Fraud Detection using Machine Learning in e-Commerce

By

Ang Su Huan

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION SYSTEMS (HONOURS) BUSINESS INFORMATION SYSTEMS

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 2025

COPYRIGHT STATEMENT

This Final Year Project proposal is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Systems (Honours) Business Information Systems at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project proposal represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project proposal may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisors, Ms. Nurul Syafidah Binti Jamil and my moderator, Mr. Choo Peng Yin for giving me the valuable opportunity to engage in a project on fraud detection using machine learning in the eCommerce domain. This project provided me with hands-on experience in real-world data analysis, model training and critical thinking, marking an important first step in establishing my career in data science and machine learning. Finally, I would like to express my sincere gratitude to my parents and my family for their endless love, support, and motivation throughout the course.

ABSTRACT

The fast growth of e-commerce has resulted in a rise in fraudulent activities, posing significant challenges to the security and trust of online transactions. Traditional fraud detection methods often fall short in effectively identifying complex fraud patterns due to issues like data imbalance, misclassification of costly errors, and the evolving nature of fraud tactics. This research proposes a machine learning-based approach to improve fraud detection performance in e-commerce platforms. Resampling techniques like SMOTE, oversampling and under-sampling are applied to address class imbalance issue. The study aims to reduce false negatives and enhance the detection of rare fraudulent transactions. Ensemble models such as Random Forest, AdaBoost, and XGBoost, will be employed to capture complex patterns and improve model performance. A systematic model evaluation was conducted using metrics such as accuracy, F1-score, MCC, precision, recall and AUC to ensure robust performance. Experimental results showed that Random Forest combined with oversampling achieved the best trade-off between precision and recall, reducing false negatives while maintaining high overall accuracy. Robustness was further validated through testing on both synthetic datasets and the Kaggle dataset, confirming the model's adaptability and reliability. Finally, the best-performing model was integrated into a Power BI dashboard, enabling real-time monitoring of fraud detection results and visualization of emerging fraud trends. This integration supports decision-making by providing stakeholders with timely insights. The study contributes to the development of adaptive fraud detection systems capable of mitigating financial risks and maintaining customer trust in the e-commerce sector.

Area of Study: Fraud Detection in E-commerce

Keywords: E-commerce, Fraud Detection, Machine Learning, SMOTE, Ensemble Learning, Power BI, Credit Card Fraud

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

LIST OF ABBREVIATIONS

AdaBoost	Adaptive Boosting
ANN	Artificial Neural Networks
AP	Average Precision
AUC-ROC	Area Under the Receiver Operating Characteristic Curve
AUC	Area Under the Curve
CNP	Card Not Present
CP	Card Present
CTGAN	Conditional Tabular Generative Adversarial Network
DAX	Data Analysis Expressions
DT	Decision Tree
FN	False Negative
FP	False Positive
iForest	Isolation Forest
KNN	K-Nearest Neighbours
LDA	Linear Discriminant Analysis
LightGBM	Light Gradient Boosting Machine
LR	Logistic Regression
LOC	Local Outlier Factor
MAE	Mean Absolute Error
MCC	Matthews Correlation Coefficient
NB	Naïve Bayes
PCA	Principal Component Analysis
RF	Random Forest
RMSE	Root Mean Squared Error
ROC	Receiver Operating Characteristic
SMOTE	Synthetic Minority Over-sampling Technique
SVM	Support Vector Machine
TVAE	Tabular Variational Autoencoder
XGBoost	Extreme Gradient Boosting

CHAPTER 1

Introduction

This chapter outlines the research background and motivation, highlighting the key problems and the need for improved fraud detection in e-commerce. This chapter also presents the objectives of the project, the scope and direction of the study, and the contributions made to the field.

E-commerce is the result of a significant change driven by the rapid evolution of digital technologies into conventional business methods [1]. It involves the buying and selling of products, services and information through electronic platforms over the Internet [1]. The rise of e-commerce has enabled business to reach global consumers, reduce costs, offer greater flexibility for consumers, respond quickly to market demands, support various payment methods and make transactions easier and faster through technology [2]. The accessibility and ease of online shopping have made it very popular, changing the way people interact with businesses and make purchases. However, with the swift expansion of e-commerce, fraud has become a significant challenge. As online transactions have increased, fraudsters have more opportunities to exploit the vulnerabilities of digital systems by using advanced techniques to bypass security measures [3].

Fraud in e-commerce can take many forms, including card not present (CNP), fake websites, chargeback fraud, account takeovers, identity thefts and phishing [4]. CNP fraud, in particular, occurs when stolen or counterfeit credit card details are used for online purchases without the physical card being required. This makes it one of the most prevalent and costly forms of fraud in digital commerce. Fraudsters can exploit these transactions to make unauthorized purchases, resulting in substantial financial losses for both consumers and businesses [5]. This form of fraud is particularly concerning due to its widespread impact. Consumers may face unauthorized charges on their accounts, while business must deal with chargebacks, legal fees and reputational damage.

These fraudulent activities not only lead to financial losses but also damage customer trust [6], which is a factor essential for the sustained success of e-commerce businesses. As customers

grow increasingly concerned about the security of their payment information, e-commerce businesses must navigate the challenge of maintaining trust while providing a seamless shopping experience. To address these risks, e-commerce platforms are deployed machine learning methods in fraud detection systems such as Logistic Regression, Random Forest, Naïve Bayes, Support Vector Machine (SVM) and others to identify and block fraudulent transactions in real time [6].

Traditional fraud detection methods are always relying on static rules and manual checks, which are become less effective against the evolving tactics of fraudsters [7]. These systems are difficult to detect complex fraud patterns because there are very few examples of fraud [6]. By using machine learning in fraud detection, e-commerce businesses can significantly improve accuracy in identifying fraudulent transactions [7]. Ultimately, this leads to better customer satisfaction, increased trust, and the continued growth of e-commerce businesses.

1.1 Problem Statement and Motivation

In the e-commerce industry, fraud detection is vital for ensuring the security of transactions and maintain customer trust. However, several problems in fraud detection systems hinder their effectiveness.

1. Data imbalance in fraud detection datasets.

Fraudulent transactions represent a very small proportion of the total dataset, often less than 1%, leading to an imbalance [8]. This imbalance leads to machine learning models become biased toward the normal transactions, which are the majority. This makes the models effective at identifying legitimate transactions but difficult to detect uncommon and critical fraudulent transactions which are the minority [1]. As a result, many fraud cases are missed, increasing financial losses and reducing trust in e-commerce system. Traditional resampling methods like oversampling and under sampling help to balance the dataset, but they can introduce new issues like overfitting or loss of useful data. To tackle this, resampling technique like SMOTE are needed to build models that effectively detect uncommon fraudulent transactions while maintaining accuracy [9].

2. Misclassification in machine learning

Misclassification in machine learning occurs when a model treats all errors equally. Most models aim to minimize errors without considering that some mistakes are more costly than others, this is known as cost-sensitive problem [9]. In fraud detection, failing to identify a fraudulent transaction (false negative) is much more damaging than incorrectly flagging a legitimate transaction as fraudulent (false positive). This is because they allow fraudulent activities to go undetected, leading to financial losses and potential reputational damage. This problem is further complicated by the overlap between legitimate and fraudulent transactions, especially when patterns change over time, then worsens this issue. Such misclassifications strain resources, impacts customer experience, and reduces overall detection effectiveness [9]. To address this issue, ensemble learning models, which are more robust and capable of capturing complex and non-linear relationships, can be combined with resampling techniques. These models can better recognize minority class patterns (i.e., fraud), thereby minimizing the costly errors, especially false negatives and improving overall prediction accuracy.

3. Evolving Nature of fraud

Fraudulent patterns are not only rare but also change over time, making it challenging for detection models to remain effective [1,10]. This phenomenon, known as concept drift, occurs when fraudsters adapt their methods to bypass detection, while legitimate users may change their spending behaviors [9]. If detection models are not updated regularly, their accuracy declines, leading to missed fraud cases, financial losses, and a poor user experience. To address this, a Power BI dashboard will be developed to monitor model performance and the fraud patterns in real-time, allowing for continuous tracking of effectiveness and enabling timely updates to adapt to shifting fraud patterns.

Motivation

The rapid expansion of e-commerce has resulted in a rise in fraudulent activities, creating major challenges to the security of online transactions. While machine learning offers an effective solution, but issues like data imbalance, misclassification of costly errors, and the evolving nature of fraud still limit model performance. Fraudulent transactions are rare, causing models to be biased towards legitimate ones, and missing fraud (false negatives) is more costly than

incorrectly flagging legitimate transactions [9]. Moreover, as fraud tactics evolve, models can become outdated and ineffective.

The motivation behind this work is to address these gaps by designing a strong machine learning model capable of accurately detects fraud while considering the costs of different types of errors. Additionally, the model will be integrated with a monitoring dashboard, enabling continuous tracking of its performance and allowing for timely updates as fraud patterns change. This approach will enhance detection accuracy, improve system adaptability, and ultimately support better security and customer trust in e-commerce.

1.2 Objectives

The aim of this research is to develop advanced machine learning models and visualization tools for improving fraud detection in e-commerce, specifically targeting credit card transactions. By addressing problems like data imbalance, cost-sensitive misclassification, and evolving fraud patterns, this study seeks to improve the performance and adaptability of fraud detection systems. In this research, ensemble learning models with resampling techniques are proposed, along with an interactive Power BI dashboard for real-time monitoring and performance tracking.

Main Objective:

To integrate machine learning algorithm with a Power BI dashboard for real-time monitoring and performance tracking.

Sub Objectives:

- To enhance fraud detection performance by addressing data imbalance through resampling techniques.
- To develop ensemble models that reduce misclassification errors.
- To visualize fraud detection model performance and fraud patterns using Power BI.

1.3 Project Scope and Direction

The research is focusing on fraud detection in e-commerce by using advanced machine learning techniques. The study aims to address critical challenges such as data imbalance, cost-sensitive problem and evolving fraud patterns, which are common in fraud detection.

This project focuses on fraud detection in e-commerce, specifically targeting credit card fraud within *Card Not Present (CNP)* transactions. Other sectors such as banking, healthcare, and insurance are excluded, as are other e-commerce fraud types like account takeovers, chargebacks, and promotional abuse. The study also excludes alternative payment methods such as e-wallets, cryptocurrencies, and bank transfers. By narrowing the scope to credit card CNP transactions in e-commerce, this research aims to design and refine machine learning models that are directly relevant to current industry challenges. This focused approach avoids the added complexity of multiple fraud types and payment methods, enabling more accurate and effective model development.

The research also focuses on employing ensemble learning models such as Random Forest, AdaBoost, and XGBoost exclusively for fraud detection tasks. Resampling techniques, such as SMOTE, Oversampling and Under-sampling are applied to these models to address the issue of class imbalance and reduce the impact of misclassification. These models and techniques aim to enhance the models' ability to identify fraudulent transactions accurately, especially by reducing false negatives, while maintaining high overall performance.

The scope of Power BI development in this research focuses on creating a user-friendly dashboard to visualize and monitor the performance of fraud detection models in real time. The dashboard will visualize key metrics such as accuracy, precision, recall, F1-score, AUC and confusion matrices, along with fraud patterns and trends.

1.4 Contributions

This research aims to enhance fraud detection in e-commerce by solving some key challenges using machine learning techniques. A major focus is addressing class imbalance, which can significantly affect model performance. To this end, this study compares different resampling methods. SMOTE, an advanced resampling method, generates synthetic examples of fraudulent transactions (minority class) to balance the dataset. In addition, the study examines basic resampling methods such as random oversampling, which duplicates existing minority class samples, and random under-sampling, which reduces the number of majority class instances. By applying and comparing these techniques, the research determines how different resampling strategies influence the model's ability to detect rare fraud cases in highly imbalanced datasets.

Another key contribution of this research is using advanced ensemble models specifically tailored for fraud detection. Unlike traditional models that treat all errors equally, these ensemble approaches can better handle varying complexities of fraudulent and legitimate transactions. By focusing on reducing false negatives (missing fraud) while maintaining accuracy for legitimate transactions, these models improve fraud detection overall performance. Furthermore, they help reduce false positives, lower manual review costs and ensure smoother transaction processing for customers.

As a further contribution, this research also focuses on the development of a Power BI dashboard to monitor fraud detection performance and emerging fraud patterns. The dashboard will provide an interactive and comprehensive view of key metrics, enabling real-time monitoring and decision-making. It will support e-commerce businesses in identifying potential fraud trends, tracking accuracy of fraud detection models and improving operational responses to fraud incidents, thereby enhancing overall fraud management in e-commerce.

1.5 Report Organization

This research is organized into several key chapters. **Chapter 2** presents a literature review of existing studies related to fraud detection in e-commerce and credit card transactions. **Chapter 3** describes the system methodology, outlining the overall approach and framework adopted for the project. **Chapter 4** details the system design, including system block diagram and dashboard wireframes. **Chapter 5** focuses on system implementation, covering the software setup, model training and integration, and dashboard development. **Chapter 6** presents the system evaluation and discussion, analysing the test results, dashboard testing and insights derived from the dashboard. Finally, **Chapter 7** concludes the study by summarizing key findings and providing recommendations for future work.

CHAPTER 2

Literature Reviews

Researchers have explored various datasets, machine learning algorithms and evaluation metrics to tackle challenges such as data imbalance [4,9,11-16], cost-sensitive problem [9] and changing fraud patterns and tactics [13,17,18]. This section reviews recent studies addressing these challenges, focusing on the datasets, preprocessing techniques, feature engineering, modelling methods, evaluation metrics and potential areas of study.

2.1 Previous works on Fraud Detection

2.1.1 Dataset

The most commonly used dataset for fraud detection studies is the **European Credit Card Fraud Detection dataset** from Kaggle [9,11,14,16,18,19]. Due to confidentiality issues, the dataset does not disclose detailed variable names. Instead, it includes anonymized features such as transaction amount, time, and class which indicate whether the transaction is fraudulent or not [9,11,18,19], as shown in *Table 2.1*. Despite its limitations, this dataset is widely adopted because of its relevance to real-world scenarios.

Features	Data types	Description
Time	Integer	Time difference between each transaction (second)
V1	Double	1 principle component
V2	Double	2 principle component
…	…	…
…	…	…
V28	Double	28 principle component
Amount	Double	Transaction amount
Class	Integer	1=fraud, 0=not fraud

Table 2.1.1: Attributes of European Dataset [16].

Other studies have proposed **e-commerce** or **shopping activity datasets**, which often provide richer feature sets [3,12,15,17,20]. For example, Najem and Kadhem [15] and Tejasri et al. [20] used similar datasets, which include user id, device id, IP address, source, browser, age, gender,

sign up time, purchase time, purchase value as shown in *Table 2.2*. Gölyeri et al. [3] used a dataset containing attributes such as total amount, order item count, successful orders, failed orders, last 24-hour return orders, last week return orders, and payment method as shown in *Table 2.3*. In another study, Kırelli et al. [15] used a dataset with selected features from 38 initial attributes. Key features included shopping amount, order hour, order day, name length, city, gender, age, category, brand, shipped amount, coupon discount, email confirmation time and label for fraud. These features are particularly valuable as they capture user behaviour patterns that can significantly enhance fraud detection in e-commerce contexts.

variables	Type
user_id	Number
signup_time	Date Time
purchase_time	
purchase_value	Number
device_id	Categorical
source	
browser	
sex	
age	Number
ip_address	Categorical
class	Number

Table 2.1.2: Attributes of E-commerce site data [15].

Feature	Feature Name	Feature Description
Feature 1	TotalAmount	basket amount
Feature 2	OrderItemCount	number of items in the basket
Feature 3	SuccessOrder	number of successful orders in the last 24 hours
Feature 4	FailedOrder	number of failed orders in the last 24 hours
Feature 5	Last24HoursReturnOrder	number of returns in the last 24 hours
Feature 6	LastWeekReturnOrder	number of returns in the last week
Feature 7	OrderID	order ID
Feature 8	PaymentMethodCode	payment method

Table 2.1.3: Attributes of E-commerce site for Boyner Group [3].

Other studies do not clearly specify the datasets used [4,8], making it challenging to evaluate the generalizability and applicability of their findings.

2.1.2 Data Preprocessing

Gölyeri et al. [3] and Ray [8] used **normalization** for feature scaling to ensure all features contributed equally to the model, while other studies applied **standardization** to scale features like transaction amounts effectively [9,15]. Adepoju et al. [13] employed the **conversion of categorical data** into **binary** format as a preprocessing step to handle categorical variables.

To handle imbalanced datasets, many studies proposed **SMOTE** to generate synthetic samples for the minority class [8-9,11-12,14-16,18]. Dornadula and Geetha [11] showed that SMOTE improved the performance of models like logistic regression, random forests and decision trees in precision and MCC as shown in *Figure 2.1*. Ray [8], Saputra and Suharjito [12] also observed that SMOTE effectively boosts the performance of models like neural networks, decision trees, random forests, and Naive Bayes by improving the classification of imbalanced data, especially for F1-score as shown in *Figure 2.2*. Abdulghani et al. [18] reported that after balancing the dataset using SMOTE, all models showed strong performance, with F1-Score exceeding 90% as shown in *Table 2.4*. Before applying SMOTE, the F1-Scores were significantly lower: Logistic Regression scored 81.68%, XGBoost 89.49%, and both LDA and Naïve Bayes only around 10% [18].

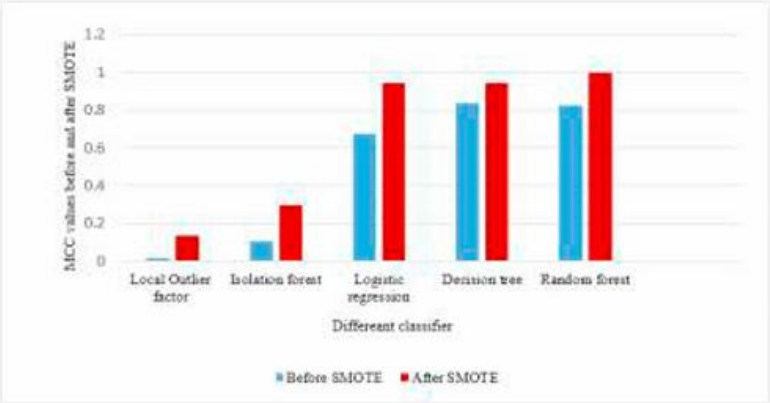


Figure 2.1.1: Comparison of MCC before and after applying SMOTE [11].

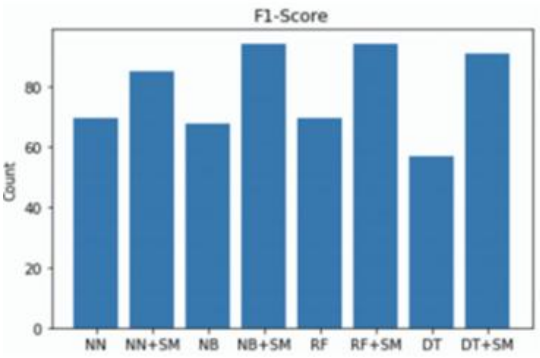


Figure 2.1.2: Comparison of F1-Score before and after applying SMOTE [8].

Classifier	accuracy %	precision %	Recall%	F1%	AUC%
LR	94.752	97.411	91.965	94.61	94.756
LDA	91.737	98.467	84.822	91.137	91.749
NB	91.338	97.158	85.198	90.786	91.349
XGBoost	99.969	99.938	100	99.969	99.969

Table 2.1.4: Performance comparison after applying SMOTE [18].

2.1.3 Feature Engineering

Most studies in fraud detection rely on **Principal Component Analysis (PCA)** for feature engineering. PCA is widely used to reduce the dimensionality of dataset while simplify data by highlighting the most important information [12]. Najem and Kdhem [15] highlighted PCA's effectiveness in improving machine learning performance by reducing the complexity of high-dimensional data. While for studies using the European Credit Card Fraud Detection dataset, the data has already been processed using PCA, except for the Amount, Time, and Class features, which are kept as is due to confidentiality. These features are retained in their original form while other variables have undergone dimensionality reduction through PCA [9,11,16,18,19].

2.1.4 Modelling

Several machine learning algorithms have been extensively studied for fraud detection in e-commerce and credit card transactions. Researchers have explored diverse models, ranging from basic classifiers to ensemble techniques and neural network-based approaches.

Ray [8] proposed four machine learning models, which are Decision Tree, Naïve Bayes, Random Forest and Neural Network. Among these models, the **neural network without SMOTE** achieved the **highest accuracy** of 96%. While for other metrics, the results were quite low. **Neural network with SMOTE** achieved superior performance compared to other models, particularly in terms of **recall** and **G-mean** as shown in *Figure 2.3* and *Figure 2.4*. This suggest that accuracy alone is not a reliable metric in imbalanced datasets. Using SMOTE can significantly improve the model's ability to detect fraud by better handling class imbalance. Naïve Bayes with SMOTE achieved the highest F1-score, showing the strong balance between precision and recall. However, its recall was the lowest without SMOTE and even with SMOTE, it remained the second lowest among all models. This means that Naïve Bayes miss many actual fraud cases, which is risky in real-world applications.

