0.9649.

Model Implementation Results with tuned Hyperparameters																
Model	Hyperparameter	ROC-AUC			Accuracy			Recall			Precision			F1-Score		
		Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap	Train	Test	Gap
Logistic Regression	default	0.9649	0.9653	0.0004	0.9649	0.9653	0.0004	0.9515	0.9528	0.0013	0.9777	0.9773	0.0004	0.9644	0.9649	0.0005
	C, penalty, solver	0.9649	0.9653	0.0004	0.9649	0.9653	0.0004	0.9515	0.9528	0.0013	0.9777	0.9773	0.0004	0.9644	0.9649	0.0005

The above table is the comparison of evaluations on Logistic Regression base model and fine-tuned Logistic Regression with Hyperparameter Tuning. As the results in the table show, we did not observe improvements in optimisation with Hyperparameter Tuning using the Logistic Regression model.

Experiment 2: Decision Tree Hyperparameter Tuning

This section documents the build of the Logistic Regression Hyperparameter Tuning. First, we would use GridSearchCV to find the best hyperparameter. Model training would be then implemented with non-reduced original dataset and best parameters.

```
# define some candidate parameter values
params_grid = {
    'max_depth': [1, 3, 5, 7, 8, 10, 12, 14, 15, 17, 18, 20],
    'min_samples_split': [8, 10, 12, 18, 20, 16]
}

# Define multiple scoring metrics
scoring_metrics = {
    "ROC-AUC": make_scorer(roc_auc_score),
    'accuracy': make_scorer(accuracy_score),
    "recall": make_scorer(recall_score),
    "precision": make_scorer(precision_score),
    'f1': make_scorer(f1_score, average='weighted')
}

# Instantiate the grid search with the model and the hyperparameters
grid_search = GridSearchCV(estimator=DT, param_grid=params_grid,
                           cv=5, n_jobs=-1, verbose=2, scoring=scoring_metrics, refit = "ROC-AUC")

# Search the best parameters with training data
grid_search.fit(X_train, y_train)

# Search the best parameters with training data
dt_best_params = grid_search.best_params_
print("Best Hyperparameters:", dt_best_params)
```

```
Fitting 5 folds for each of 72 candidates, totalling 360 fits
Best Hyperparameters: {'max_depth': 20, 'min_samples_split': 8}
```

Figure 56: Tune Hyperparameters with GridSearchCV and return the results.

The code in Figure 56 demonstrated instructions to fine-tune a logistic regression model using GridSearchCV. First, we have defined potential hyperparameters, assessed their performance using metrics like ROC-AUC, accuracy, recall, precision, and F1 score, and instantiated the grid search object. The grid search fit the model to the training data set, cycling through hyperparameter combinations to maximise the ROC-AUC score. Finally, each optimal parameter is identified and returned.

Then we would train the model class with original train sets and previously defined best parameters, make predictions on the train set and the test set.

```
# Define the best hyperparameters
max_depth = dt_best_params["max_depth"]
min_samples_split = dt_best_params["min_samples_split"]

# Re-create the Logistic Regression class instance with the defined hyperparameters
dt_tuned = DecisionTreeClassifier(max_depth = max_depth, min_samples_split = min_samples_split)

# Fitting the model with training data
dt_tuned.fit(X_train, y_train)

# Re-predict the train and test set
dt_tuned_train_preds = dt_tuned.predict(X_train)
dt_tuned_test_preds = dt_tuned.predict(X_test)
```

Figure 57: Create an Instance of the LogisticRegression class with discovered parameters, fit and predict on the model with train and test set (original).

Next, the source codes in Figure 58 and 59 are implemented and an Evaluation is performed to show how well the model fits the Train set and the Test set respectively.

```python
# Evaluate the Train set
print(classification_report(y_train, dt_tuned_train_preds))
dt_tuned_cm_train = confusion_matrix(y_train, dt_tuned_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = dt_tuned_cm_train, display_labels = dt_tuned.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_train, dt_tuned_train_preds), 4)
accuracy = round(accuracy_score(y_train, dt_tuned_train_preds), 4)
recall = round(recall_score(y_train, dt_tuned_train_preds), 4)
precision = round(precision_score(y_train, dt_tuned_train_preds), 4)
f_one = round(f1_score(y_train, dt_tuned_train_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

```python
# Evaluate the Test set
print(classification_report(y_test, dt_tuned_test_preds))
dt_tuned_cm_test = confusion_matrix(y_test, dt_tuned_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = dt_tuned_cm_test, display_labels = dt_tuned.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_test, dt_tuned_test_preds), 4)
accuracy = round(accuracy_score(y_test, dt_tuned_test_preds), 4)
recall = round(recall_score(y_test, dt_tuned_test_preds), 4)
precision = round(precision_score(y_test, dt_tuned_test_preds), 4)
f_one = round(f1_score(y_test, dt_tuned_test_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

Figure 58 and 59: Perform evaluations on reduced train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

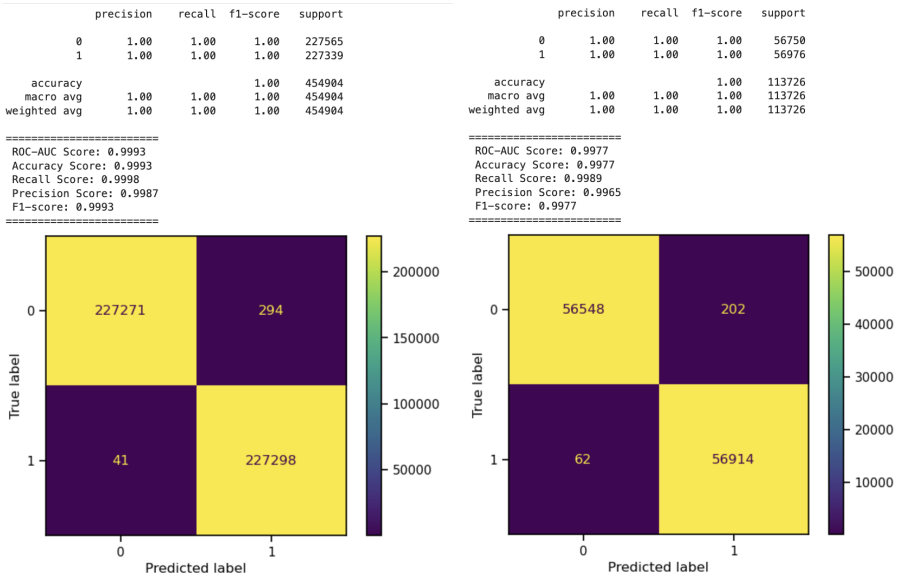


Figure 60 and 61: Results of modelling.

The fine-tuned decision tree model obtained 0.9993 ROC-AUC and accuracy, 0.9998 precision, 0.9987 recall, and 0.9993 F1-score on the training set. ROC-AUC on the test set dropped little to 0.9977, accuracy to 0.9977, recall to 0.9989, precision to 0.9965, and F1-score to 0.9977.

Model Implementation Results with tuned Hyperparameters																	
Model	**Hyperparameter**	**ROC-AUC**			**Accuracy**			**Recall**			**Precision**			**F1-Score**			
		Train	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	
Decision Tree	default	**1.0000**	**0.9981**	0.0019	**1.0000**	**0.9981**	0.0019	**1.0000**	**0.9990**	0.0010	**1.0000**	**0.9972**	0.0028	**1.0000**	**0.9981**	0.0019	
	max_depth, min_samples_split	0.9993	0.9977	**0.0016**	0.9993	0.9977	**0.0016**	0.9998	0.9989	**0.0009**	0.9987	0.9965	**0.0022**	0.9993	0.9977	**0.0016**	

The above table is the comparison of evaluations on Decision Tree base model and fine-tuned Decision Tree with Hyperparameter Tuning. Although the difference in results between the Train and Test sets was slightly reduced, the overall Evaluation was better for the Base Model before Hyperparameter Tuning, and the Decision Tree also showed no dramatic improvement using Hyperparameter Tuning.