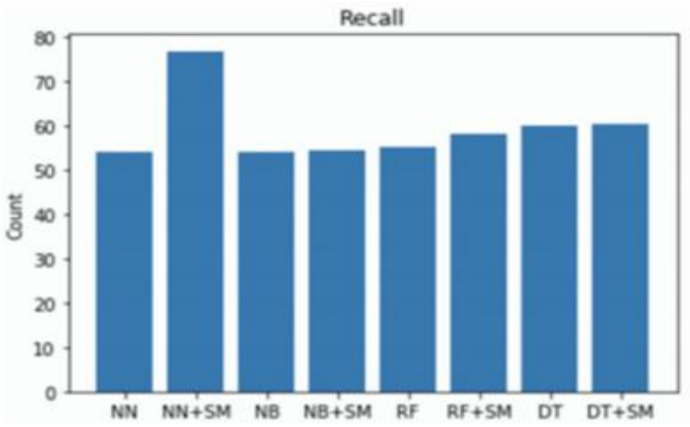


Figure 2.1.3: Comparison of Recall before and after applying SMOTE [8].

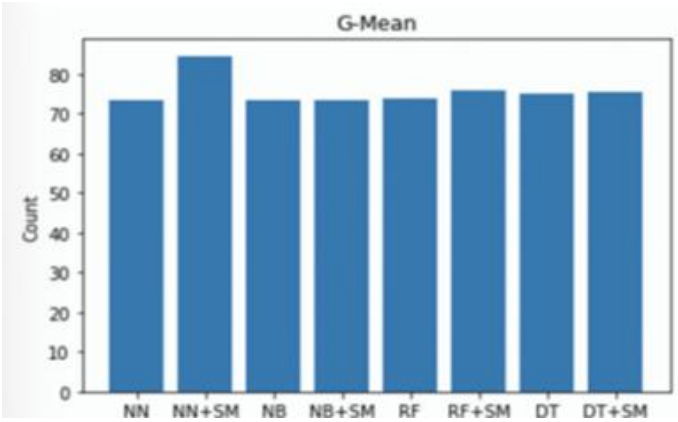


Figure 2.1.4: Comparison of G-Mean before and after applying SMOTE [8].

Najem and Kadhem [15] applied ensemble models, including LightGBM, XGBoost and Random Forest. After applying standardization and PCA, XGBoost and LightGBM achieved the highest performance with perfect accuracy (100%), outperforming Random Forest with a slightly lower accuracy of 99%. Using additional metrics like precision, recall, F1-score and AUC, the results for XGBoost and LightGBM remain nearly perfect or perfect, confirming their superior performance in fraud detection.

In Puh and Brkić [9] research, Random Forest outperformed SVM and Logistic Regression, achieving the highest AUC and AP scores of 0.9448 and 0.8483 respectively. SVM recorded the lowest AUC score (0.8877), showing its weaker capability in separating classes, while Logistic Regression had the lowest AP score (0.7337), reflecting more false positives and less reliability when detecting fraud at higher recall levels.

Dornadula and Geetha [11] observed that Random Forest, Decision Tree and Logistic regression performed better than Isolation Forest and Local Outlier Factor in detecting credit card fraud. Among these, Random Forest achieve the highest accuracy, precision and MCC consistently as shown in *Table 2.5*. Among these, Random Forest achieve the highest accuracy of 99.98%, precision of 99.96% and MCC of 0.9996 as shown in Table 2.5. After applying SMOTE, Random Forest's results became nearly perfect, showing its strong performance in handling minority class detection.

Methods	Accuracy	Precision	MCC
Local Outlier factor	0.4582	0.2941	0.1376
Isolation forest	0.5883	0.9447	0.2961
Logistic regression	0.9718	0.9831	0.9438
Decision tree	0.9708	0.9814	0.9420
Random forest	0.9998	0.9996	0.9996

Table 2.1.5: Performance comparison of LOF, iForest, LR, DT and RF [11].

Sadeneni [4] compared five models, including ANN, Random Forest, Decision Tree, Logistic Regression and SVM based on accuracy, precision, and false alarm rate as shown in *Table 2.6*. ANN achieved the highest accuracy of 99.92%, precision of 99.57% and a very low false alarm rate with only 0.1%, suggesting that it is very effective at fraud detection. However, ANN comes with high training costs and hardware dependency, making it less practical for all business. Random Forest is a strong alternative, as it achieved a very high accuracy of 99.21% and precision of 92.34%. SVM had the highest false alarm rate at 4.9%, suggesting that it is less ideal for fraud detection where minimising false alarms is important. Decision Tree performed slightly better than Logistic Regression across all metrics.

ML Technique	Accuracy	Precision	False Alarm Rate
Random Forest	99.21%	92.34%	3.8%
Decision Tree	98.47%	84.98%	2.0%
Logistic Regression	95.55%	83.76%	2.7%
Support Vector Machine	95.16%	88.42%	4.9%
Artificial Neural Networks	99.92%	99.57%	0.1%

Table 2.1.6: Performance comparison of RF, DT, LR, SVM and ANN [4].

Gölyeri et al. [3] demonstrated including the IsGuestOrder feature significantly improved fraud detection. Before including IsGuestOrder feature, XGBoost initially outperformed other models with 0.90 accuracy and 0.92 recall, while Logistic Regression showed superior precision (0.91) despite its lower recall (0.86) as shown in *Table 2.7*. After including the feature, Logistic Regression achieved the highest accuracy at 0.93 and F1-score at 0.92 as shown in *Table 2.8*. This showed that a good feature can boost a simple model's performance when it

captures the fraud pattern, while XGBoost already handled complex patterns, so a single binary feature does not significantly change its performance.

Classifier	Accuracy	Precision	Recall	F-measure
Decision Tree	0.83	0.82	0.78	0.80
Logistic Regression	0.89	0.91	0.86	0.88
Extreme Gradient Boosting	0.90	0.85	0.92	0.88
Random Forest	0.81	0.80	0.87	0.83

Table 2.1.7: Model performance before including the *IsGuestOrder* feature [3].

Classifier	Accuracy	Precision	Recall	F-measure
Decision Tree	0.83	0.82	0.79	0.80
Logistic Regression	0.93	0.95	0.89	0.92
Extreme Gradient Boosting	0.90	0.87	0.91	0.89
Random Forest	0.86	0.81	0.90	0.85

Table 2.1.8: Model performance after including the *IsGuestOrder* feature [3].

Sailusha et al. [19] focused on comparing Random Forest and AdaBoost in credit card fraud detection. While both models had the same high accuracy, Random Forest outperformed AdaBoost in precision, recall and F1-score as shown in *Figure 2.5*, making it more reliable for fraud detection. Random Forest detected more fraud cases with a recall over 70%, while for AdaBoost with recall over 60%. Both models showed the signs of overfitting as the training data was significantly better than that of test data. The AUC of AdaBoost was better, suggesting it might be more effective in distinguishing between fraud and non-fraud cases among different thresholds.

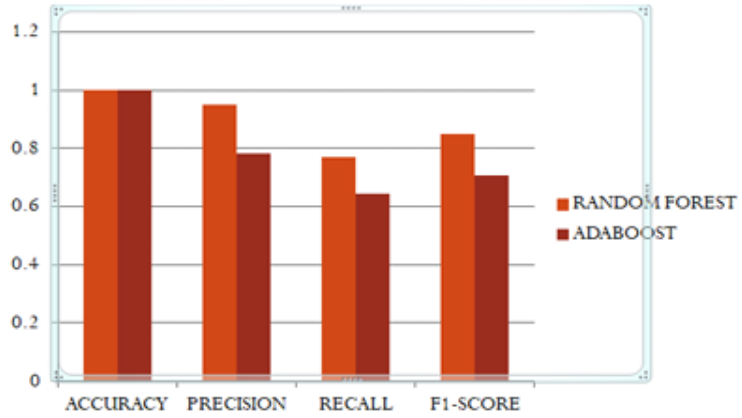


Figure 2.1.5: Performance comparison of Random Forest and AdaBoost [19].

Based on the study of Adepoju et al. [13], Logistic Regression emerged as the best-performing model for credit card fraud detection, with the highest accuracy of 99.074%. It also achieved the perfect sensitivity (recall) at 100%, meaning it was able to detect all actual fraud cases in the test data. SVM also performed well with an accuracy of 97.53% and a recall of 97.56%. KNN followed closely with 96.91% accuracy, however, its recall was only 89.36%, meaning that it missed a portion of fraud cases. Naïve Bayes performed poorly despite having a perfect specificity and precision of 100%. However, its recall was 0%, meaning it failed to identity any of the actual fraud cases. This outcome is concerning because if a system cannot detect fraud, it is useless, even it works well for normal transactions.

Ileberi et al. [14] proposed combining models like Decision Tree, Random Forest, Extra Tree, XGBoost and Logistic Regression with AdaBoost. Without AdaBoost, Random Forest performed the best with an MCC of 0.88 and an accuracy of 99.95% as shown in *Table 2.9*. When combining with AdaBoost, all models showed significant improvement, with all metrics above 90% as shown in *Table 2.10*. The recall for Extra Tree and XGBoost increased a lot from 78.19% and 59.39% to 99.96% and 99.97% respectively, making them became the best-performing model, with nearly perfect scores across all metrics after combining with AdaBoost. Among all the models, Logistic Regression performed the worst. This is due to its linear nature, which limits its ability to capture complex and non-linear relationships in the data. While ensemble methods like Random Forest, XGBoost and Extra Tree show superior performance, especially when combined with AdaBoost, because they effectively capture complex, non-linear relationships and reduce errors through boosting.

Model	AC	RC	PR	MCC
DT	99.91%	75.57%	79.83%	0.78
RF	99.95%	79.38%	97.19%	0.88
ET	99.95%	78.19%	96.29%	0.86
XGB	99.90%	59.39%	84.04%	0.71
LR	99.90%	56.55%	85.18%	0.59

Table 2.1.9: Performance comparison of base models [14].

Model	AC	RC	PR	MCC
DT	99.67%	99.00%	98.79%	0.98
RF	99.95%	99.77%	99.91%	0.99
ET	99.98%	99.96%	99.93%	0.99
XGB	99.98%	99.97%	99.92%	0.99
LR	98.75%	93.83%	97.56 %	0.94

Table 2.1.10: Performance comparison of models combined with AdaBoost [14].

Abdulghani et al. [18] focused on machine learning algorithms like Logistic Regression, LDA, Naïve Bayes and XGBoost. Among the models, XGBoost was the most effective one, achieving the highest accuracy of 99.969%, precision of 99.938%, recall of 100%, F1-score of 99.969% and AUC of 99.969%. This best performance highlights XGBoost's robustness in handling large datasets and detecting fraud. In contrast, Logistic regression, LDA and Naïve Bayes showed lower accuracy with 94.752%, 91.737% and 91.338% respectively.

Mohbey et al. [16] compared the performance of Naïve Bayes, SVM, Logistic, Random Forest, Decision Tree and XGBoost for credit card fraud detection as shown in *Table 2.11*. Among these, XGBoost showed the highest accuracy at 96.44%, significantly outperforming the others. For instance, Logistic Regression achieved an accuracy of 94.43%, while SVM, Random Forest and Decision Tree are below 94%. The precision, recall, F1-score and AUC values also favoured XGBoost, indicating its robustness in handling imbalanced datasets and complex transaction patterns. Naïve Bayes performed the worst, with an accuracy of 89.34% and F1-score of 89%, which aligns with previous studies. This study highlights the effectiveness of XGBoost as an ensemble model in improving performance, especially in scenarios involving imbalanced datasets.

	Accuracy	Precision	Recall	F1-Score
Naive Bayes	0.8934	0.90	0.90	0.89
SVM	0.9390	0.94	0.94	0.94
Logistic Regression	0.9443	0.94	0.95	0.94
RandomForest	0.9341	0.93	0.94	0.94
DecisionTree	0.9340	0.92	0.92	0.92
Xgboost	0.9644	0.96	0.97	0.96

Table 2.1.11: Performance comparison of NB, SVM, LR, RF, DT and XGboost [16].

Based on the extensive research, the most effective models for fraud detection in e-commerce and credit card transactions are **Random Forest** and **XGBoost**, which are widely used and consistently achieve high scores across accuracy, precision, recall, F1-score and AUC. ANN also performs well but are resource intensive. Logistic Regression and Decision Trees offer good performance with simplicity, especially when enhanced with strong features or boosting techniques, although they are generally outperformed by ensemble models. In contrast, Naïve Bayes and SVM often perform poorly. Naïve Bayes has trouble in detecting fraud because its recall is usually low, while SVM tends to produce more false alarms and lower AUC scores. Overall, ensemble models are the most reliable and widely used, while simpler models require enhancements to be competitive.

2.1.5 Evaluation Metrics

Accuracy is the most widely used metrics and often the primary parameter in many studies for evaluating model performance. It is commonly considered as the base measure, but it is not always a good metric, especially for imbalanced dataset [11,12]. Dornadula and Geetha S [11] proposed the use of **Matthews Correlation Coefficient (MCC)** as more reliable measure for evaluating binary (two-class) classifiers. The MCC considers all true and false values, making it a balanced metric that works well even when the dataset contains imbalanced classes [11]. This is why MCC is often preferred in such scenarios, as it provides a more comprehensive evaluation of model performance.

In addition to MCC, metrics such as **recall, precision,** and **F1-score** are commonly used alongside accuracy. **G-mean** is another metric that measures a model's overall performance by assessing its ability to correctly classify both majority and minority classes [8,12]. The **F1-score** is particularly valuable in imbalanced datasets, as it balances precision and recall evaluating the classification of the minority class effectively [8,12].

Another important evaluation metric is **Area Under the Curve (AUC)**, often used with the **Receiver Operating Characteristic (ROC) curve**. However, in imbalanced datasets, where the number of true negatives greatly exceeds true positives, the ROC curve may not be the most appropriate metric. Puh and Brkic [9] proposed using **Precision-Recall curves** instead, as they

focus on precision, which compares false positives to true positives, making them less sensitive to class imbalance.

Among these studies, only Md. Nur-E-Arefin [15] used **Mean Absolute Error (MAE)** and **Root Mean Squared Error (RMSE)** as evaluation metrics. However, these metrics may not be suitable for this research as they are more commonly applied to regression tasks rather than classification problems.

2.2 Literature Matrix Table

Author/year	Task/Title	Problem Mentioned	Dataset	Pre-processing techniques	Feature Engineering	Method for Modeling	Evaluation Metrics	Future study/ Conclusion
Murat Gölyeri , Sedat Çelik, Fatma Bozyiğit, Deniz Kılınç, 2023 [3]	Fraud Detection on E-commerce Transactions Using Machine Learning Techniques	*Not mentioned in the research paper.*	Shopping activities during ninety days on the e-commerce website and mobile application of Boyner Group (total amount, order item count, success order, failed order, last 24 hour return order, last week return order, payment method)	SimpleImputer and StandardScaler classes from the scikit-learn library, ChiSquare feature selection, 10-fold cross validation	*Not mentioned in the research paper.*	Decision tree, Logistic regression, Random Forest, XGBoost	Accuracy, Precision, Recall, F1-score	- Model performance improved with the inclusion of the *IsGuestOrder* feature. - Logistic regression achieved over 92% accuracy, making the findings promising for future research. - Future work include developing classification software for the company.
Praveen Kumar Sadineni, 2020 [4]	Detection of Fraudulent Transactions in Credit Card using Machine Learning Algorithms	- Imbalanced data	150000 transactions data from Kaggle (time of transcation, amount, class)	*Not mentioned in the research paper.*	PCA	Random Forest, Decision Tree, Logistic Regression, SVM, ANN	Accuracy, Precision, False Alarm rate	- ANN achieves the highest accuracy (99.92%) and precision (99.57%) and the lowest false alarm rate (0.1%)
Samrat Ray, 2022 [8]	Fraud Detection in E-Commerce Using Machine Learning	- Datasets with extremely small class proportions result in	Online business fraud dataset.	SMOTE, Normalization, Scale of characteristics,	PCA	Decision Tree, Naïve Bayes, Random	Confusion Matrix, Accuracy, Recall,	- Leveraging advanced computations or deep learning can improve the detection of e-

		biased or unbalanced information.		Feature extraction		Forest, Neural Network	Precision, F1-score, G-mean	commerce fraud and boost neural network performance with the SMOTE technique.
Maja Puh, Ljiljana Brkic, 2019 [9]	Detecting Credit Card Fraud Using Selected Machine Learning Algorithms	-Data deficiency -Imbalanced data -Cost sensitive problem -Behavioral variation	Transaction made in September 2013 by European cardholders.	SMOTE, feature scaling for amount using standardization, data split (70:30)	PCA	Random Forest, SVM, Logistic regression	Area Under ROC Curve (AUC), Average precision (AUPRC)	- SVM has slightly lower results than other two in AUC and Recall scores - Models with incremental learning have better results. - Future work includes exploring incremental learning on a more realistic dataset.
Vaishnavi Nath Dornadula, Geetha S, 2019 [11]	Credit Card Fraud Detection using chine arning Algorithms	- Dataset is highly imbalanced	European Credit card transaction dataset (transaction id, cardholder id, amount, time, label)	SMOTE	PCA	Local Outlier factor, Isolation Forest, SVM, Logistic regression, Decision tree, Random Forest	Accuracy, precision, MCC	- Random Forest performed the best among the models with accuracy (0.9998), precision (0.9996), MCC (0.9996) - MCC is the better metric for evaluating imbalance dataset. - By applying the SMOTE, the models perform better than before. - LR, DT and RF achieved better results.
Adi Saputra, Suharjito, 2019 [12]	Fraud Detection using Machine Learning in	- Imbalanced data	E-commerce fraud dataset sourced from	SMOTE, feature extraction,	PCA	Decision tree, Naïve Bayes, Random	Accuracy, precision, recall, G-mean, F1	- The results showed NN has the highest accuracy with 96%,

	e-Commerce		Kaggle	transformation, normalization		Forest, and Neural network	Score	then NB and Random Forest are 95%, DT accuracy is 91%. - Using SMOTE on NN, RF, DT, and NB was able to handle dataset imbalance by producing higher G-Mean and F-1 scores. - Future work is to use other algorithms or deep learning for fraud detection in e-commerce. - Improve neural network accuracy using SMOTE.
Olawale Adepoju, Julius Wosowei, Shiwani lawte, Hemaint Jaiman, 2019 [13]	Comparative Evaluation of Credit Card Fraud Detection Using Machine Learning Techniques	- Dynamic fraudulent behavior patterns make detection more challenging. - Datasets are often limited and imbalanced. - Model performance relies heavily on testing and feature selection. - Evolving data can lead to reversed or outdated classifications over time.	Card transaction dataset (average daily transaction amount, transaction amount, transaction declined, foreign transaction, high risk transaction, six-month average balance)	Binary encoding, Data split 80:20	*Not mentioned in the research paper.*	Logistic Regression, KNN, Naïve Bayes, SVM	Accuracy, Sensitivity, Specificity (Recall), Precision	- LR was the most accurate in detecting credit card fraud, with accuracy 99.074. - Using a larger dataset with more fraudulent cases is recommended. - Other resampling strategies, cost-sensitive learning methods, and ensemble learning methods could be explored in future to better handle a skewed dataset.

Emmanuel Ileberi, Yanxia Sun, Zenghui Wang, 2021 [14]	Performance Evaluation of Machine Learning Methods for Credit Card Fraud Detection Using SMOTE and AdaBoost	Imbalanced data	European credit card dataset.	SMOTE	*Not mentioned in the research paper.*	SVM, Random Forest, Extra Tree, XGBoost, Logistic Regression, Decision tree, ADABoost	Accuracy, recall, precision, MCC, AUC	- DT-AdaBoost, RF-AdaBoost, ET-AdaBoost,and XGB-AdaBoost achieved accuracies of 99.67%, 99.95%, 99.98%, and 99.98%, respectively. - The results confirm that AdaBoost significantly enhances the performance of machine learning models. - Future work will focus on testing and validating the framework using real credit card fraud datasets from financial institutions.
Suha M. Najem, Suhad M. Kadhem, 2021 [15]	An Efficient Feature Engineering Method for Fraud Detection in E-commerce	- Imbalance in Datasets	Clothing sales transaction dataset (device-id, IP address, source, browser, age, country, sex, signup-time, purchase-time, purchase-value)	SMOTE, standardization for feature scaling	PCA	LightGBM, XGboost, Random Forest	Accuracy, Precision, Recall, F1-score, AUC-ROC	- LightGBM and XGboost achieved the best accuracy after preprocessing the dataset. - Future work is to use larger dataset with new feature engineering
Krishna Kumar Mohbey, Mohammad	Credit Card Fraud Prediction Using XGBoost: An ensemble	Data imbalance.	European credit card dataset.	Standardization, normalization, data split 70:30	PCA	Naïve Bayes, SVM, Random Forest,	Precision, recall, accuracy, AUC, f-measure	XGBoost performed better.