Experiment 3: K-Nearest Neighbors Hyperparameter Tuning

This section documents the build of the K-Nearest Neighbors Hyperparameter Tuning. First, we would use GridSearchCV to find the best hyperparameter. Model training would be then implemented with non-reduced original dataset and best parameters.

```
# Define the hyperparameters and their possible values
params_grid = {
    'n_neighbors' : [5,7,9,11,13,15],
    'weights' : ['uniform','distance'],
    'metric' : ['minkowski','euclidean','manhattan']
}

# Define multiple scoring metrics
scoring_metrics = {
    "ROC-AUC": make_scorer(roc_auc_score),
    'accuracy': make_scorer(accuracy_score),
    "recall": make_scorer(recall_score),
    "precision": make_scorer(precision_score),
    'f1': make_scorer(f1_score, average='weighted')
}

# Instantiate the grid search with the model and the hyperparameters
grid_search = GridSearchCV(estimator=KNC, param_grid=params_grid,
                           cv=5, n_jobs=-1, verbose=2, scoring=scoring_metrics, refit='ROC-AUC')

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Print the best hyperparameters
KNC_best_params = grid_search.best_params_
print("Best Hyperparameters:", KNC_best_params)
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
Best Hyperparameters: {'metric': 'manhattan', 'n_neighbors': 5, 'weights': 'distance'}
```

Figure 62: Tune Hyperparameters with GridSearchCV and return the results.

The code in Figure 62 demonstrated instructions to fine-tune a decision tree model using GridSearchCV. First, we have defined potential hyperparameters, assessed their performance using metrics like ROC-AUC, accuracy, recall, precision, and F1 score, and instantiated the grid search object. The grid search fit the model to the training data set, cycling through hyperparameter combinations to maximise the ROC-AUC score. Finally, each optimal parameter is identified and returned.

Then we would train the k-nearest neighbors classifier with original train sets and previously defined best parameters, make predictions on the train set and the test set.

```
# Define the best hyperparameters
n_neighbors = KNC_best_params["n_neighbors"]
weights = KNC_best_params["weights"]
metric = KNC_best_params["metric"]

# Re-create the Logistic Regression class instance with the defined hyperparameters
knc_tuned = KNeighborsClassifier(n_neighbors = n_neighbors, weights = weights, metric = metric)

# Fitting the model with training data
knc_tuned.fit(X_train, y_train)

# Re-predict the train and test set
knc_tuned_train_preds = knc_tuned.predict(X_train)
knc_tuned_test_preds = knc_tuned.predict(X_test)
```

Figure 63: Create an Instance of the KNearestClassifier with discovered parameters, fit and predict on the model with train and test set (original).

Next, the source codes in Figure 64 and 65 are implemented and an Evaluation is performed to show how well the model fits the Train set and the Test set respectively.

```
# Evaluate the Train set
print(classification_report(y_train, knc_tuned_train_preds))
knc_tuned_cm_train = confusion_matrix(y_train, knc_tuned_train_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = knc_tuned_cm_train, display_labels = KNC_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_train, knc_tuned_train_preds), 4)
accuracy = round(accuracy_score(y_train, knc_tuned_train_preds), 4)
recall = round(recall_score(y_train, knc_tuned_train_preds), 4)
precision = round(precision_score(y_train, knc_tuned_train_preds), 4)
f_one = round(f1_score(y_train, knc_tuned_train_preds), 4)

# Return evaluation scores
print(f"========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"========================")
```

```
# Evaluate the Test set
print(classification_report(y_test, knc_tuned_test_preds))
knc_tuned_cm_test = confusion_matrix(y_test, knc_tuned_test_preds)
sns.set_context("notebook")
ConfusionMatrixDisplay(confusion_matrix = knc_tuned_cm_test, display_labels = KNC_reduced.classes_).plot()

# Evaluation Scores
roc_auc = round(roc_auc_score(y_test, knc_tuned_test_preds), 4)
accuracy = round(accuracy_score(y_test, knc_tuned_test_preds), 4)
recall = round(recall_score(y_test, knc_tuned_test_preds), 4)
precision = round(precision_score(y_test, knc_tuned_test_preds), 4)
f_one = round(f1_score(y_test, knc_tuned_test_preds), 4)

# Return evaluation scores
print(f"=========================")
print(f" ROC-AUC Score: {roc_auc}")
print(f" Accuracy Score: {accuracy}")
print(f" Recall Score: {recall}")
print(f" Precision Score: {precision}")
print(f" F1-score: {f_one}")
print(f"=========================")
```

Figure 64 and 65: Perform evaluations on reduced train and test set via 5 evaluation scores.

The results for both sets were obtained and the returned evaluation scores were respectively as follows.

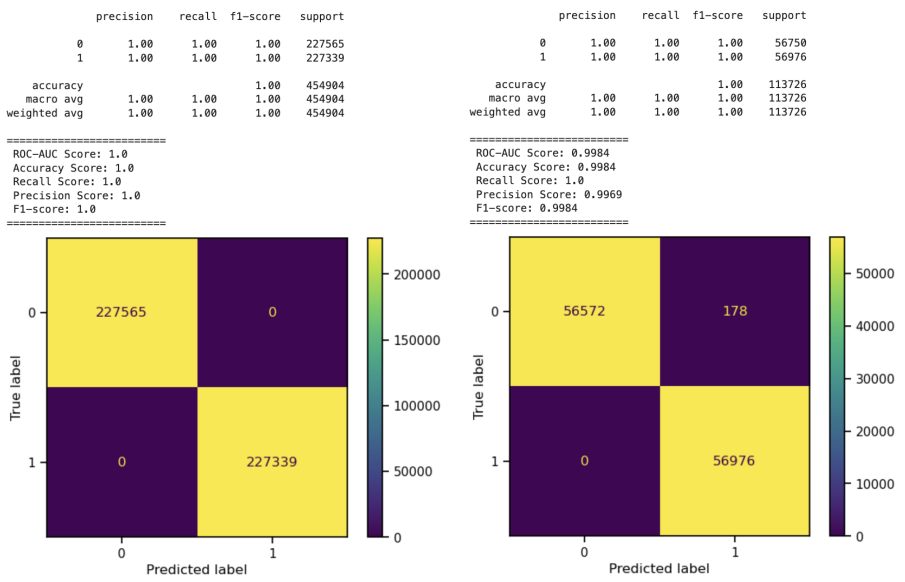


Figure 66 and 67: Results of modelling.

The fine-tuned k-nearest neighbors model obtained 1.0 ROC-AUC and accuracy, precision, recall, and F1-score on the training set. ROC-AUC on the test set dropped little to 0.9984, accuracy to 0.9984, recall to 1.0, precision to 0.9969, and F1-score to 0.9984.

Model Implementation Results with tuned Hyperparameters																	
Model	**Hyperparameter**	**ROCAUC**			**Accuracy**			**Recall**			**Precision**			**F1Score**			
		Train	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	
KNearest Neighbors	default	0.9986	0.9978	**0.0008**	0.9986	0.9978	**0.0008**	**1.0000**	**1.0000**	**0.0000**	0.9973	0.9957	**0.0016**	0.9986	0.9978	**0.0008**	
	n_neighbors, weights, metric	**1.0000**	**0.9984**	0.0016	**1.0000**	**0.9984**	0.0016	**1.0000**	**1.0000**	**0.0000**	**1.0000**	**0.9969**	0.0031	**1.0000**	**0.9984**	0.0016	

The above table is the comparison of evaluations on K-Nearest Neighbors base model and fine-tuned Logistic Regression with Hyperparameter Tuning. The results in the table show that for KNN, finding and specifying the optimal parameters through Hyperparameter Tuning resulted in a slight improvement in the results for both the Training and Test sets. On the other hand, the difference between the results of the Training set and the Test set was slightly wider than the Evaluation of the Base Model, suggesting that the robustness of the model may have been weakened by this.