Zubair Khan, Ajay Indian, 2022 [16]	Learning Approach					Logistic Regression, XGBoost		Hybrid models can be built to improve the research.
Yasin Kırelli, Seher Arslankaya, Muhammed Taha Zeren, 2020 [17]	Detection of Credit Card Fraud in E-Commerce Using Data Mining	Fraudsters are changing their strategies and new fraud patterns are emerging as now.	E-commerce dataset (shopping amount, order hour, order day, name length, city, gender, age, category, brand, shipment amount, discount, isFraud)	Gain Ratio, Info Gain and Chi-Squared (feature selection), data split 70:30	*Not mentioned in the research paper.*	Naive Bayesian, Naive Bayes Tree, Decision Tree J48, KNN, ANN, RBF Network	TP rate, FP rate, Precision, Recall, F-measure, ROC Area	KNN achieved the highest Precision (0.956), Recall (0.959) and F-measure (0.955) among the models. Naïve Bayesian and NBTree perform better in ROC Area with 0.963.
Ahmed Qasim Abdulghani, Osman Nuri UCAN, Khattab M. Ali Alheeti, 2021 [18]	Credit Card Fraud Detection Using XGBoost Algorithm	Significant changes in fraud methods and ever-changing strategies.	European credit card dataset.	SMOTE	PCA	Logistic Regression, LDA, Naïve Bayes, XGBoost	Accuracy, precision, recall, F1-score, AUC, confusion matrix	- XGBoost performs the best. - Performance is good after balancing dataset.
Ruttala Sailusha, V. Gnaneswar, R. Ramesh, G. Ramakoteswara Rao, 2020 [19]	Credit Card Fraud Detection Using Machine Learning	Data mining techniques are used, but the results are not very accurate in detecting credit card fraud.	European credit card dataset.	*Not mentioned in the research paper.*	PCA	Random Forest, AdaBoost	Accuracy, precision, recall, F1-score	- Accuracy is the same for both the Random Forest and the Adaboost algorithms. - Precision, recall, and the F1-score the Random Forest has the highest value than the Adaboost. - Future work is to implement deep learning algorithms to detect credit card fraud accurately

C. Tejasri, CH Sai Ushanth Aryan, D. Deekshith, Arrolla Chintu, Dr. T. Subba Reddy, 2022 [20]	Fraud Detection in E-commerce using Machine Learning	Restricted to identifying the features that will be used to classify transactions as either fraudulent or non-fraudulent.	E-commerce fraud dataset from Kaggle (user id, device id, gender, age, source browser, purchase time, sign up time, purchase value, ip address, label)	Feature extraction, transformation, normalization	*Not mentioned in the research paper.*	Random Forest, Decision Tree,	Accuracy	Random Forest algorithm can achieve higher accuracy in fraud detection.

2.3 Limitation of previous Studies

Many studies using the European Credit Card Fraud Detection **dataset lack detailed information about the data**. The anonymizes 28 variables (v1, v2…v28), and only disclose three features: time, amount and class [9,11,16,22,23]. Confidentiality issues often prevent researchers from revealing variable names or detailing the original and engineered features. Furthermore, some studies lack clear descriptions of the dataset or its features. This anonymization and lack of transparency make it difficult to interpret and evaluate the importance of individual features in fraud detection. Consequently, it becomes challenging to identify key features driving fraud detection and to understand how these findings apply to real-world scenarios, especially in terms of their impact on model performance and detection accuracy.

Although class imbalance is a common issue in fraud detection, **some studies do not apply any resampling techniques** despite working with heavily imbalanced datasets. This oversight can lead to biased models favour the majority class, reducing the effectiveness of fraud detection. While many studies used SMOTE and reported improved results, there is **limited exploration of alternative resampling techniques** such as Random Oversampling and Random Under-sampling. Most studies also lack comprehensive comparisons between different resampling methods, leaving a gap in understanding which techniques are most suitable across various datasets and models.

Furthermore, **existing studies have limited focus on developing interactive dashboards** to visualize and monitor the real-time performance of fraud detection models and the evolving fraud patterns**.** The lack of such dashboards makes it difficult for e-commerce businesses to track important metrics and ensures that the system remains reliable. This gap limits the ability to identify performance drops quickly, observe the fraud cases and trends in real-time, and adjust the model accordingly, which ultimately impacts effective fraud detection and decision-making.

2.4 Proposed Solutions

This project proposes that the **use of dataset with clear, well-documented features** is essential to ensure transparency and interpretability in fraud detection. This project will focus on utilizing datasets that provide variables with explicit feature names, which will enable a clearer understanding of the important factors influencing fraud detection. For example, publicly available datasets that **contain transactional or demographic features**, such as age, transaction amount, transaction time, product category and job, these data provide richer context and allow for a more comprehensive analysis. Using datasets with detailed information will not only enhance model interpretability but also improve the ability to reproduce findings and apply them to real-world scenarios.

To address the limitation of inadequate handling of imbalanced datasets, this project will **implement and compare multiple resampling methods**, including **SMOTE, Random Oversampling** and **Random Under-sampling**. These methods are easy to implement and offer a balanced trade-off between effectiveness and computational efficiency. These methods will be applied across different models to determine their individual and comparative impacts on fraud detection performance. Performance will be assessed before and after applying resampling methods to ensure that improvements are consistent and significant. By conducting systematic analysis using metrics like F1-Score, Precision, Recall and MCC, the study aims to identify the most effective resampling method for improving minority class detection.

Another proposed solution is the **development of an interactive dashboard** that can track and visualize the real-time fraud detection models performance and fraud patterns. The dashboard would provide continuous monitoring of confusion matrix and key metrics such as accuracy, precision, recall, MCC, F1-score and AUC. Additionally, it would allow businesses to monitor specific factors contributing to fraud, such as age, gender, transaction amount, transaction hour and product category, which may influence model performance. This real-time monitoring will enable stakeholders to identify performance drops promptly and make necessary adjustments to improve detection accuracy. By integrating these features, the dashboard would enhance decision-making and optimize fraud management, offering a practical tool for e-commerce businesses.

CHAPTER 3

System Methodology/Approach

The chapter outlines the system requirements, design, use case and project timeline. It details the hardware and software specifications, the overall system design for data preprocessing, model training and deployment. The use case diagram illustrates this end-to-end flow from data input to visual reporting.

3.1 System Requirement

3.1.1 Hardware

The hardware used in this project includes a personal laptop. The laptop is essential for performing machine learning tasks such as data preprocessing, model training and result analysis. A capable processor, sufficient RAM, and storage are required to handle large datasets and computational workloads efficiently. Additionally, the same system is used for developing and testing Power BI dashboard, which will visualise the performance of the machine learning models.

Description	Specifications
Model	Inspiron 15 3511
Processor	11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz
Operating System	Windows 11
Graphic	Intel® Iris® Xe Graphics
Memory	8GB DDR4 RAM
Storage	512GB NVMe Micron SSD

Table 3.1.1: Specifications of laptop

3.1.2 Software/Tools

The software and libraries used in this project are important for implementing the machine learning tasks and generating Power BI dashboard. The following software are required for different tasks such as data preprocessing, model training and visualization.

Description	Specifications
Development Environment	Jupyter Notebook, Google Colab
Programming Language	Python
Machine Learning Libraries	Pandas, Numpy, Scikit-learn, Matplotlib, Seaborn, XGBoost, Imbalanced-learn, Joblib
Dashboard	Power BI Desktop, Power BI Service

Table 3.1.2: Specifications of software

The primary development environment for the project is **Jupyter Notebook**, a platform that supports interactive and iterative coding. **Google Colab** is also used to Jupyter Notebook, especially when more advanced or larger visualizations are required, as it can handle complex computations more efficiently.

Python is the main programming language, including rich libraries like **pandas** and **NumPy** used for data manipulation, cleaning and preprocessing. Data visualization and EDA are conducted using **matplotlib** and **seaborn**, which provide insightful plots and visualizations, such as heatmaps, boxplots and various charts.

For data preprocessing, **scikit-learn** is used for encoding categorical features and splitting datasets. To address class imbalances, the **imbalanced-learn** library is used to facilitate synthetic data generation using SMOTE as well as Oversampling and Under-sampling.

During modelling phase, ensemble learning algorithms like Random Forest, AdaBoost and XGBoost are implemented using **scikit-learn** and **XGBoost**. Model evaluation is also conducted with **scikit-learn** to compute metrics such as accuracy, precision, recall, F1-score, MCC, AUC, classification report and confusion matrix. Additionally, visualizations of performance metrics, including confusion matrices, are generated using **matplotlib** and **seaborn**.

For model deployment, the **joblib** library is used to export and import the trained model, ensuring portability and ease of integration. The deployed model is integrated into **Power BI** using **Python scripts** to enable fraud detection and visualization. Once the dashboard is finalized, it is **published to the Power BI Service**, allowing online access. Finally, testing is conducted to validate functionality, performance, and usability, leveraging both manual testing processes and Python utilities for monitoring execution and responsiveness.

3.2 System Design

Figure 3.2.1 illustrates the steps of an e-commerce fraud detection system using machine learning. The process includes data collection, EDA, preprocessing and resampling to handle class imbalance. The data is splitting into training and testing sets and models like Random Forest, XGBoost and AdaBoost are trained and fine-tuned. The best model is evaluated and deployed, with predictions monitored in real time through a dashboard.

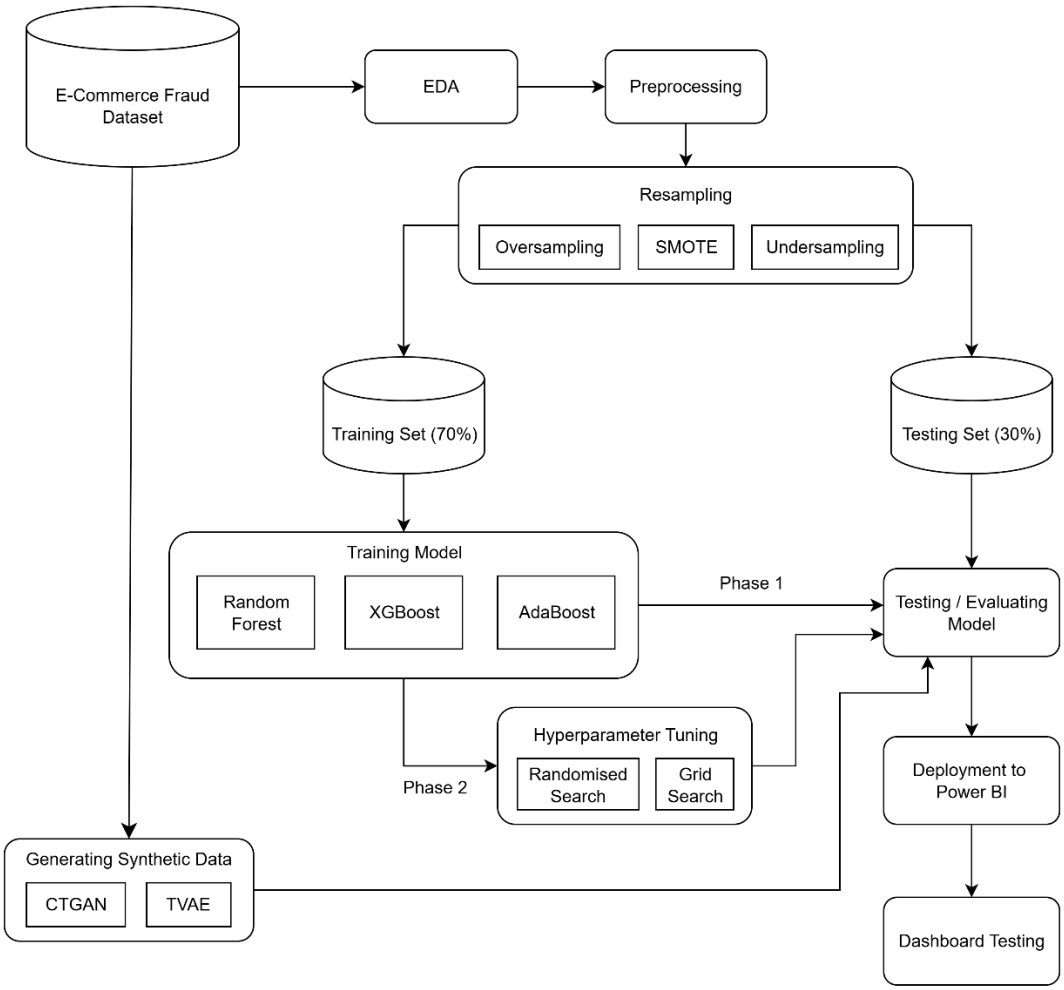


Figure 3.2.1: Project Workflow Overview

3.2.1 Dataset Collection

The first step involves gathering relevant and high-quality data to train and evaluate the model. For this project, the required data includes **transaction details** (e.g., transaction date and time, transaction amounts, product category, order quantity), **user information** (e.g., customer age, gender, location, job) and **labels** indicating whether each transaction is fraudulent or legitimate.

These data are sourced from open repositories like **Kaggle**, which offer well-organized datasets such as the Credit Card Fraud Detection Dataset and E-commerce Fraud Dataset. The source is chosen for its domain relevance, detailed documentation, and reliable data quality. To ensure the sufficiency of the dataset, **at least 50,000 transactions** will be targeted for analysis.

3.2.2 EDA & Data Preprocessing

Data preprocessing is a critical step in preparing the dataset for machine learning. It involves cleaning, transforming, and organizing the raw data into a suitable format for analysis.

The first task is **handling missing values**. It is important to check for missing values in the dataset, as they can cause data loss and bias. The *isnull().sum()* function helps to find any missing values, while the *info()* method gives an overview of the dataset, showing how many values are present in each column and also the number of null values.

Since duplicate records can distort analysis and impact model performance, if **duplicate** rows are found, they need to be **removed**. It is common to retain only the first occurrence of each record and remove subsequent duplicates.

Outliers are detected using the *sns.boxplot* function which generates boxplots that visually reveal any unusual data points. To **handle outliers**, if an outlier is associated with the target variable showing "fraud", it will be kept as it might represent a legitimate high-risk transaction. However, if the outlier is not associated with fraud, it will be removed from the dataset using the *drop()* function.

Correlation analysis is performed using the *corr()* function to calculate the relationships between numerical features and visualised with a heatmap. This helps identify strong correlations, detect multicollinearity, and reveal insights such as relationships between features and fraud risk.

Important **features are extracted** from existing fields to enrich the dataset. For example, the customers' age can be calculated from their date of birth, the hour and day of the week can be derived from the transaction timestamp. Additionally, geographical distance to the merchant can be calculated using location coordinates. These newly derived features help to improve the model's ability to recognize complex patterns in the data.

Categorical features are transformed using appropriate encoding techniques. **One-hot encoding** is applied to nominal categories, **binary encoding** is used for high-cardinality variables. **Target encoding** can be applied by extracting statistical information from the original features, such as calculating the fraud rate for each credit card number, then replacing the values with these aggregated metrics. This approach captures the likelihood of fraud associated with each card, enabling the model to learn which cards are more susceptible to fraudulent transactions. These methods convert non-numeric data into a format that can be effectively understood and processed by machine learning models.

In cases of imbalanced datasets, where fraudulent transactions are far less frequent than legitimate ones, techniques like **SMOTE, Oversampling** and **Under-sampling** are applied. SMOTE generates synthetic samples for the minority class to balance the dataset. Oversampling duplicates minority class data and under-sampling reduces majority class data.

The dataset is then split into **training and testing sets** using a 70:30 ratio, meaning 70% of the data is used for training model, and the remaining 30% is reserved for testing model. This ensures the model is evaluated on unseen data, which helps assess its generalization ability.

3.2.3 Model Selection

The Modelling phase involves the application of three ensemble machine learning algorithms: **Random Forest**, **XGBoost** and **AdaBoost**. Each algorithm has its unique strengths, and this section will detail how they are applied to the fraud detection problem in e-commerce transactions.

Random Forest

Random Forest is a robust ensemble learning algorithm widely used for classification tasks due to its simplicity and interpretability [9]. It builds multiple decision trees using a bagging (bootstrap aggregating) approach, where each tree is trained on random subsets of data and features. The randomness creates diversity among the trees, which helps minimise overfitting and boosts the model's generalization ability [3,9].

In e-commerce fraud detection, each tree is constructed using a random vector value with a consistent distribution across all trees, and a predefined maximum depth is set to control

complexity and prevent overfitting [12]. Once the trees are built, the final prediction is made through majority voting, where the class most frequently predicted by the ensemble of trees is selected as the model's output [4,9]. *Figure 3.2.2* demonstrates the structure of the Random Forest process, starting with a dataset divided into random subsets to train individual decision trees. Each tree predicts a class, and the majority vote determines the final class.

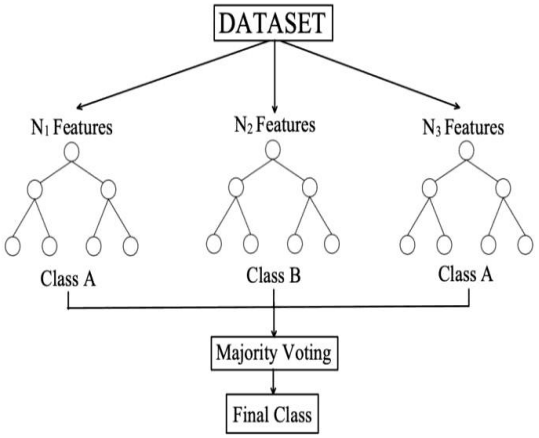


Figure 3.2.2: Workflow of Random Forest [21]

XGBoost

XGBoost (Extreme Gradient Boosting) is an advanced ensemble tree algorithm developed from Gradient Boosting Decision Trees (GBDT). It is especially well-suited in managing high-dimensional data and identifying complex, non-linear relationships between variables, making it highly effective for classification tasks [3].

In XGBoost, the training process starts by splitting the data and training the first decision tree. Each tree in the sequence is trained to correct the residual errors from the previous tree. The process is repeated, with each subsequent tree focusing on the misclassified instances from the prior tree. After all trees are trained, the results of all trees are combined by summing (or weighted summing) their predictions to produce the final result. *Figure 3.2.3* illustrates the workflow of XGBoost.

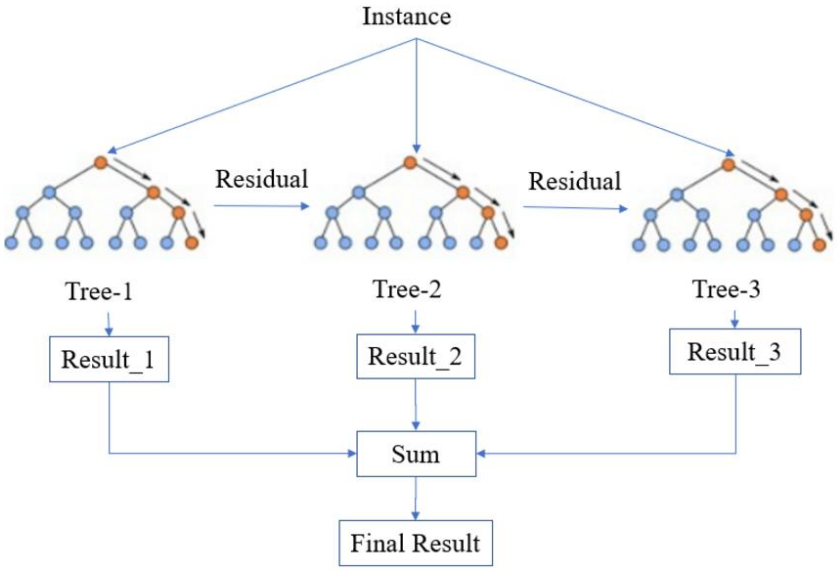


Figure 3.2.3: Workflow of XGBoost [22]

AdaBoost

AdaBoost (Adaptive Boosting) is an ensemble learning method that strengthens classification performance by sequentially combining multiple weak classifiers into a single robust model. *Figure 3.2.4* illustrates the workflow of the AdaBoost algorithm. It begins with training a weak learner, then iteratively adjusts the weights of misclassified instances to focus on harder-to-classify samples. Each weak learner contributes to the final model through a weighted sum based on its performance. This process continues until the specified number of iterations is reached or the dataset is accurately classified [4,19].

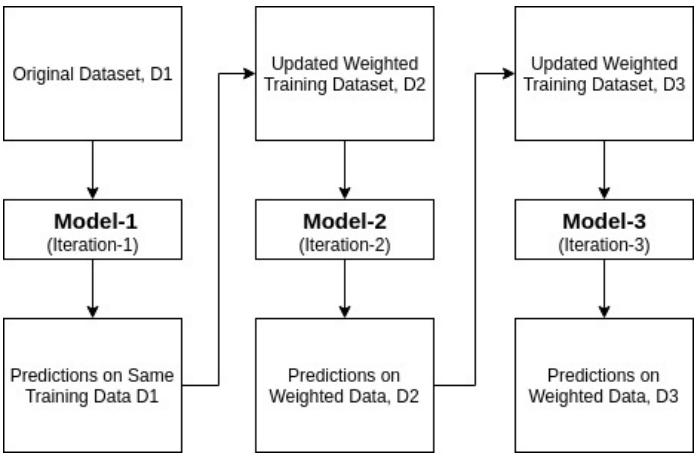


Figure 3.2.4: Workflow of the AdaBoost [19]

AdaBoost is particularly effective for binary classification tasks, making it ideal for e-commerce fraud detection. Its ability to adapt and emphasize challenging samples ensures high accuracy while reducing false positives. But it is sensitive to noisy data and outliers, which can impact performance. Despite this, AdaBoost's iterative improvement and compatibility with weak learners make it a powerful option for tackling complex fraud detection problems [14,19].

3.2.4 Model Evaluation

Once the models are trained, the next step is to evaluate their performance uisng the test data. The models are assessed on the 30% testing set to determine how well they generalize to unseen data. The performance evaluation is done using several key metrics to understand the models' effectiveness and accuracy.

Accuracy measures the proportion of total correct predictions (both fraudulent and legitimate transactions) out of all predictions made by the model.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (1)$$

Precision represents the proportion of predicted fraudulent transactions that are actually fraudulent.

$$Precision = \frac{TP}{TP+FP} \quad (2)$$

Recall, also known as **sensitivity**, measures the proportion of actual fraudulent transactions that are correctly identified by the model.

$$Recall = \frac{TP}{TP+FN} \quad (3)$$

F1-score, defined as the harmonic mean of precision and recall, provides a single metric that balances the trade-off between the two.

$$F1\ Score = 2 \times \frac{Precision \times Recall}{Precision+Recall} \quad (4)$$

Matthews Correlation Coefficient (MCC) evaluates the correlation between predicted and actual classifications, taking into account all four elements of the confusion matrix: true positives, true negatives, false positives, and false negatives.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (5)$$

Area Under the Curve (AUC) measures the model's capability to differentiate between fraudulent and legitimate transactions. It summarises the performance of the ROC curve, which depicts the relationship between true positive rate (recall) and the false positive rate across different classification thresholds. A higher AUC reflects stronger model performance in separating the two classes.

$$AUC = \int_0^1 ROC\ Curve \quad (6)$$

Classification Report provides a summary of evaluation metrics like precision, recall F1-Score and support for each class, allowing to determine how well the model performs in differentiate fraudulent and legitimate transactions.

Confusion Matrix shows the number of correct and incorrect predictions, break down by class like fraudulent and legitimate transactions. It helps to identify the types of errors the model makes, including false positives and false negatives.

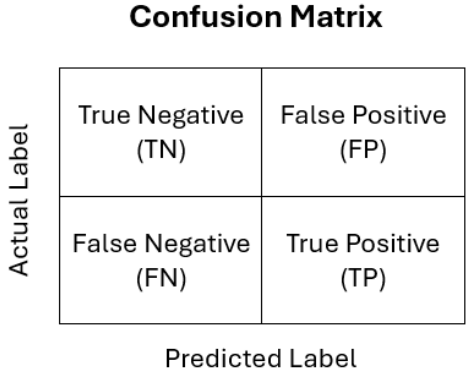


Table 3.2.1: Confusion Matrix

3.2.5 Hyperparameter Tuning

After the initial evaluation using default hyperparameters, the next step is to fine-tune these hyperparameters to improve the model's performance. Hyperparameter tuning involves searching for the optimal set of hyperparameters that enable the model to generalize better on unseen data.

In this study, both Grid Search and Randomized Search approaches were applied and compared. **Grid Search** exhaustively evaluates all possible combinations of hyperparameters within the defined search space. While it is more comprehensive, it is also computationally expensive, especially when the number of parameters and search ranges are large. **Randomized Search**, on the other hand, selects a fixed number of random combinations from the specified parameter distributions. This makes it more efficient and faster than Grid Search, though it may miss some optimal combinations.

The key hyperparameters considered for ensemble models include **n_estimators, learning_rate, max_depth, min_samples_split, min_samples_leaf**, which are relevant to models such as Random Forest, XGBoost, and AdaBoost. These parameters directly influence model complexity, learning behavior, and generalization performance.

By applying and **comparing both methods**, the trade-off between computational efficiency (Randomized Search) and thoroughness (Grid Search) can be evaluated. In cases where the default hyperparameters already produce effective results, the model may retain those settings to balance performance with computational cost.

3.2.6 Synthetic Data Generation

To enhance model evaluation and dashboard deployment, synthetic data was generated as an alternative test set. Such datasets preserve privacy while maintaining the statistical properties of the original, enabling reliable testing beyond limited real-world samples.

Generative models are commonly used to create synthetic data. **Generative Adversarial Networks (GANs)** can produce new data by learning patterns from real datasets, so the generated data looks statistically similar to the original [23][24]. **Variational Autoencoders (VAEs)** work by compressing data into a smaller hidden space and then reconstructing it back,

which allows them to generate new samples that still follow the main characteristics of the original data [24][25].

In this study, two specialized approaches for tabular data were applied:

- **Tabular Variational AutoEncoders (TVAE):** TVAE **excels in data replication and augmentation** by effectively learning the underlying data distribution through latent space representations [26].
- **Conditional Tabular Generative Adversarial Networks (CTGAN):** CTGAN **achieves a balance across fidelity, synthesis quality, efficiency, privacy, and graph structure** [26].

The synthetic test sets produced by both TVAE and CTGAN were used to **re-evaluate model performance**. This comparison aimed to identify whether synthetic data could serve as a reliable proxy for real-world test data, especially during dashboard integration and continuous evaluation. Finally, the results on the synthetic test sets were compared against the performance on the **Kaggle real test set**. This comparison provided insights into the trade-offs between using synthetic and real data, supporting the decision on whether to adopt synthetic datasets or to retain reliance on the real Kaggle test set for final deployment.

3.2.7 Model Deployment to Power BI

Once the best-performing model is selected and thoroughly validated, the next step is to deploy it into a production environment where it can provide real-time predictions and insights. In this project, the model is integrated with Power BI, allowing users to visualize, monitor and interact with fraud detection system dashboard efficiently.

1. Export the model for Deployment

The first step in model deployment is to export the trained model into a suitable format that can be used in a production environment. The *joblib* library is used to serialize the trained model into a file, making it portable and shareable across different environments. The model export process is a key step to ensure that the model can be reloaded and reused without retraining.

2. Integrating the Model with Power BI

Once the model is exported, it is integrated into Power BI for fraud detection. This is done by adding a Python script in Power BI's query editor to load the saved model and run predictions on new data. Before making predictions, preprocessing steps like encoding techniques are applied within the same Python script to ensure the new data is in the correct format for the model. The model then predicts whether transactions are fraudulent or legitimate. The script is executed automatically each time the data is refreshed in Power BI.

3. Designing the Dashboard Layout and Add Visualizations

The core of the deployment process is to create an **interactive Power BI dashboard** that presents the **fraud patterns, fraud prediction results and relevant metrics** in a clear and intuitive manner. This dashboard is tailored to help both data analysts and e-commerce administrators monitor the effectiveness of the fraud detection system and identify actionable insights quickly.

To improve interpretation, different types of visual elements such as **cards, clustered column charts, stacked column charts, doughnut charts, line charts, tables** and others**.** will be used to represent fraud patterns, model performance and other insights. Slicers will also be added to allow users to filter data by variables such as date and prediction result to enable more flexible and targeted analysis.

3.2.8 Dashboard Testing

Testing a Power BI dashboard involves a thorough process to ensure that data, visuals, interactivity and overall user experience are functioning correctly.

1. Data accuracy testing

This involves checking that all data sources are properly connected and pulling the latest data when refresh data. Sample data from the dashboard should be cross-checked against the raw data to confirm consistency, and calculated metrics should be accurate.

2. Visual accuracy testing

Each visual should accurately represent the intended metric. Charts like line, bar and pie must accurately reflect trends, distributions and proportions. Labels, axes, legends and data points should be clear, properly formatted and easy to interpret. Visual design, such as font sizes, colours, and spacing should also be verified to maintain readability and consistency.

3. Slicer testing

Slicers and filters should be tested to confirm they filter data correctly by categories or ranges. It is important to verify that slicer selections update related visuals dynamically and do not cause any display errors. Multiple slicers should function together without conflicts and removing filters should reset visuals to their default view.

4. Performance testing

Performance testing is to make sure the dashboard responds quickly and operates smoothly. This includes evaluating the loading time of the report, the responsiveness of visuals when interacting with slicers and overall usability with large datasets. Measures and complex visuals should be reviewed and optimised to avoid performance delays. Additionally, the time taken to refresh data should be monitored to ensure it within an acceptable range.

5. User acceptance testing

This testing is conducted using the **System Usability Scale (SUS) questionnaire**. SUS is used because it is recognized as the most commonly adopted instrument for dashboard evaluation, providing a general and consistent measure of usability [27]. Respondents were first asked to use the dashboard and then complete the SUS survey, which assesses key aspects such as usability, clarity, responsiveness, interactivity, and overall user satisfaction.

The SUS consists of 10 statements rated on a 5-point Likert scale, ranging from *"Strongly Disagree" (1)* to *"Strongly Agree" (5)*, with both positively and negatively worded items. For scoring, positive items are calculated as **(Response − 1)**, and negative items are calculated as **(5 − Response)**. This process standardises all values to a range of 0 to 4. The recoded values for each respondent are then summed to obtain a total score between 0 and 40, which is multiplied by 2.5 to yield a final SUS score ranging from 0 to 100.

The success criteria for this testing include achieving an average SUS score of at least 70, ensuring no major technical or usability issues, and confirming that the dashboard meets user expectations. Feedback collected from the SUS survey is subsequently used to refine and improve the dashboard, ensuring that it effectively supports fraud monitoring and analysis tasks while providing a user-friendly experience.

3.3 User Case

3.3.1 Use Case Diagram

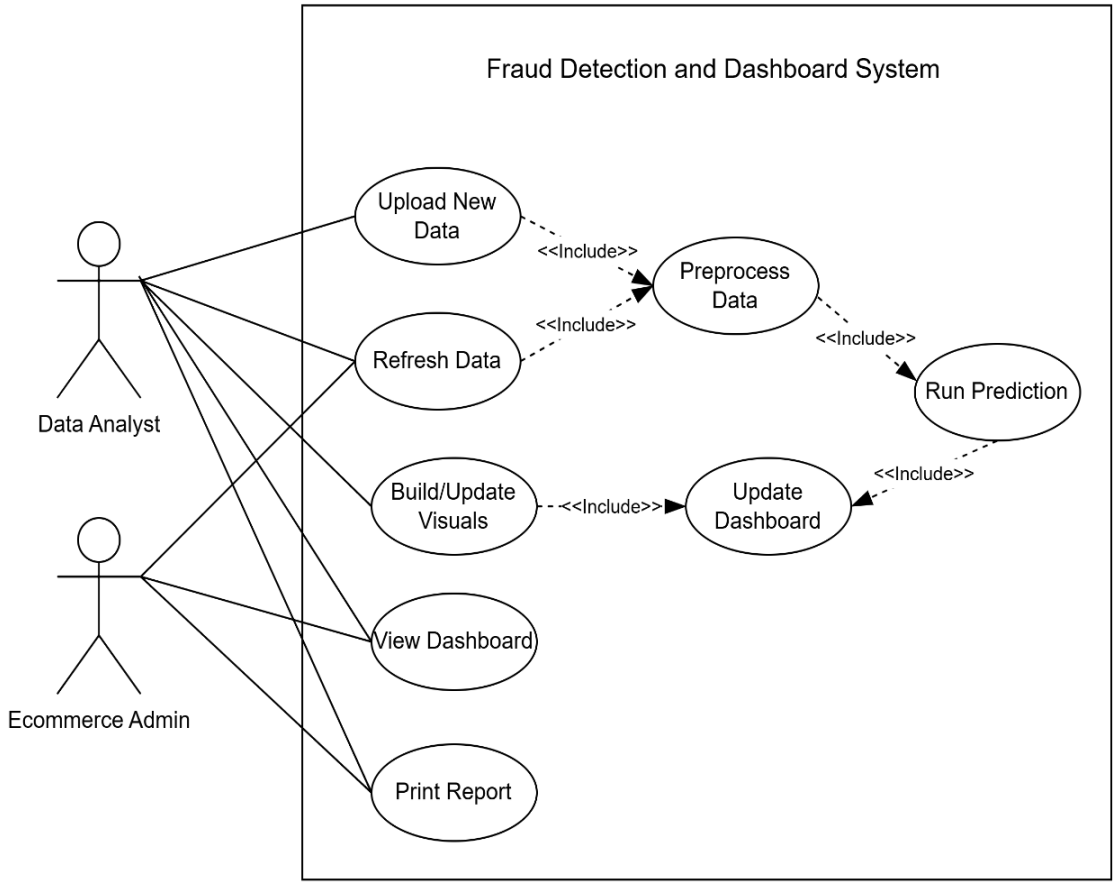


Figure 3.3.1: Use case diagram

3.3.2 Use Case Description

<table>
<tr><td colspan="2">Use Case Name: Upload New Data</td><td>ID: 01</td><td>Importance Level: High</td></tr>
<tr><td colspan="2">Primary Actor(s): Data Analyst</td><td colspan="2">Use Case Type: Detail, essential</td></tr>
<tr><td colspan="4">Stakeholders and Interests:
Data Analyst: Wants to upload new data for updated analysis, predictions, and dashboard reporting.</td></tr>
<tr><td colspan="4">Brief Description: Data Analyst uploads new dataset to the system. The system then automatically preprocesses the data, applies a trained prediction model, and updates the Power BI dashboard with the new data and predictions.</td></tr>
<tr><td colspan="4">Trigger: A new or updated data file (e.g., CSV) is available and selected by the Data Analyst for upload.</td></tr>
<tr><td colspan="4">Relationship:
Association: Data Analyst
Include: Preprocess Data, Run prediction, Update Dashboard
Extend: None
Generalization: None</td></tr>
<tr><td colspan="4">Normal Flow of Events:
1. The Data Analyst selects Get Data and chooses the data source type (e.g., CSV).
2. The Data Analyst provides connection details (e.g., file path).
3. The system retrieves the data from the source and displays a preview.
4. The Data Analyst confirms the data and clicks Load.
5. The system loads the data into Power BI.
6. The system creates a duplicate of the dataset to serve as a working copy (preserves original/raw data).
7. The system applies preprocessing techniques (e.g., encoding) to the working dataset.
8. The trained prediction model is retrieved and initialized.
9. The system runs the prediction algorithm on the pre-processed data.
10. Prediction results are appended to the dataset.
11. The system checks for changes or additions in the data.
12. The dashboard elements are automatically updated based on the new data and predictions.</td></tr>
<tr><td colspan="4">Sub Flows: None</td></tr>
<tr><td colspan="4">Alternative/Exceptional Flows: None</td></tr>
<tr><td colspan="2">Use Case Name: Refresh Data</td><td>ID: 02</td><td>Importance Level: High</td></tr>
</table>

Primary Actor(s): Data Analyst, Ecommerce Admin	Use Case Type: Detail, essential
Stakeholders and Interests: Data Analyst: Requires up-to-date data for accurate analysis, dashboard reporting, and insights. Ecommerce Admin: Depends on current data to support timely decision-making and operational strategies.	
Brief Description: Data Analyst or Ecommerce Admin manually initiates a data refresh in Power BI. This triggers automatic preprocessing and prediction steps using the updated data. The results are then reflected in the Power BI dashboard.	
Trigger: Manual selection of the "Refresh" option within Power BI.	
Relationship: Association: Data Analyst, Ecommerce Admin Include: Preprocess Data, Run Prediction, Update Dashboard Extend: None Generalization: None	
Normal Flow of Events: 1. Data Analyst or Ecommerce Admin selects the option to refresh the data. 2. Power BI retrieves the latest data from the connected source. 3. The system creates a duplicate of the dataset to serve as a working copy (preserves original/raw data). 4. The system applies preprocessing techniques (e.g., encoding) to the working dataset. 5. The trained prediction model is retrieved and initialized. 6. The system runs the prediction algorithm on the pre-processed data. 7. Prediction results are appended to the dataset. 8. The system checks for changes or additions in the data. 9. The dashboard elements are automatically updated based on the new data and predictions.	
Sub Flows: None	
Alternative/Exceptional Flows: None	

Use Case Name: Build/Update Visuals	ID: 03	Importance Level: High
Primary Actor(s): Data Analyst	Use Case Type: Detail, essential	
Stakeholders and Interests: Data Analyst: Requires the flexibility to create or customize dashboard visuals to align with business goals and analytical needs. Ecommerce Admin: Benefits from clear, relevant, and easy-to-understand visuals to monitor fraud trends and make informed decisions.		
Brief Description: The Data Analyst creates new visuals or updates existing ones in the dashboard using selected data fields and visual types.		
Trigger: Data Analyst initiates the creation or update of dashboard visuals in Power BI.		
Relationship: Association: Data Analyst Include: Update Dashboard Extend: None Generalization: None		
Normal Flow of Events: 1. The Data Analyst selects data fields from the model within Power BI. 2. The Data Analyst chooses the type of visual (e.g., bar chart, line chart, cards). 3. The system generates new visual based on selected fields and format. 4. The system automatically refreshes and displays the updated dashboard view.		
Sub Flows: Update Existing Visual: If an existing visual is being updated, the system replaces the current visual with the newly configured one while retaining layout consistency.		
Alternative/Exceptional Flows: None		

Use Case Name: View Dashboard	ID: 04	Importance Level: High
Primary Actor(s): Data Analyst, Ecommerce Admin	Use Case Type: Detail, essential	
Stakeholders and Interests: Data Analyst: Requires visibility into up-to-date data, predictive outputs, and visual trends for monitoring and analysis. Ecommerce Admin: Uses dashboard insights to support decision-making and guide business strategy.		
Brief Description: Data analyst and Ecommerce admin view the Power BI dashboard to access current data, prediction results, and visualizations related to model performance and trends.		
Trigger: Open or navigate to the dashboard within the Power BI platform.		
Relationship: Association: Data Analyst, Ecommerce Admin Include: None Extend: None Generalization: None		
Normal Flow of Events: 1. User navigates to the dashboard. 2. The system displays the dashboard. 3. User selects a specific page to view (e.g., Overview, Fraud Patterns, Model Performance).		
Sub Flows: None		
Alternative/Exceptional Flows: None		

<table>
<tr><td>Use Case Name: Export Report</td><td>ID: 05</td><td>Importance Level: Medium</td></tr>
<tr><td>Primary Actor(s): Data Analyst, Ecommerce Admin</td><td colspan="2">Use Case Type: Detail, essential</td></tr>
<tr><td colspan="3">Stakeholders and Interests:
Data Analyst: Needs to generate and share visual reports for meetings, documentation, or offline analysis.
Ecommerce Admin: Requires snapshot reports to review business performance and share insights with stakeholders.</td></tr>
<tr><td colspan="3">Brief Description: Data analyst and Ecommerce admin export a report of the current dashboard view, including visuals, in PDF format.</td></tr>
<tr><td colspan="3">Trigger: Initiates the export process from the Power BI dashboard settings menu.</td></tr>
<tr><td colspan="3">Relationship:
Association: Data Analyst, Ecommerce Admin
Include: None
Extend: None
Generalization: None</td></tr>
<tr><td colspan="3">Normal Flow of Events:
1. The user can apply desired filters/slicers to select specific data to include in the report.
2. The user clicks on the ‘File’ tab and selects ‘Export’ > ‘Export to PDF’.
3. The system generates a PDF report of the current dashboard view.
4. The system automatically downloads the PDF file to the user's device.</td></tr>
<tr><td colspan="3">Sub Flows: None</td></tr>
<tr><td colspan="3">Alternative/Exceptional Flows: None</td></tr>
</table>

3.4 Timeline

The project timeline is divided across two semesters. In **Final Year Project 1**, it will be start with research and planning, followed by data collection and preprocessing. Once the data is prepared, preliminary modelling will be conducted, and progress will be documented in the report. In **Final Year Project 2**, the work will begin with hyperparameter tuning to optimize model performance, followed by finalizing the best model. The selected model will then be deployed to Power BI, where a functional dashboard will be created to visualize and interact with predictions. Afterward, various testing will be carried out to evaluate the dashboard's effectiveness. The project will conclude with the preparation and submission of the final report and presentation. Gantt charts are used to illustrate the detailed timeline and key milestones for both semesters.