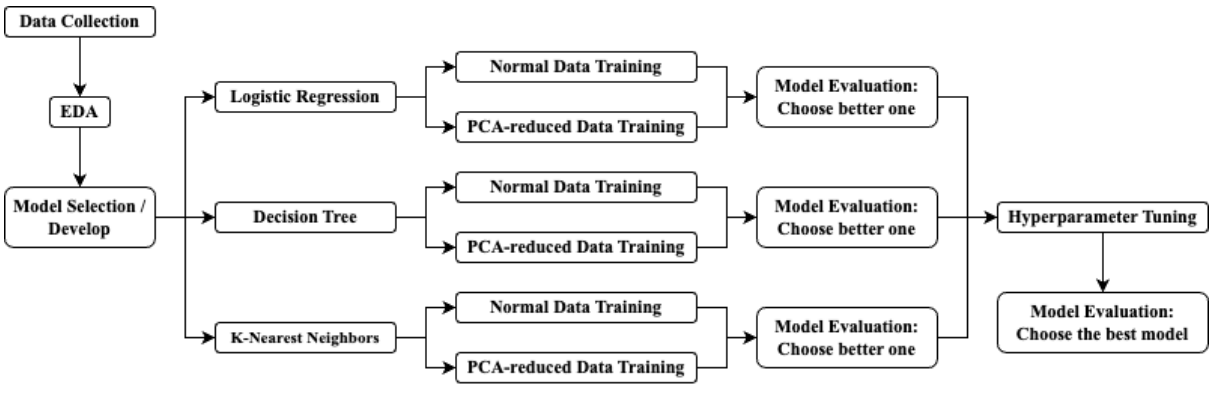
Summary

Model Implementation Results with tuned Hyperparameters																	
Model	**Hyperparameter**	**ROCAUC**			**Accuracy**			**Recall**			**Precision**			**F1Score**			
		Train	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	
Logistic Regression	C, penalty, solver	0.9649	0.9653	**0.0004**	0.9649	0.9653	**0.0004**	0.9515	0.9528	0.0013	0.9777	0.9773	**0.0004**	0.9644	0.9649	**0.0005**	
Decision Tree	max_depth, min_samples_split	0.9993	0.9977	0.0016	0.9993	0.9977	0.0016	0.9998	0.9989	0.0009	0.9987	0.9965	0.0022	0.9993	0.9977	0.0016	
K-Nearest Neighbors	n_neighbors, weights, metric	**1.0000**	**0.9984**	0.0016	**1.0000**	**0.9984**	0.0016	**1.0000**	**1.0000**	**0.0000**	**1.0000**	0.9969	0.0031	**1.0000**	**0.9984**	0.0016	

In this section, the models with better results in logistic regression, decision trees and k-nearest neighbour methods were selected from the results of the experiments up to the previous section, and the best hyper-parameters for each were found through GridSearchCV to see if further optimisation of the evaluation could be expected.

As a result, K-Nearest Neighbors with fine-tuned hyperparameters was outperformed the most among three models with fine-tuned hyperparameters.

4.6 Summary of Implementations and Results



This study section presents a thorough method for applying machine learning models to identify credit card fraud. It starts with an implementation pipeline, showing the step-by-step process. The initial phase involved Exploratory Data Analysis (EDA), examining dataset

summaries to understand the data's structure and statistical properties. The dataset consisted of 31 columns and 568,630 rows, signifying considerable data volume.

The analysis confirmed no missing values, simplifying preprocessing by avoiding data imputation. The target feature, "Class," showed a balanced distribution between "Frauds" and "No Frauds," implying suitability for most machine learning models. Tools like Matplotlib and Seaborn were used for more descriptive EDA. A correlation matrix heatmap revealed variable relationships, avoiding multicollinearity for model accuracy.

Principal Component Analysis (PCA) was used for dimensionality reduction, enhancing model performance. The process involved feature standardisation and data transformation to reduced dimensional space. The optimal number of principal components was determined using an "Elbow Curve" of eigenvalues, preventing overfitting and reducing computational costs.

A new DataFrame with PCA-reduced features and the original target variable 'Class' was created, ready for machine learning tasks. The data was split into training and testing sets, preparing for the modelling phase.

Three machine learning models: Logistic Regression, Decision Tree, and K-Nearest Neighbors were evaluated using both original and PCA-reduced datasets. The best dataset was selected for further optimization through hyperparameter tuning, ensuring the most effective model for credit card fraud detection.