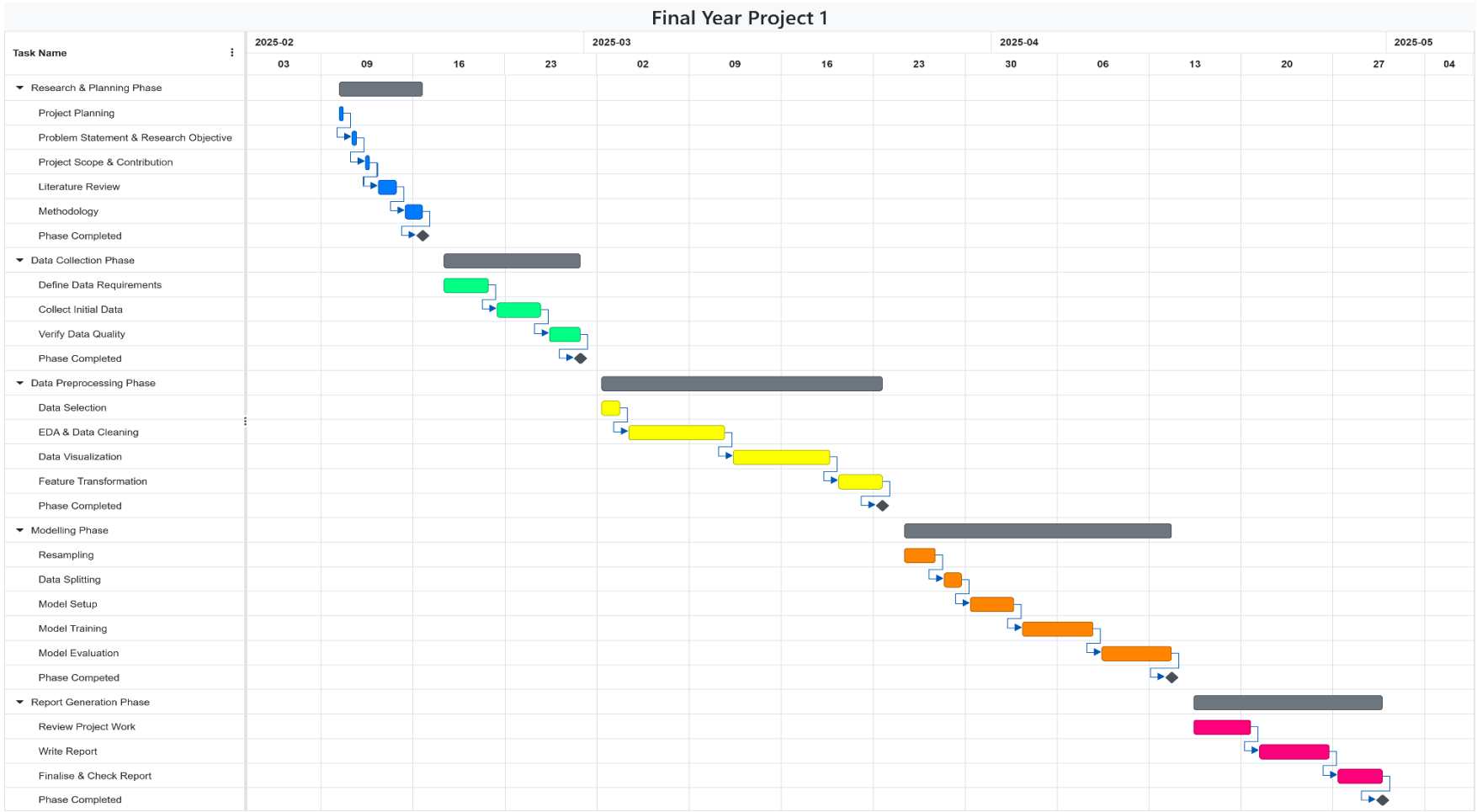


Figure 3.4.1: Gantt Chart for Final Year Project 1

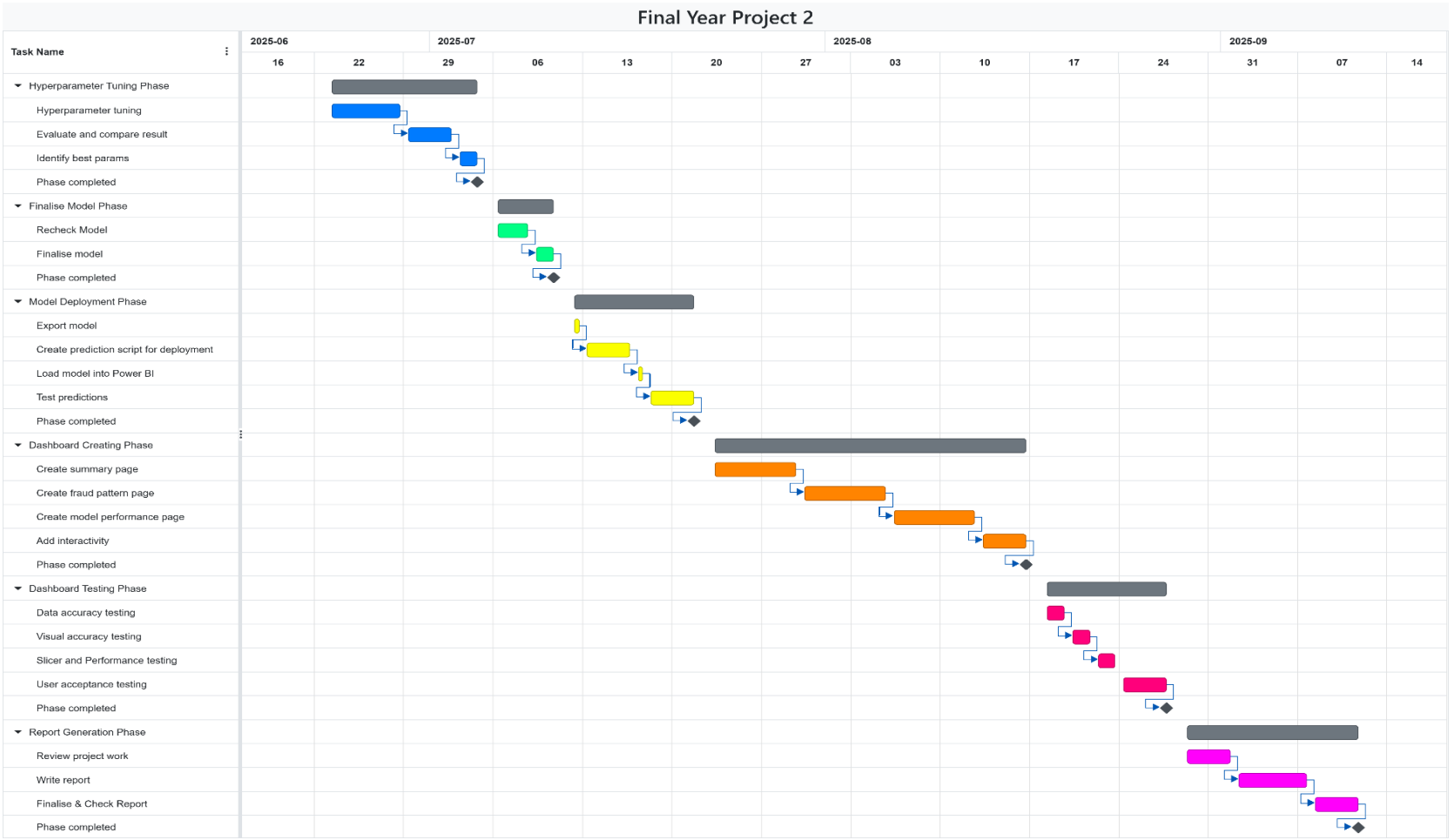


Figure 3.4.2: Gantt Chart for Final Year Project 2

CHAPTER 4

System Design

This chapter outlines the design of the fraud detection dashboard, covering data flow, model integration from Jupyter to Power BI, and the dashboard's layout and user interactions through wireframes.

4.1 System Block Diagram

The **system block diagram** illustrates the overall structure of the fraud detection dashboard system, as shown in Figure 4.1.1. It emphasizes how data flows from the input dataset to the end-user interface, as well as how machine learning components, which were originally developed in Jupyter Notebook, are integrated into Power BI for automated use.

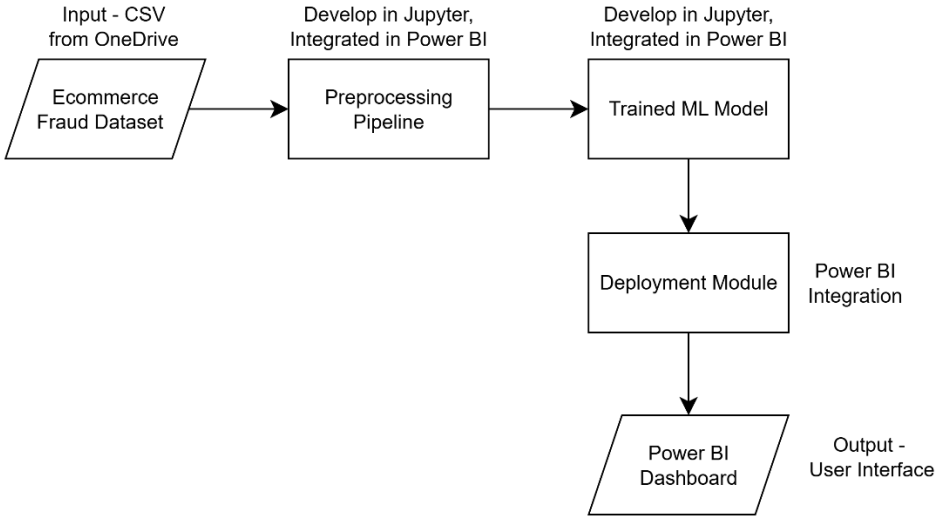


Figure 4.1.1: System Block Diagram of the Fraud Detection Dashboard

E-commerce Fraud Dataset (Input Source)

The dataset, stored in OneDrive in CSV format, acts as the primary input. It contains transaction records that include both fraudulent and non-fraudulent cases. The dataset can be updated by replacing the file or link, after which the dashboard will automatically refresh.

Preprocessing Pipeline (Developed in Jupyter, Integrated in Power BI)

During development, preprocessing steps such as encoding and resampling were designed and tested in Jupyter Notebook. Once finalized, the pipeline was exported and embedded into Power BI through Python scripting. Inside Power BI, the pipeline is not retrained but reused to transform any new incoming dataset consistently.

Trained Machine Learning Model (Developed in Jupyter, Integrated in Power BI)
Model training and evaluation (e.g., Random Forest) were conducted in Jupyter Notebook using the processed dataset. The final trained model was saved and then integrated into Power BI. Similar to the preprocessing pipeline, the model does not undergo retraining in Power BI. Instead, it is loaded and applied directly to generate predictions whenever the dashboard data is refreshed.

Deployment Module (Power BI Integration)
After preprocessing and prediction, the results are loaded into Power BI's data model. This deployment step links the Python output with Power BI tables, ensuring that visuals (charts, KPIs, metrics) automatically update based on the latest dataset.

Power BI Dashboard (User Interface)
The dashboard presents the final results to end-users in a structured and interactive way. It consists of multiple pages—Homepage, Overview, Time Analysis, Geography, Demographics, Behavioural Analysis, Model Performance, Prediction Confidence & Key Influencers, Credit Crad Transaction and Transaction Details—that allow fraud patterns and model performance to be explored at different levels of detail. From the user's perspective, the workflow is simple: they only need to refresh the dashboard, and all preprocessing, prediction, and visualization updates occur automatically in the background.

4.2 System Components Design (Wireframe)

The wireframe acts as a blueprint for the fraud detection dashboard, showing the layout, key components, and user interactions. It ensures clarity in organizing visuals and navigation from summary insights to detailed analysis.

Homepage

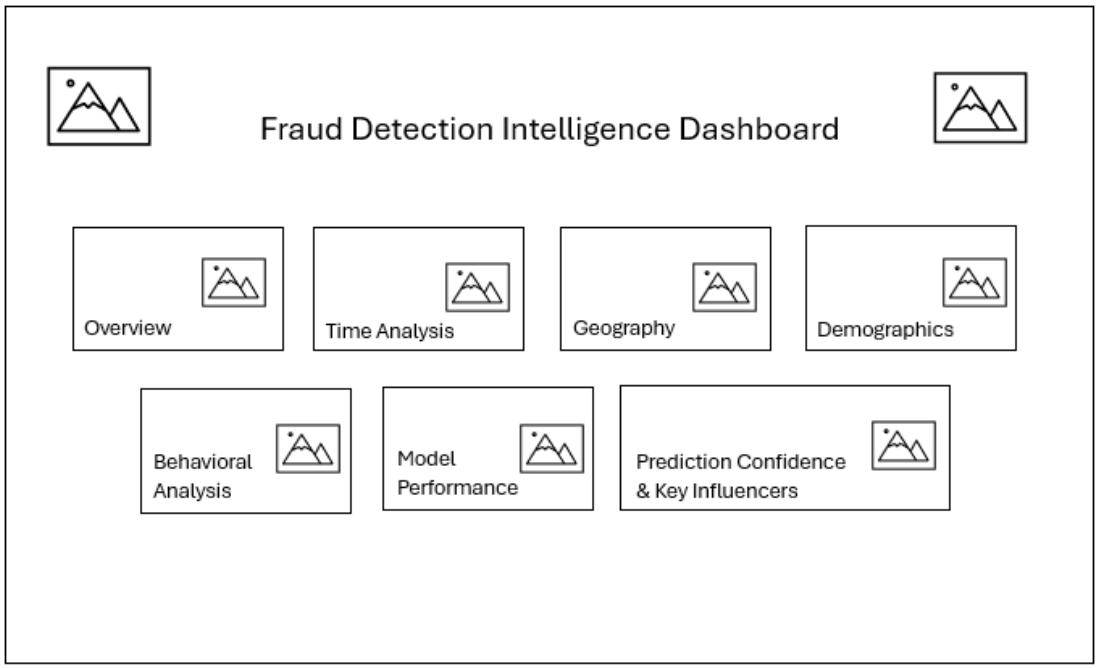


Figure 4.2.1: Wireframe of Homepage

Purpose: Entry point of the system; provides navigation to all pages.

Components & Implementation:

- **Navigation Buttons:** Power BI **Blank Buttons**, set **Action = Page Navigation**. Redirects to Overview, Time Analysis, Geography, Demographics, Behavioral Analysis, Model Performance, Prediction Confidence & Key Influencers, Credit Card Transactions, and Transaction Details.

Overview Page

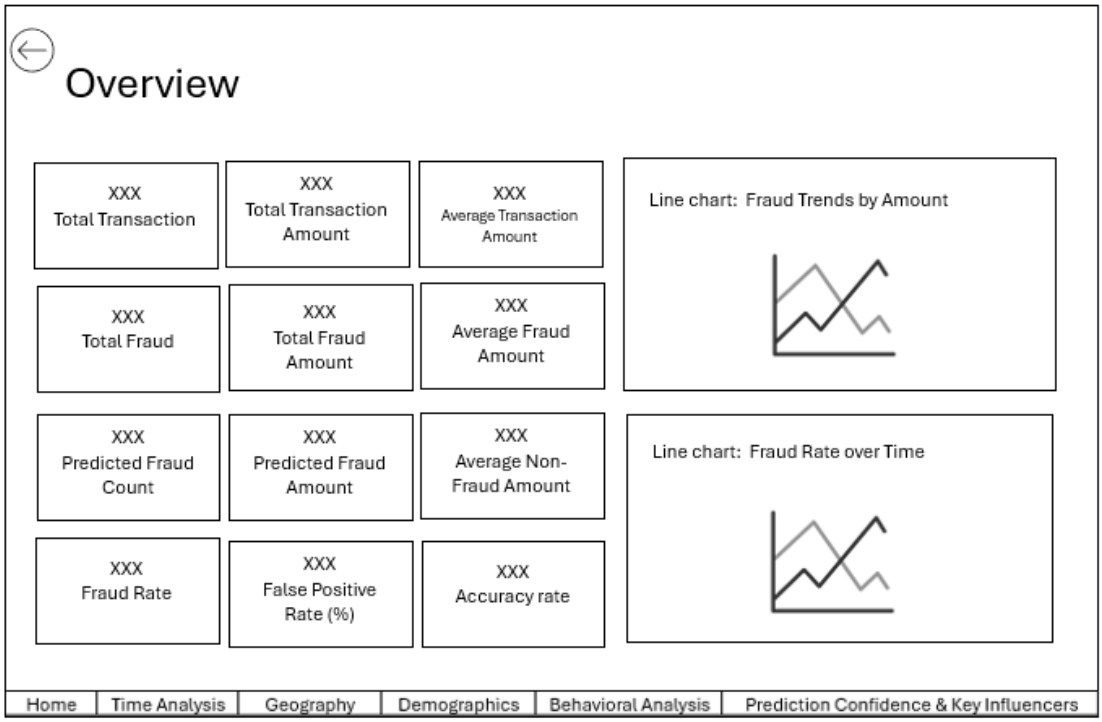


Figure 4.2.2: Wireframe of Overview Page

Purpose: Summary dashboard of dataset insights and model performance.

Components & Implementation:

- **KPI Metrics Cards:** Total transactions, total & average transaction amounts, fraud count & amount, predicted fraud count & amount, fraud rate, model accuracy, false positive rate.
    - Linked to **DAX measures** (SUM, COUNTROWS, AVERAGE, CALCULATE).
    - Conditional formatting:
        - Fraud rate: <0.1 green, <0.3 yellow, >0.3 red
        - Accuracy: >0.9 green, >0.75 yellow, <0.75 red
- **Line Charts:** Fraud vs non-fraud trends over time, Fraud rate over time
- **Navigation Buttons:** Bottom of page using **Page Navigator buttons**.

Time Analysis Page

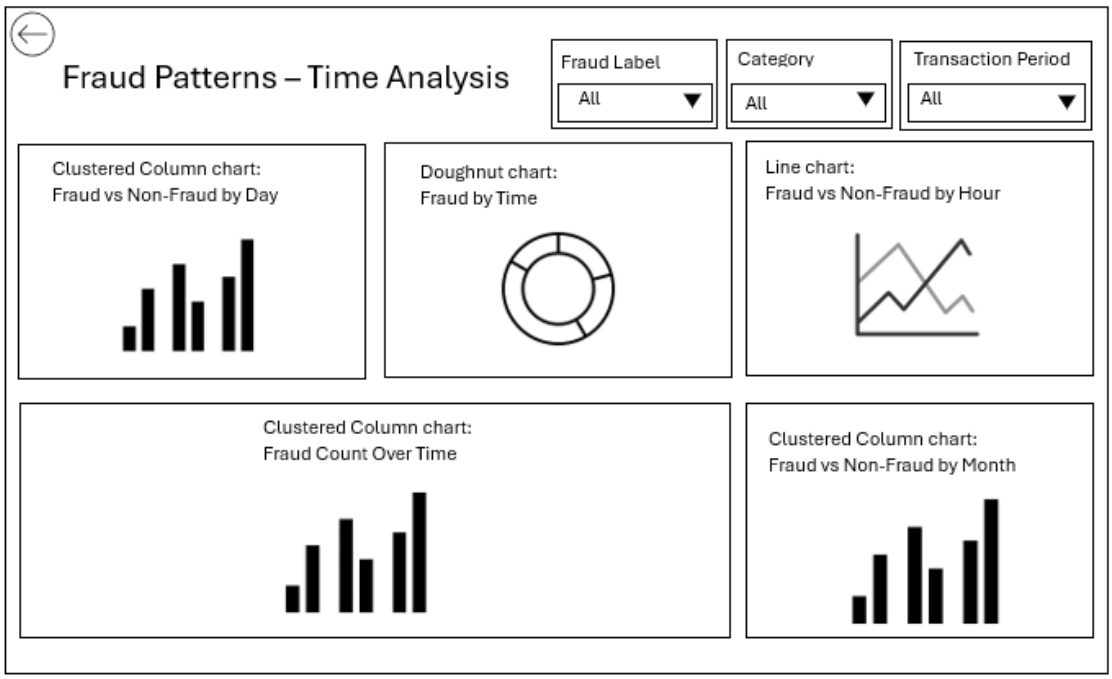


Figure 4.2.3: Wireframe of Time Analysis Page

Purpose: Identify fraud patterns across periods.

Components & Implementation:

- **Column Charts:** Fraud vs non-fraud by day of week, Fraud count over time (day of month), and Fraud vs non-fraud by month.
- **Doughnut Chart:** Fraud by time (day/night).
- **Line Chart:** Fraud vs non-fraud by hour.
- **Slicers:** Fraud label, transaction category, transaction period.

Geography Page

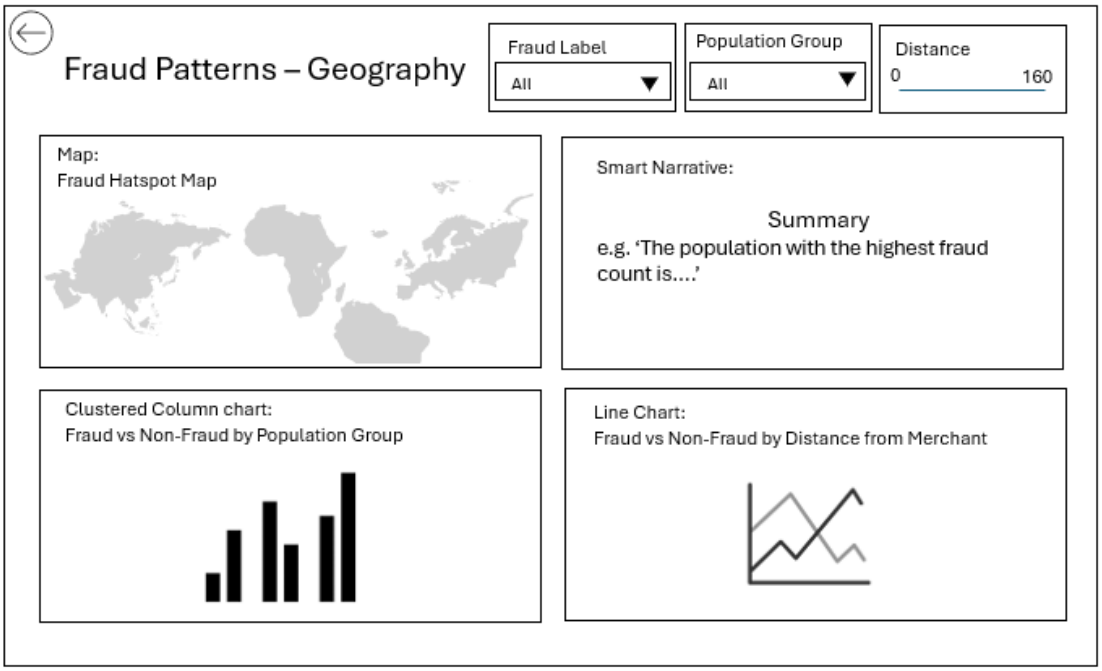


Figure 4.2.4: Wireframe of Geography Page

Purpose: Spatial insights for fraud detection.

Components & Implementation:

- **Map Visual:** Plot transactions by city.
    - Color: Blue = normal, Red = fraud.
    - Tooltips: city, fraud label, lat/long, transaction count, distance from merchant.
    - Enable **zoom/pan**.
- **Column Chart:** Fraud vs non-fraud by population group.
- **Line Chart:** Fraud vs non-fraud by distance from merchant.
- **Summary: Smart Narrative Visual**, linked to DAX measures.
- **Slicers:** Fraud label, population group, distance from merchant (**between-style slicer**).

Demographics Page

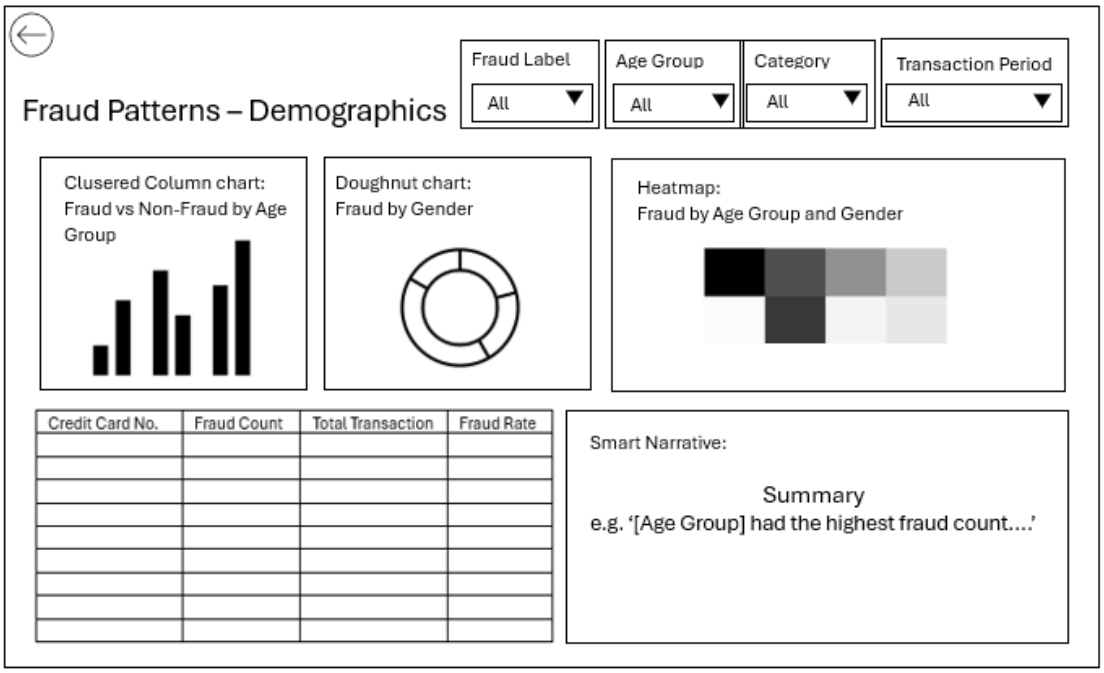


Figure 4.2.5: Wireframe of Demographics Page

Purpose: Explore fraud based on customer attributes.

Components & Implementation:

- **Column Chart:** Fraud vs non-fraud by age group.
- **Doughnut Chart:** Fraud by gender.
- **Cross Table:** Fraud by age group and gender using **Heatmap**.
- **Credit Card Transactions Table:**
  - Columns: Credit card no, fraud count, total transactions, fraud rate.
  - Drill-through to **Credit Card Transactions Page** via right-click on credit card no.
  - Conditional formatting: Fraud rate near 100% = red, near 0% = no color.
- **Summary: Smart Narrative Visual**, DAX measures for insights.
- **Slicers:** Fraud label, age group, category, transaction period.

Behavioral Analysis Page

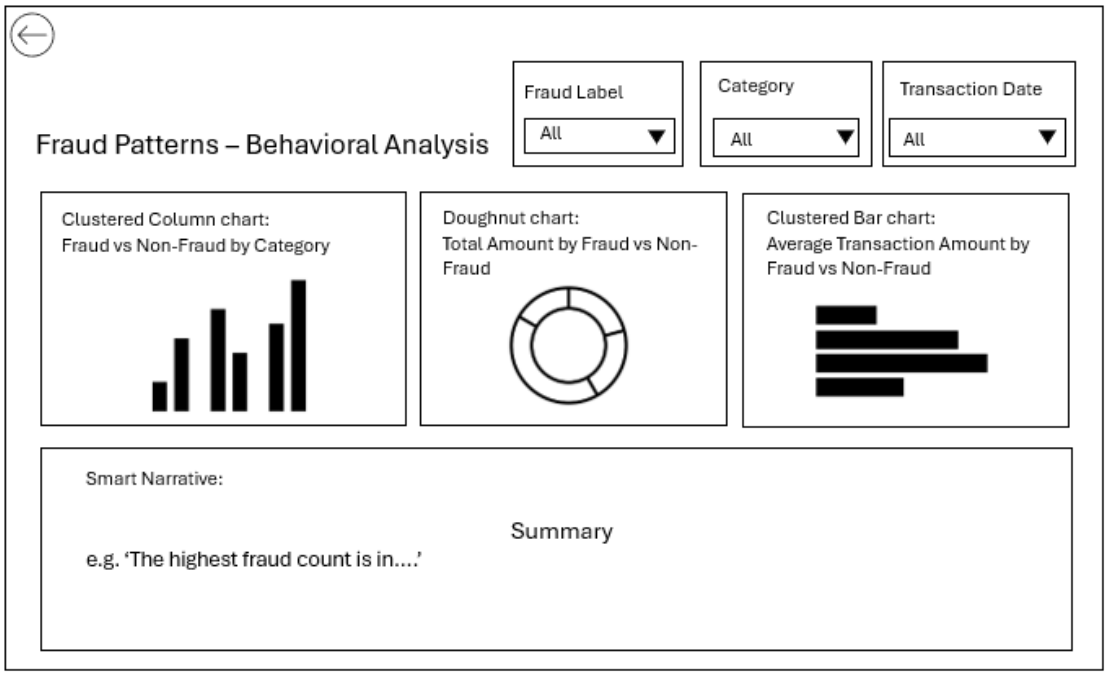


Figure 4.2.6: Wireframe of Behavioral Analysis Page

Purpose: Examine patterns in customer behaviour for anomalies.

Components & Implementation:

- **Column Chart:** Fraud vs non-fraud by category.
- **Doughnut Chart:** Total transaction amount by fraud label.
- **Bar Chart:** Average transaction amount by fraud label.
- **Summary: Smart Narrative Visual**, DAX measures for insights.
- **Slicers:** Fraud label, category, transaction date.

Model Performance Page

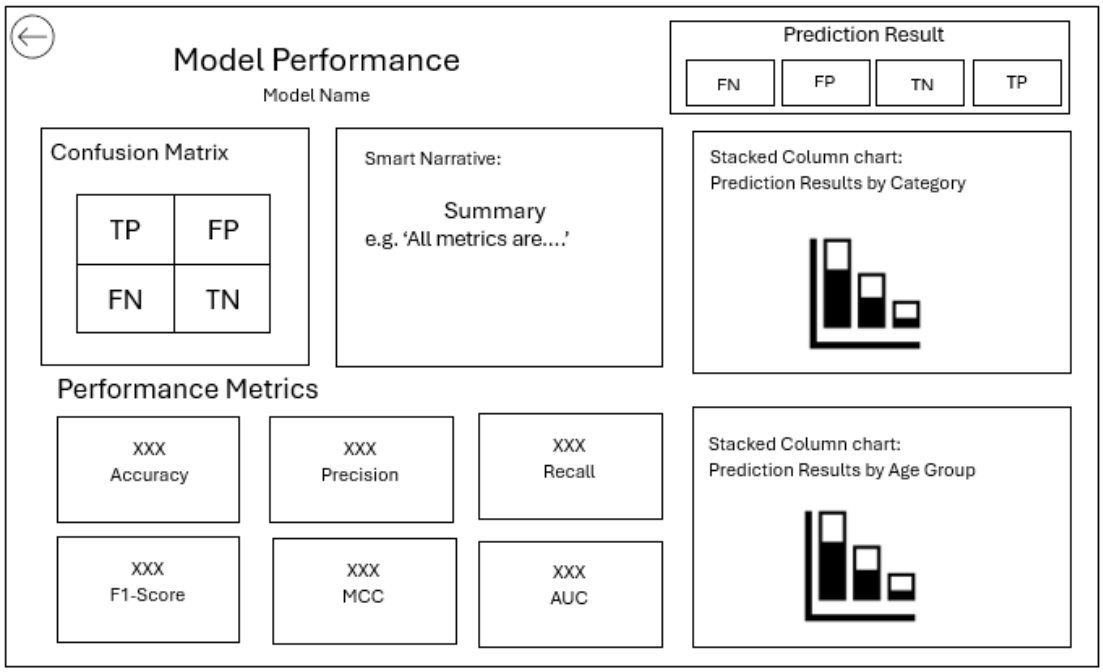


Figure 4.2.7: Wireframe of Model Performance Page

Purpose: Evaluate ML model performance in detecting fraud.

Components & Implementation:

- **Confusion Matrix:** Using **Heatmap visual**.
- **Performance Metrics Cards:** Accuracy, Precision, Recall, F1-score, MCC, AUC.
    - Conditional formatting: >90% green, >75% yellow, <75% red.
- **Column Charts:** Prediction results by category and age group
- **Summary: Smart Narrative Visual** with dynamic insights based on DAX measures.
- **Slicer:** Prediction result (TP, FP, TN, FN); default selection = FN & FP.

Prediction Confidence & Key Influencers Page

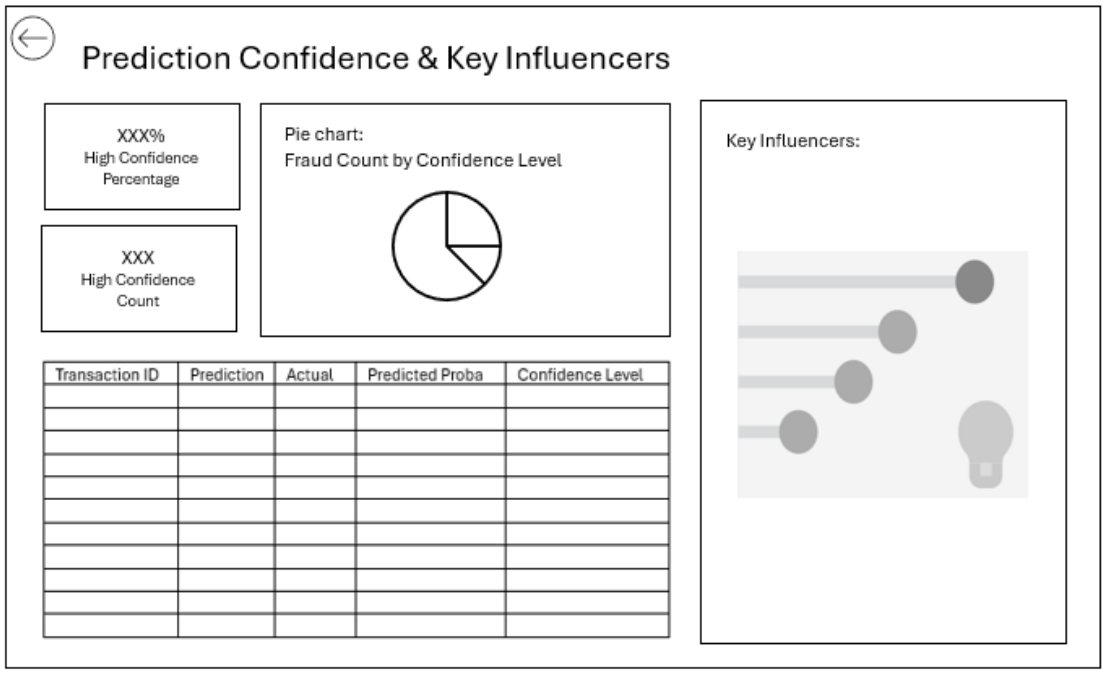


Figure 4.2.8: Wireframe of Prediction Confidence & Key Influencers Page

Purpose: Provide model explainability and key influencers for fraud prediction.

Components & Implementation:

- **Cards:** High confidence count & percentage (Predicted probability >0.8 is considered High confidence).
- **Pie Chart:** Fraud count by confidence level (Very High >0.9, High >0.8, Medium >0.5, Low <0.5).
- **Transaction History Table:**
    - Columns: Transaction ID, prediction, actual, predicted probability, confidence level.
    - Drill-through to **Transactions Page** via right-click on credit card no.
- **Key Influencers Visual:** AI visual automatically identify which features (e.g., transaction amount, category, time of day) most strongly influence whether a transaction is **fraud** or **non-fraud**. The visual then ranks these fields by **influence strength**, expressed as a **relative factor (x times more likely)**.

Credit Card Transactions Page

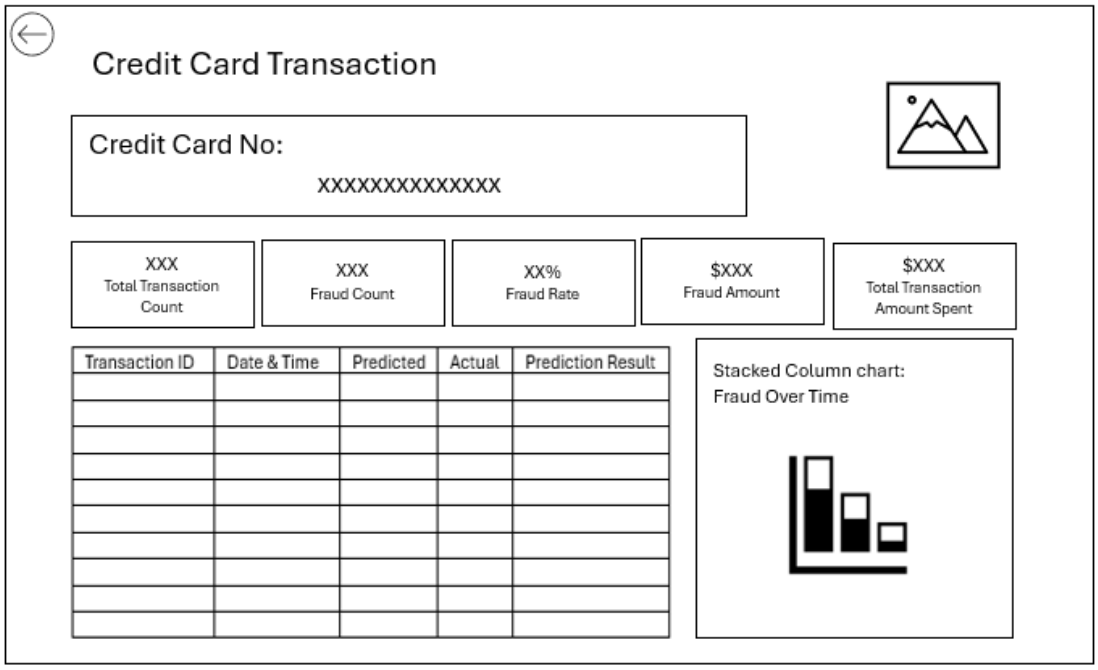


Figure 4.2.9: Wireframe of Credit Card Transactions Page

Purpose: Aggregated view of credit card activity; drill-through from Demographics Page.

Components & Implementation:

- **Cards:** Credit card no, Fraud count, total transactions, fraud rate, fraud amount, total transaction amount.
    - Conditional formatting: For Fraud Count card, if Fraud >0 highlighted red.
- **Transaction History Table:**
    - Columns: Transaction ID, date & time, predicted label, actual label, prediction result.
    - Drill-through to **Transaction Detail Page** via right-click on one of the transactions.
    - Conditional formatting: Fraud = red, Non-Fraud = Green; Wrong prediction (False Negative/ False Positive) = red, Correct prediction = Green.
- **Column Chart:** Fraud over time.

Transaction Details Page

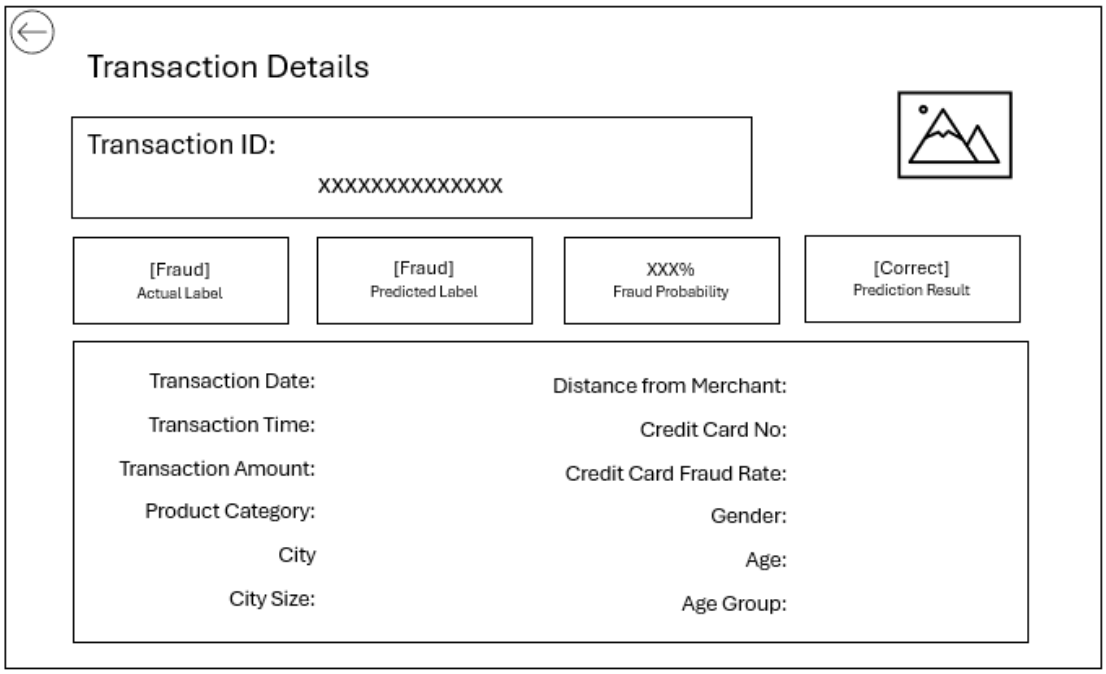


Figure 4.2.10: Wireframe of Transaction Details Page

Purpose: Most detailed analysis; drill-through from other pages.

Components & Implementation:

- **Cards:** Transaction ID, actual label, predicted label, fraud probability, prediction result.
    - Actual Label & Predicted Label: Fraud = red, Non-fraud = green.
    - Fraud probability: Gradient formatting, 0% = green, 100% = red.
    - Prediction result Correct = green; False Negative/False Positive = red.
- **Transaction Details Cards:** Date, time, amount, category, city, city size, distance, credit card no, credit card fraud rate, gender, age, age group.

CHAPTER 5

System Implementation

This chapter details the project implementation, covering software setup, data understanding, and preprocessing for machine learning model development. It also describes initial model testing, hyperparameter tuning, and performance evaluation to optimize each algorithm. Additionally, synthetic data generation was performed to support robust model evaluation and dashboard deployment. Finally, the trained models and preprocessing pipelines were integrated into Power BI, where the interactive dashboard was developed to visualize predictions and insights.

5.1 Setting up

5.1.1 Software/Tools

Before starting the project, there are several software/tools are downloaded and installed on the laptop. These includes:

- **Jupyter Notebook**
- **Python**
- **Google Collaboratory (no need installation)**
- **Power BI Desktop**

Figure 4.1.1 shows the versions of the key Python libraries used in this project, including pandas, numpy, scikit-learn, matplotlib, seaborn, xgboost, and imbalanced-learn.

```
Python: 3.11.7 | packaged by Anaconda, Inc. | (main, Dec 15 2023, 18:05:47) [MSC v.1916 64 bit (AMD64)]
pandas: 2.1.4
numpy: 1.26.4
scikit-learn: 1.6.1
matplotlib: 3.8.0
seaborn: 0.12.2
xgboost: 2.1.3
imbalanced-learn: 0.13.0
```

Figure 5.1.1: Version of Python and Various Libraries.

5.2 Initial Dataset (Aborted)

Aborted in here refers to the decision to discontinue the use of the initially selected dataset for model training due to the weak correlations between features and the target variable (*is_fraud*).

This limitation may reduce its effectiveness for fraud detection. A more suitable dataset with stronger predictive features was sourced instead.