Below table shows evaluations of each experiment’s outcome, and the best model.

Logistic Regression																		
Algorithm	**Description**	**Dataset Type**	**ROCAUC**			**Accuracy**			**Recall**			**Precision**			**F1Score**		
			Train	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**	**Train**	**Test**	**Gap**
Logistic Regression	Base model	default	0.9649	0.9653	**0.0004**	0.9649	0.9653	**0.0004**	0.9515	0.9528	0.0013	0.9777	0.9773	0.0004	0.9644	0.9649	**0.0005**
	Base model	PCA Reduced	0.9495	0.9501	0.0006	0.9495	0.9501	0.0006	0.9201	0.9213	0.0012	0.9776	0.9777	**0.0001**	0.9479	0.9487	0.0008
	Hyperparameter Tuning	default	0.9649	0.9653	**0.0004**	0.9649	0.9653	**0.0004**	0.9515	0.9528	0.0013	0.9777	0.9773	0.0004	0.9644	0.9649	**0.0005**
Decision Tree	Base model	default	**1.0000**	0.9981	0.0019	**1.0000**	0.9981	0.0019	**1.0000**	0.9990	0.0010	**1.0000**	**0.9972**	0.0028	**1.0000**	0.9981	0.0019
	Base model	PCA Reduced	**1.0000**	0.9926	0.0074	**1.0000**	0.9926	0.0074	**1.0000**	0.9939	0.0061	**1.0000**	0.9914	0.0086	**1.0000**	0.9926	0.0074
	Hyperparameter Tuning	default	0.9993	0.9977	0.0016	0.9993	0.9977	0.0016	0.9998	0.9989	0.0009	0.9987	0.9965	0.0022	0.9993	0.9977	0.0016
KNearest Neighbors	Base model	default	0.9986	0.9978	0.0008	0.9986	0.9978	0.0008	1.0000	1.0000	0.0000	0.9973	0.9957	0.0016	0.9986	0.9978	0.0008
	Base model	PCA Reduced	0.9921	0.9886	0.0035	0.9921	0.9886	0.0035	0.9992	0.9981	0.0011	0.9853	0.9796	0.0057	0.9922	0.9887	0.0035
	Hyperparameter Tuning	default	**1.0000**	**0.9984**	0.0016	**1.0000**	**0.9984**	0.0016	**1.0000**	**1.0000**	**0.0000**	**1.0000**	0.9969	0.0031	**1.0000**	**0.9984**	0.0016

5.0 Analyses and Recommendations

This study aimed to ensure the best performance of predictive models in real-world scenarios by considering various machine learning models and understanding which models can most accurately predict fraudulent transactions based on historical and real-time data, while optimising and fine-tuning aspects of the machine learning pipeline, such as data pre-processing, feature engineering and hyper-parameter tuning.

With analysing the experimental outcomes to achieve these objectives, it became evident that optimisation methods like PCA dimensionality reduction or hyperparameter tuning do not always guarantee better outcomes. Indeed, models with PCA-reduced training and testing data set performed less accurately than original data sets, on all three machine learning models. Furthermore, the logistic regression with fine-tuned hyperparameters was not optimised at all and showed almost identical evaluation scores to the previous evaluation, the decision tree with fine-tuned hyperparameters reduced the gap in evaluation scores between Train-Tests to some extent, but the overall evaluation was worse than before tuning, and the K-Nearest Neighbors with fine-tuned hyperparameters, the overall rating could be slightly raised, but it would create a gap in the Train-Test rating score that was almost twice as large.