5.2.1 Dataset Selection

The initial dataset used in this project is the "Financial Transactions Dataset: Analytics" from Kaggle, which consisted of five separate files, with a total of 39 columns as shown in *Figure 4.2.1*. The dataset is designed for various financial applications such as fraud detection, customer analytics and expense forecasting. The dataset includes the following files:

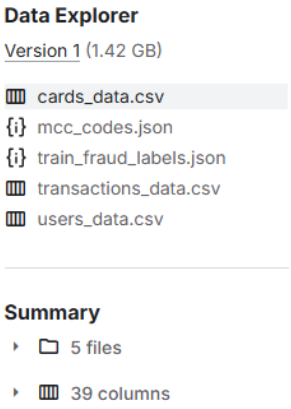


Figure 5.2.1: Initial Dataset Information

- **transactions_data.csv**: Containing detailed records of transactions such as timestamps, amounts, and merchant information
- **users_data.csv**: Containing demographic and account-related information about users
- **cards_data.csv**: Containing card-specific information including card types and limits
- **train_fraud_labels.json**: Containing binary fraud labels (fraudulent vs. legitimate) for supervised learning
- **mcc_codes.json**: Containing listed merchant category codes (MCC) with corresponding descriptions for categorizing transaction types.

5.2.2 EDA and Preprocessing of Initial Dataset

All **datasets are loaded**. To process the fraud labels and MCC descriptions, the data is converted from a json file into a DataFrame.

```
# Load the CSV files into DataFrames
transactions_df = pd.read_csv('transactions_data.csv')
users_df = pd.read_csv('users_data.csv')
cards_df = pd.read_csv('cards_data.csv')
```

```
with open('train_fraud_labels.json', 'r') as file:
    fraud = json.load(file)

fraud_id = list(fraud['target'].keys())
fraud_status = list(fraud['target'].values())
fraud = pd.DataFrame({"ID": fraud_id, 'is_fraud': fraud_status})
```

```
with open('mcc_codes.json', 'r') as file:
    code = json.load(file)

codes = list(code.keys())
name = list(code.values())
description = pd.DataFrame({'MCC': codes, 'mcc_name': name})
```

```
fraud['ID'] = fraud['ID'].astype(int)
```

```
description['MCC'] = description['MCC'].astype(int)
```

Figure 5.2.2: Loading Dataset

Transactions are **merged** with **fraud labels** using the transaction id, then merged with **MCC codes** via the *mcc* column. **User data** is merged using *client_id* as the key, followed by **card data** using *card_id*. Duplicate ID-related columns are removed after merging.

```
merge_df = pd.merge(transactions_df, fraud, how='left', left_on='id', right_on='ID')
merge_df = merge_df.dropna(subset=['is_fraud'])
# Drop the 'ID' column from the merged DataFrame
merge_df = merge_df.drop(columns=['ID'])
```

```
merge_df = pd.merge(merge_df, description, how='left', left_on='mcc', right_on='MCC')
# Drop the 'mcc' column from the merged DataFrame
merge_df = merge_df.drop(columns=['MCC'])
```

```
merge_df = pd.merge(merge_df, users_df, left_on='client_id', right_on='id', how='left')
# Drop the 'id_y' column from the merged DataFrame
merge_df = merge_df.drop(columns=['id_y'])
```

```
merge_df = pd.merge(merge_df, cards_df, left_on='card_id', right_on='id', how='left')
# Drop the 'id' & 'client_id_y' column from the merged DataFrame
merge_df = merge_df.drop(columns=['id', 'client_id_y'])
```

Figure 5.2.3: Merging Dataset

After merging, transactions are **filtered** to retain only **online transactions made by credit card**. Before filtering, the dataset contained over 8 million transactions, after filtering, only 313,783 transactions remained.

```
merge_df.shape
```

```
(8914963, 38)
```

```
df = merge_df[(merge_df['use_chip'] == 'Online Transaction') & (merge_df['card_type'] == 'Credit')]
```

```
df.shape
```

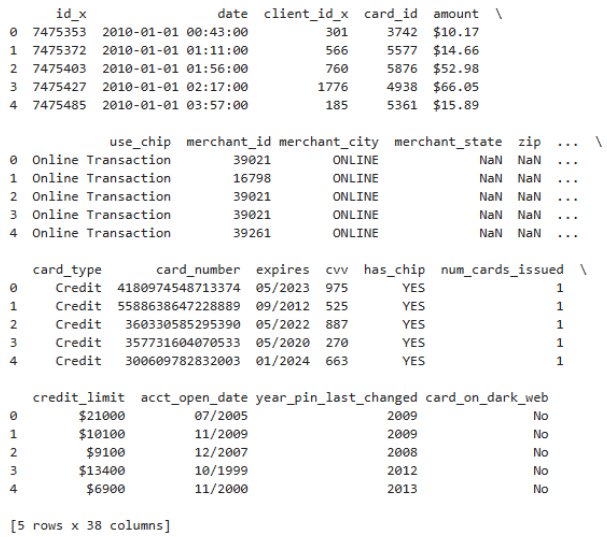
```
(313783, 38)
```

Figure 5.2.4: Dataset Size Before and After Filtering for Online Credit Card Transactions

Next, obtaining a summary of the dataset using *df.info()* and displaying the first 5 rows with *df.head(5)* to **check its structure**. The dataset contains 313,783 entries and 38 columns with a mix of numerical and categorical data. Some columns like *merchant_state* and *zip* are entirely null and are dropped in later a step. While *'errors'* has significant missing values. Financial data is stored as objects, requiring cleaning and conversion to numeric types. Date columns need conversion to datetime format, and categorical features may require encoding. The target variable, *'is_fraud'*, classifies transactions as fraudulent or not.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 313783 entries, 0 to 313782
Data columns (total 38 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   id_x                  313783 non-null  int64
 1   date                  313783 non-null  object
 2   client_id_x           313783 non-null  int64
 3   card_id               313783 non-null  int64
 4   amount                313783 non-null  object
 5   use_chip              313783 non-null  object
 6   merchant_id           313783 non-null  int64
 7   merchant_city         313783 non-null  object
 8   merchant_state        0 non-null       float64
 9   zip                   0 non-null       float64
 10  mcc                   313783 non-null  int64
 11  errors                7569 non-null    object
 12  is_fraud              313783 non-null  object
 13  mcc_name              313783 non-null  object
 14  current_age           313783 non-null  int64
 15  retirement_age        313783 non-null  int64
 16  birth_year            313783 non-null  int64
 17  birth_month           313783 non-null  int64
 18  gender                313783 non-null  object
 19  address               313783 non-null  object
 20  latitude              313783 non-null  float64
 21  longitude             313783 non-null  float64
 22  per_capita_income     313783 non-null  object
 23  yearly_income         313783 non-null  object
 24  total_debt            313783 non-null  object
 25  credit_score          313783 non-null  int64
 26  num_credit_cards      313783 non-null  int64
 27  card_brand            313783 non-null  object
 28  card_type             313783 non-null  object
 29  card_number           313783 non-null  int64
 30  expires               313783 non-null  object
 31  cvv                   313783 non-null  int64
 32  has_chip              313783 non-null  object
 33  num_cards_issued      313783 non-null  int64
 34  credit_limit          313783 non-null  object
 35  acct_open_date        313783 non-null  object
 36  year_pin_last_changed 313783 non-null  int64
 37  card_on_dark_web      313783 non-null  object
dtypes: float64(4), int64(15), object(19)
memory usage: 91.0+ MB
```

Figure 5.2.5: Initial Dataset Summary



```
      id_x                date  client_id_x  card_id  amount  \
0  7475353  2010-01-01 00:43:00          301     3742  $10.17
1  7475372  2010-01-01 01:11:00          566     5577  $14.66
2  7475403  2010-01-01 01:56:00          760     5876  $52.98
3  7475427  2010-01-01 02:17:00         1776     4938  $66.05
4  7475485  2010-01-01 03:57:00          185     5361  $15.89

            use_chip  merchant_id merchant_city  merchant_state  zip  ...  \
0  Online Transaction        39021        ONLINE             NaN  NaN  ...
1  Online Transaction        16798        ONLINE             NaN  NaN  ...
2  Online Transaction        39021        ONLINE             NaN  NaN  ...
3  Online Transaction        39021        ONLINE             NaN  NaN  ...
4  Online Transaction        39261        ONLINE             NaN  NaN  ...

   card_type       card_number  expires  cvv  has_chip  num_cards_issued  \
0     Credit  4180974548713374  05/2023  975       YES                 1
1     Credit  5588638647228889  09/2012  525       YES                 1
2     Credit   360330585295390  05/2022  887       YES                 1
3     Credit   357731604070533  05/2020  270       YES                 1
4     Credit   300609782832003  01/2024  663       YES                 1

   credit_limit  acct_open_date year_pin_last_changed card_on_dark_web
0       $21000         07/2005                  2009               No
1       $10100         11/2009                  2009               No
2        $9100         12/2007                  2008               No
3       $13400         10/1999                  2012               No
4        $6900         11/2000                  2013               No

[5 rows x 38 columns]
```

Figure 5.2.6: First Five Rows of Initial Dataset

Dollar signs and commas in monetary value columns are **removed**, allowing conversion to numerical formats.

	amount	per_capita_income	yearly_income	total_debt	credit_limit
0	10.17	25654.0	52308.0	135319.0	21000.0
1	14.66	22680.0	46244.0	108449.0	10100.0
2	52.98	18420.0	37558.0	72514.0	9100.0
3	66.05	21156.0	43133.0	44263.0	13400.0
4	15.89	15175.0	30942.0	71066.0	6900.0

Figure 5.2.7: Monetary Columns After Removing Dollar Signs and Commas

Date-related columns like *date* and *expires* are **converted into datetime** format to allow further manipulation. For the *expires* column, the expiration date is set to the last day of the month. The ***days_to_expiry*** is computed by calculating the difference between the transaction date (*date*) and the expiration date (*expires*). Additionally, the account age is computed by calculating the difference between the transaction date and the account opening date, converting this into the ***account_age*** column in years. Finally, the age of the PIN, represented by the ***year_pin_last_changed*** column, is calculated by finding the difference between the transaction year and the year of the last PIN change. If the PIN is changed after the transaction date, the age is set to 0.

	date	expires	days_to_expiry	acct_open_date	account_age	pin_age_years
0	2010-01-01 00:43:00	2023-05-31	4897	2005-07-01	4	1
1	2010-01-01 01:11:00	2012-09-30	1002	2009-11-01	0	1
2	2010-01-01 01:56:00	2022-05-31	4532	2007-12-01	2	2
3	2010-01-01 02:17:00	2020-05-31	3802	1999-10-01	10	0
4	2010-01-01 03:57:00	2024-01-31	5142	2000-11-01	9	0

Figure 5.2.8: Feature Extraction from Date-Related Columns

Next, **columns were dropped** based on their uniqueness and relevance. Features like *id_x, client_id_x, card_id, merchant_id,* and *mcc* are removed as they are **unique identifiers**, while *use_chip, merchant_city, card_type,* and *card_on_dark_web* are dropped because they contain **only one unique value**. Columns like *address, latitude,* and *longitude* are excluded due to **high cardinality** and because the variables themselves **do not provide meaningful information**. Features related to card details, such as *card_number* and *cvv*, are removed as they are **not useful for fraud detection**.

While *birth_year* and *birth_month* are dropped because **age had already been extracted**, making them redundant. Columns like *date, expires, acct_open_date,* and *year_pin_last_changed* are removed since they have already been used to **create new features** such as *days_to_expiry, account_age*, and *year_pin_last_changed*. By removing these unnecessary columns, the dataset is streamlined, improving efficiency for fraud detection analysis.

```
df.nunique()
```

```
id_x                    313783
date                    301958
client_id_x                854
card_id                   1345
amount                   34409
use_chip                     1
merchant_id                215
merchant_city                1
mcc                         76
errors                      16
is_fraud                     2
mcc_name                    75
current_age                 73
retirement_age              26
birth_year                  73
birth_month                 12
gender                       2
address                    854
latitude                   579
longitude                  663
per_capita_income          801
yearly_income              847
total_debt                 807
credit_score               239
num_credit_cards             9
card_brand                   4
card_type                    1
card_number               1345
expires                    157
cvv                        741
has_chip                     2
num_cards_issued             3
credit_limit               263
acct_open_date             254
year_pin_last_changed       19
card_on_dark_web             1
days_to_expiry            5464
account_age                 31
pin_age_years               18
dtype: int64
```

Figure 5.2.9: Number of Unique Values for Each Feature

After dropping the specified columns, the dataset is checked for duplicate rows, and **21 duplicate rows are removed**.

```
# Check for duplicate rows
duplicates = df[df.duplicated()]

# Count the number of duplicate rows
duplicate_count = df.duplicated().sum()

print("Number of duplicate rows: ", duplicate_count)
```

```
Number of duplicate rows:  21
```

Figure 5.2.10: Number of Duplicated Rows

The next step is to **remove outliers** using the **Interquartile Range (IQR)** method. It visualized the distribution of the data using **boxplots** to identify potential outliers. Then, for each numeric column, the IQR is calculated, and values outside the range of 1.5 * IQR from the first and third quartiles are marked as outliers.

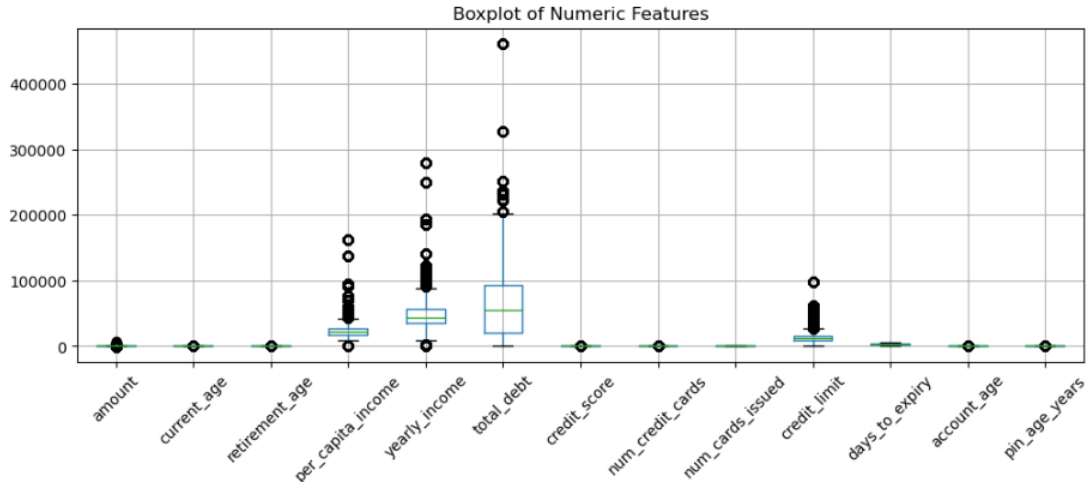


Figure 5.2.11: Boxplots of Numeric Features

The analysis below shows that columns like *amount, current_age, per_capita_income, yearly_income* and *credit_limit* have many outliers, suggesting the presence of extreme values or potential data quality issues, while columns like *num_cards_issued* and *days_to_expiry* have no outliers, indicating more consistency in those features.

```
Number of outliers in each column:
amount               29631
current_age          10459
retirement_age       9240
per_capita_income    19129
yearly_income        17535
total_debt           1436
credit_score         6804
num_credit_cards     236
num_cards_issued     0
credit_limit         10056
days_to_expiry       0
account_age          1792
pin_age_years        538
```

Figure 5.2.12: Number of Outliers in Each Column

The outliers are split into fraud and non-fraud cases. **Fraudulent outliers** are **kept**, while **non-fraudulent outliers** are **removed** from the dataset to retain only relevant fraud data.

In the updated analysis below, 1,283 fraud-related outliers are kept, as they could represent important, extreme fraudulent cases important for fraud detection. 105,573 non-fraudulent outliers are removed to eliminate irrelevant extreme values that could distort further analysis. As a result, the dataset was **reduced to 242,680 records**. Overall, this process helps refine the dataset by retaining important fraud data while removing non-relevant outliers.

```
Total outliers kept for fraud cases: 1283
Total outliers removed for non-fraud cases: 105573
New dataset size after removing outliers: 242680
```

Figure 5.2.13: Outlier Handling Summary

Next, **one-hot encoding** is applied to categorical columns such as *errors* and *card_brand*. **Binary encoding** is then used for columns like *is_fraud*, *gender* and *has_chip*.

```
# List of all possible error types
error_types = ['Bad Card Number', 'Insufficient Balance', 'Bad Expiration', 'Bad CVV', 'Technical Glitch']

# Create binary columns for each error type and fill with 1 if error is present
for error in error_types:
    df[error] = df['errors'].apply(lambda x: 1 if isinstance(x, str) and error in x else 0)

# Drop the original 'error' column
df = df.drop(columns=['errors'])
```

```
# Apply one-hot encoding to 'card_brand' column
df = pd.get_dummies(df, columns=['card_brand'], dtype=int)
```

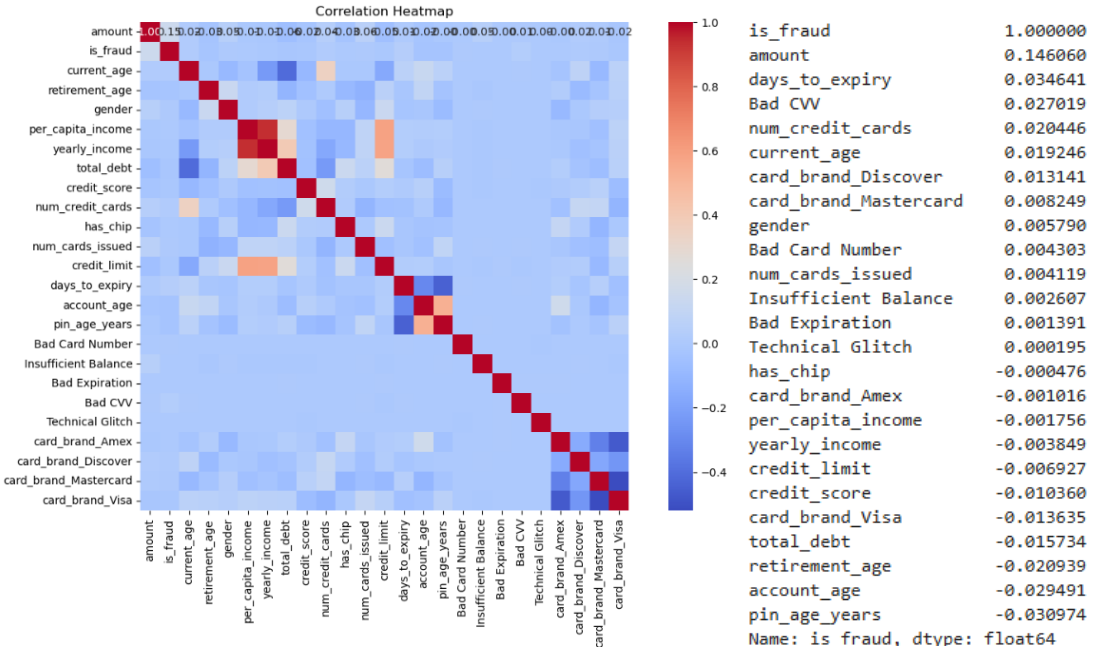
```
# Replace 'Yes' with 1 and 'No' with 0 in the 'is_fraud_1' column
df['is_fraud'] = df['is_fraud'].replace({'Yes': 1, 'No': 0})
```

```
# Replace 'Male' with 1 and 'Female' with 0 in the 'gender' column
df['gender'] = df['gender'].replace({'Male': 1, 'Female': 0})
```

```
# Replace 'Yes' with 1 and 'No' with 0 in the 'has_chip' column
df['has_chip'] = df['has_chip'].replace({'YES': 1, 'NO': 0})
```

Figure 5.2.14: One-Hot Encoding and Binary Encoding for Categorical Columns

After using **heatmap** and **correlation analysis** to identify potential relationships between the features and the target variable (*is_fraud*), it proves that many of the correlations are weak or near zero. The presence of a lot of blue in the heatmap as shown in *Figure 4.2.15*, typically **indicates weak or insignificant correlations** between variables.