Author	Ratio of Classes	Feature Selection	Resampling	Model Types	AUC	Accuracy	F1Score	Precision	Recall
Alfaiz, N. S., & Fati, S. M. (2022).	0.2 99.8 Fraud NotFraud	No	AIIKNN	AIIKNN + CatBoost	0.9794	**0.9996**	0.9591	0.8028	0.8740
Mienye, I. D., & Sun, Y. (2023).	0.2 99.8 Fraud NotFraud	No	No	IGGAW	0.9900	N.A.	N.A.	N.A.	0.9970
Malik, E. F., Khaw, K. W., Belaton, B., Wong, W. P., & Chew, X. (2022).	3.0 97.0 Fraud NotFraud	SVMRFE	SVMRFE	AdaBoost + LGBM	N.A.	0.0020	0.7700	0.9700	0.6400
Our model	50.0 50.0 Fraud NotFraud	No	No	KNN + Hyperparameter Tuning	**0.9984**	0.9984	**0.9984**	**0.9969**	**1.0000**

In the experiment evaluation outcomes among benchmark and our implemented results, our implementation has gotten over the highest score in most of evaluation metrics, despite Accuracy score is slightly lower than the model of AllKNN + CatBoost implemented by Alfaiz. N. S., & Fati. S. M. (2022). Furthermore, it is worth noting that the evaluation scores for logistic regression and decision trees, as well as for other conditions, were above for a significant number of metrics, with almost no difference when compared to the benchmark.

On the other hand, from the perspective of further optimising the evaluation of machine learning models using Hyperparameter Tuning and PCA Dimensionality Reduction, we could not achieve the desired results. Our method could not achieve what we have expected, and this could be due to a number of factors, including the reduction of even variables that should

have contained important information, the fact that the initial parameters used by default were already sufficiently optimal, and over-fitting caused by Hyperparameter Tuning.

Based on the observations from the analyses, the recommendation for model optimization would involve a multifaceted approach.

- Reassessment of Feature Selection: Instead of relying solely on PCA for dimensionality reduction, which may remove features containing important information, alternative feature selection methods should be explored. Techniques such as feature importance from tree-based models, or domain-specific feature engineering could yield better performance.

- Hyperparameter Tuning Strategy: The process of hyperparameter tuning should be refined. It could involve a more exhaustive search, such as a randomised search to explore a broader range of values, or Bayesian optimization methods which are more efficient than GridSearchCV.

- Overfitting Prevention: Special attention should be paid to avoid overfitting during hyperparameter tuning. This can be done by using cross-validation strategies that include a validation set to monitor for overfitting and by implementing regularisation techniques.

6.0 Conclusion

Throughout our comprehensive and thorough exploration of a variety of machine learning models and optimisation techniques, fine-tuned K-Nearest Neighbors with original dataset was evaluated as the most accurate for credit card fraud detection. This model achieved a ROC-AUC score of 0.9984, accompanied by accuracy of 0.9984, F1-score of 0.9984, precision of 0.9969, and recall of 1.0000. Further practical validation may be warranted.

7.0 References

Alfaiz, N. S., & Fati, S. M. (2022). Enhanced Credit Card Fraud Detection Model Using Machine Learning. Electronics, 11(4), 662. https://www.mdpi.com/2079-9292/11/4/662.

Bock, T. (2018). What is a Correlation Matrix? | Displayr. Displayr. https://www.displayr.com/what-is-a-correlation-matrix/.

Chung, J., & Lee, K. (2023). Credit Card Fraud Detection: An Improved Strategy for High Recall Using KNN, LDA, and Linear Regression. Sensors, 23(18), 7788. https://www.mdpi.com/1424-8220/23/18/7788.

Elgiriyewithana, N. (2023, September). Credit Card Fraud Detection Dataset 2023. Www.kaggle.com. https://www.kaggle.com/datasets/nelgiriyewithana/credit-card-fraud-detection-dataset-2023.

Malik, E. F., Khaw, K. W., Belaton, B., Wong, W. P., & Chew, X. (2022). Credit Card Fraud Detection Using a New Hybrid Machine Learning Architecture. Mathematics, 10(9), 1480. https://www.mdpi.com/2227-7390/10/9/1480.

Mienye, I. D., & Sun, Y. (2023). A Machine Learning Method with Hybrid Feature Selection for Improved Credit Card Fraud Detection. Applied Sciences, 13(12), 7254. https://www.mdpi.com/2076-3417/13/12/7254.

Rej, M. (2023, June 20). Credit Card Fraud Statistics (2023) | Merchant Cost Consulting. Merchantcostconsulting.com. https://merchantcostconsulting.com/lower-credit-card-processing-fees/credit-card-fraud-statistics/.