```
is_fraud                1.000000
amount                  0.146060
days_to_expiry          0.034641
Bad CVV                 0.027019
num_credit_cards        0.020446
current_age             0.019246
card_brand_Discover     0.013141
card_brand_Mastercard   0.008249
gender                  0.005790
Bad Card Number         0.004303
num_cards_issued        0.004119
Insufficient Balance    0.002607
Bad Expiration          0.001391
Technical Glitch        0.000195
has_chip               -0.000476
card_brand_Amex        -0.001016
per_capita_income      -0.001756
yearly_income          -0.003849
credit_limit           -0.006927
credit_score           -0.010360
card_brand_Visa        -0.013635
total_debt             -0.015734
retirement_age         -0.020939
account_age            -0.029491
pin_age_years          -0.030974
Name: is_fraud, dtype: float64
```

Figure 5.2.15: Heatmap and Correlation Table for *is_fraud*

The correlation analysis between target variable and other numerical features reveals several key insights. The highest correlation is with amount (0.14606), suggesting a weak positive relationship. Other features like *days_to_expiry, Bad CVV* and *num_credit_cards* had very weak positive correlations, while *total debt, retirement_age, account_age* and *pin_age_years* show weak negative correlations. Many variables, such as *Technical Glitch* and *has_chip*, exhibit minimal correlations, indicating limited relevance for fraud prediction.

Given the weak correlations, the current dataset may not provide strong predictive power for fraud detection because most features do not show meaningful relationships with target variable. This may make it difficult for the models to differentiate fraudulent from legitimate transactions. This limitation prompts the need to **explore a different dataset** with stronger relationships for better model performance.

5.3 Final Dataset

5.3.1 Data Selection

In this project, the final dataset chosen is the "Credit Card Transactions Fraud Detection Dataset" from Kaggle. Unlike the **previous dataset, which has low correlation and limited usage**, this dataset is considered **more reliable** due to its widespread use by other researchers. It contains both fraudulent and non-fraudulent transactions recorded between January 1, 2019, and December 31, 2020. It provides a rich set of features, including transaction date and time, amount, merchant details, product categories, job, cardholder information and geographical data as shown in *Table 4.3.1*.

Feature	**Description**
trans_date_trans_time	Date and time of the transaction
cc_num	Credit card number used for the transaction
merchant	Name of the merchant where the transaction occurred
category	Type of merchant or business category
amt	Amount of money spent in the transaction
first	First name of the cardholder
last	Last name of the cardholder
gender	Gender of the cardholder (Male or Female)
street	Street address of the cardholder
city	City of the cardholder
state	State of the cardholder
zip	ZIP code of the cardholder's address
lat	Latitude of the cardholder's location
long	Longitude of the cardholder's location
city_pop	Population of the cardholder's city
job	Job title of the cardholder
dob	Date of birth of the cardholder
trans_num	Unique transaction ID
unix_time	Transaction time in Unix timestamp format
merch_lat	Latitude of the merchant's location

merch_long	Longitude of the merchant's location
is_fraud	Target variable showing if the transaction is fraud (1) or non-fraud (0)

Table 5.3.1: Feature Description of Credit Card Transactions Fraud Detection Dataset

5.3.2 EDA and Data Cleaning

1. Handling null value and duplicate

Since there were no null values and duplicate, then skipped to the step of removing irrelevant columns.

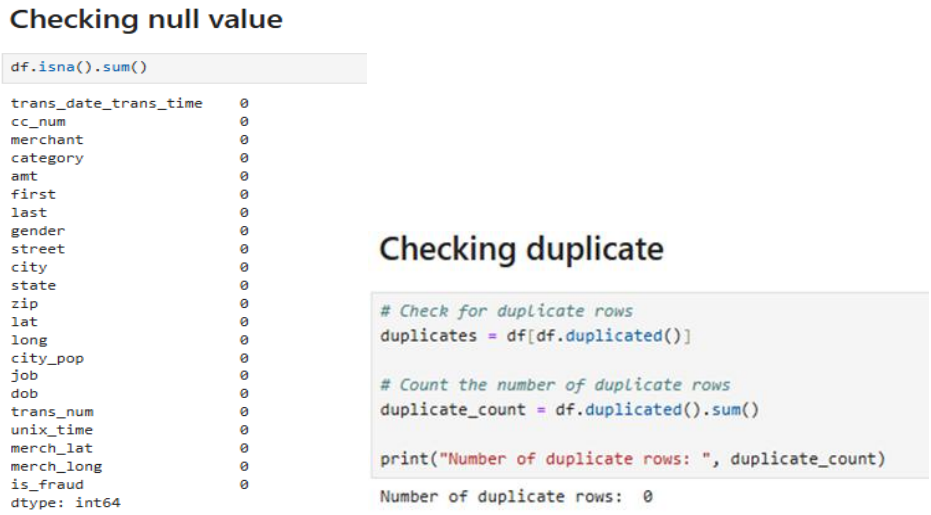


Figure 5.3.1: Null Values and Duplicates Check in the Dataset

2. Remove irrelevant columns

Features like *trans_num, first*, and *last* were removed because they contained transaction or personal identifiers that do not contribute meaningfully to fraud detection. Features like *street, city, state,* and *zip* were removed since location information is already represented by *lat, long, merch_lat* and *merch_long*, making them redundant.

```
df.drop(columns=['trans_num','first','last'],inplace=True)
```

```
df.drop(columns=['street','city','state','zip'],inplace=True)
```

Figure 5.3.2: Drop Irrelevant Columns

3. Remove outlier

To detect outliers in the dataset, the Interquartile Range (IQR) method is applied to numeric features. Each feature is analysed separately, and the results are shown in the boxplots below.

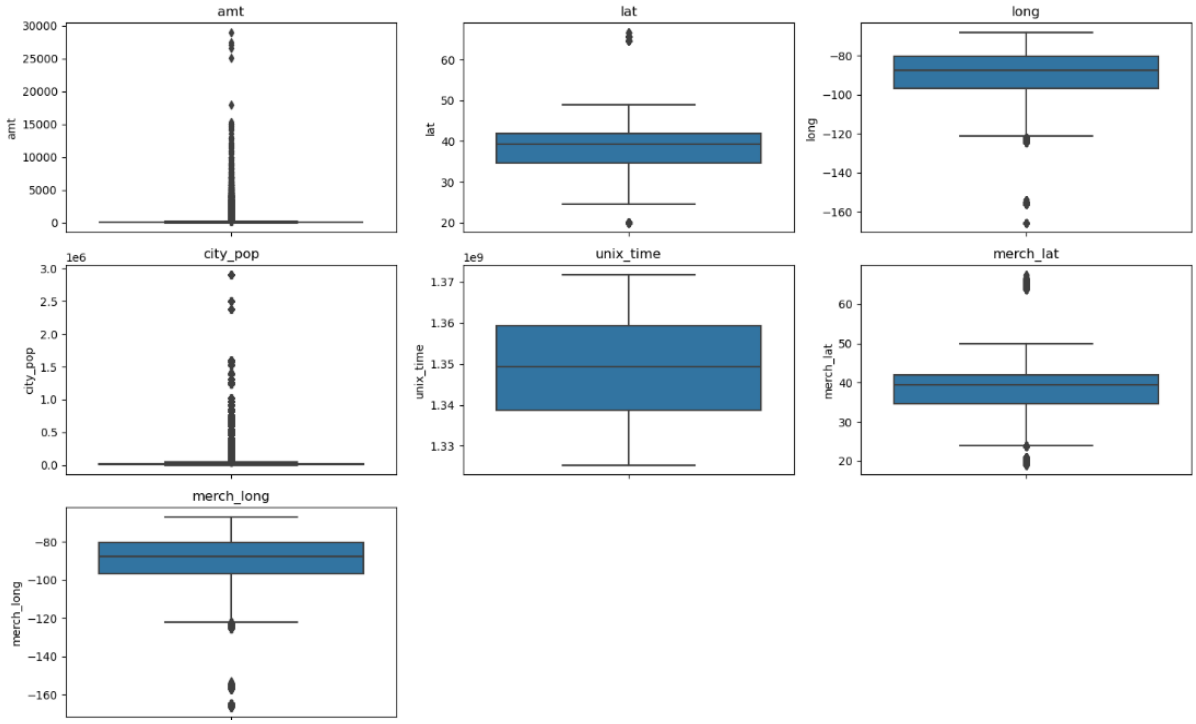


Figure 5.3.3: Boxplots for Numerical Features

The outlier detection step identifies a significant number of anomalies across various features, with a total of **328,615 rows containing at least one outlier**.

```
Total rows containing at least one outlier (IQR method): 328615

Number of outliers per column:
amt            67290
lat             4679
long           49922
city_pop      242674
unix_time          0
merch_lat       4967
merch_long     41994
dtype: int64
```

Figure 5.3.4: Total Rows with Outliers and Outliers per Column

The **transaction amount (*amt*)** has 67,290 outliers, indicating a wide range of transaction values with extreme cases, primarily involving high transaction amounts, as observed in the boxplot.

Geographical features, such as **latitude (*lat*), longitude (*long*), merchant latitude (merch_*lat*)** and **merchant longitude (merch_*long*)**, show 4,679, 49,922, 4,967 and 41,994 outliers, respectively. The boxplots show that latitude has outliers at both ends of the range, while longitude outliers are mostly on the lower extreme, for both customer and merchant locations.

City population (city_pop) has the highest number of outliers at 242,674, likely due to the extreme variations in population sizes across different cities.

Transaction time (unix_time) does not show any outliers, indicating a uniform distribution of transactions over time. These findings highlight the need for proper handling of extreme values to improve the fraud detection model's performance.

```
Feature: amt
Total fraudulent outliers: 5705
------------------------------
Feature: lat
Total fraudulent outliers: 43
------------------------------
Feature: long
Total fraudulent outliers: 298
------------------------------
Feature: city_pop
Total fraudulent outliers: 1434
------------------------------
Feature: unix_time
Total fraudulent outliers: 0
------------------------------
Feature: merch_lat
Total fraudulent outliers: 46
------------------------------
Feature: merch_long
Total fraudulent outliers: 261
------------------------------

Total rows with fraudulent outliers:  6081
Total rows with non-fraudulent outliers:  322534
```

Figure 5.3.5: Summary of Fraudulent Outliers Across Features

To check whether outliers contributed to fraudulent transactions, the dataset is split into fraudulent and non-fraudulent transactions. The analysis shows **6,081 fraudulent transactions with outliers** and **322,534 non-fraudulent transactions with outliers**. A total of 242,674 outliers were detected in the city population, but only 1,434 are linked to fraud. The **transaction amount** has the **highest number of fraudulent outliers**, totalling **5,705**, followed by city population (1,434), longitude (298), merchant longitude (261), merchant latitude (46) and latitude (43). Since most outliers are found in non-fraudulent transactions, this suggests that extreme values alone do not necessarily indicate fraud.

Since fraudulent transactions with outliers may contain valuable fraud patterns, **only non-fraudulent outliers were removed** from the dataset. This step aimed to reduce noise while retaining important fraud indicators. After removing these non-fraudulent outliers, the **dataset size** decreased from 1,296,675 to **974,141**.

```
Total rows after removing non-fraudulent outliers: 974141
```

Figure 5.3.6: Outlier Handling Summary

5.3.3 EDA and Data Visualization

1. Fraud Count and Rate

To understand the **distribution of fraudulent transactions** within the dataset, an analysis of fraud occurrence rates and their proportion to non-fraudulent transactions.

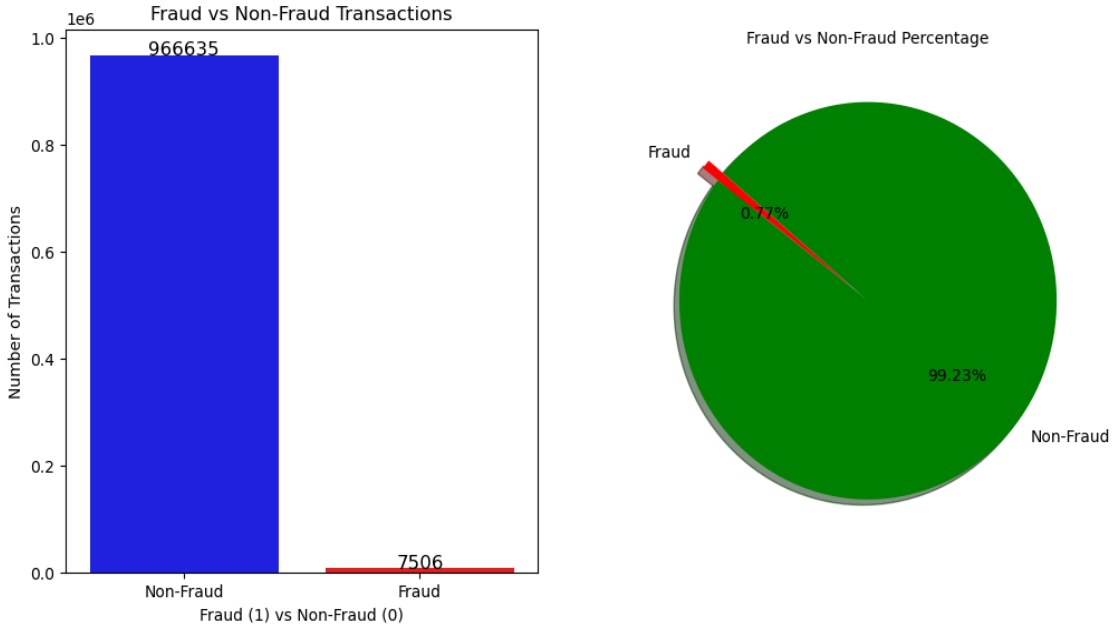


Figure 5.3.7: Fraud vs Non-Fraud Transactions and Percentage Distribution

After removing outliers, the dataset still displays a serious class imbalance, with **non-fraudulent transaction** making up **99.23% (966,635 records)** and fraudulent transactions only **0.77% (7,506 records)**. This large contrast is visually showed in the **bar chart**, where

fraud cases are barely visible compared to non-fraud transactions, and in the **pie chart**, which shows that fraud makes up less than **1%** of all transactions.

This imbalance creates a challenge for machine learning models, as they may **bias towards the majority class (non-fraud)**, resulting in high accuracy but poor fraud detection. To solve this, techniques like **SMOTE, oversampling and under-sampling** will be applied in the later steps to improve model performance.

2. Transaction Amount

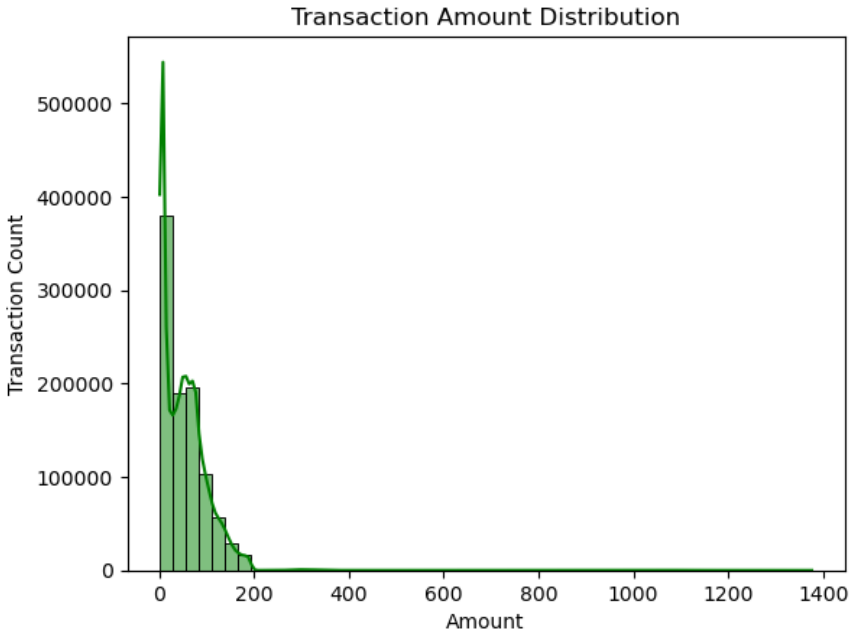


Figure 5.3.8: Transaction Amount Distribution

The **transaction amount distribution** plot shows that **most transactions are small amounts**, with focus on transactions **below $200**. This distribution is highly right skewed, meaning a few transactions involve significantly larger amounts.

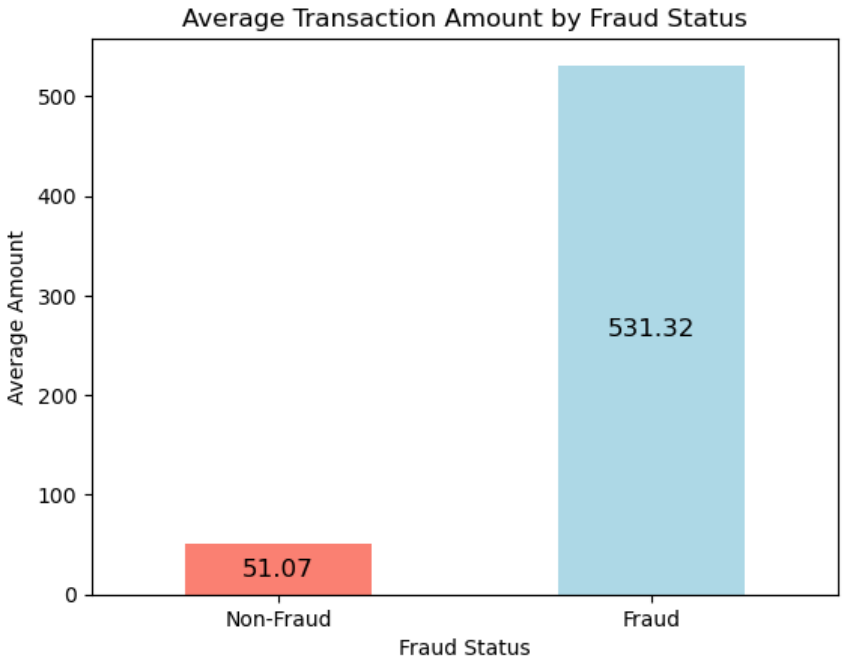


Figure 5.3.9: Average Transaction Amount by Fraud and Non-Fraud

The bar chart comparing **average transaction amounts** between fraudulent and non-fraudulent transactions. Average amount for **non-fraudulent transactions** is **$51.07**, while for **fraudulent transactions** is **$531.32**. This means that fraudulent transactions normally involve higher amounts compared to legitimate transactions.

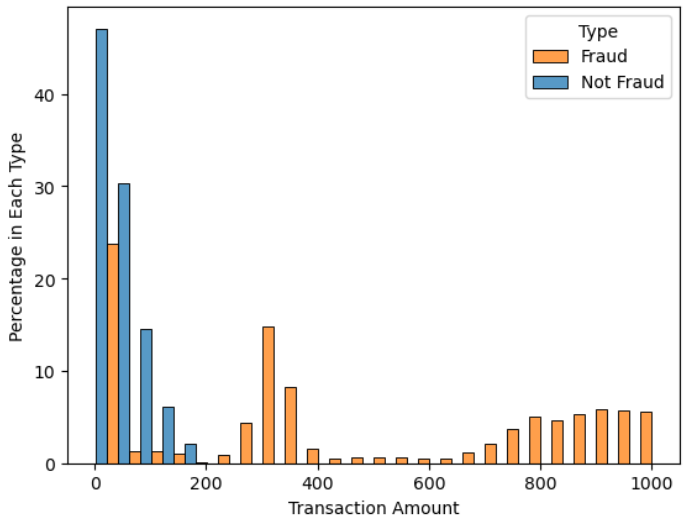


Figure 5.3.10: Percentage Distribution of Fraud and Non-Fraud by Transaction Amount

This histogram comparing **fraud and non-fraud rate** across different transaction amounts further supports previous findings. **Lower transaction amounts** have a **higher non-fraud rate**, while the **fraud rate increases as transaction amounts rise**, especially in the range from $200 to $1,000. This pattern suggests that transaction amount is an important feature for fraud detection models. Higher amount transactions should be closely monitored, as they are more likely to be fraudulent.

3. Gender

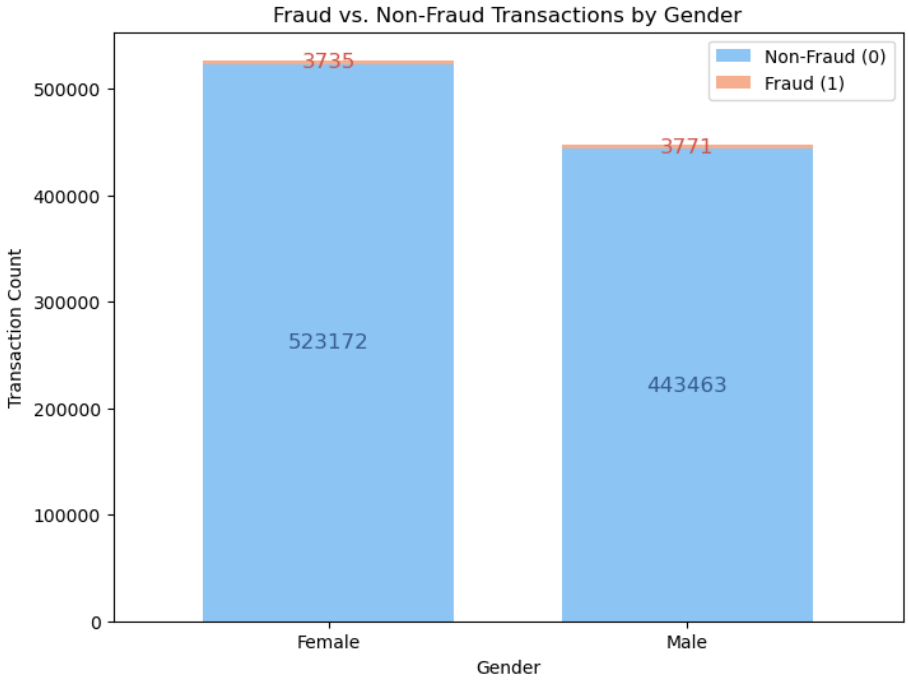


Figure 5.3.11: Fraud and Non-Fraud Transactions by Gender

The analysis of fraud and non-fraud transactions by gender reveals that the total number of transactions is higher for females (**523,172**) compared to males (**443,463**). However, the number of **fraudulent transactions is nearly same** for both genders, with **3,735 fraud cases among females** and **3,771 among males**.

Figure 5.3.12: Fraud Ratio by Gender

Even though the number of fraud cases is almost the same for both genders, the fraud ratio is different because the total number of transactions varies. **Males have a higher fraud ratio of 0.84%**, while **females have a slightly lower fraud ratio of 0.71%.** This suggests that compared to their total transactions, fraud is more common among males than females.

4. Category

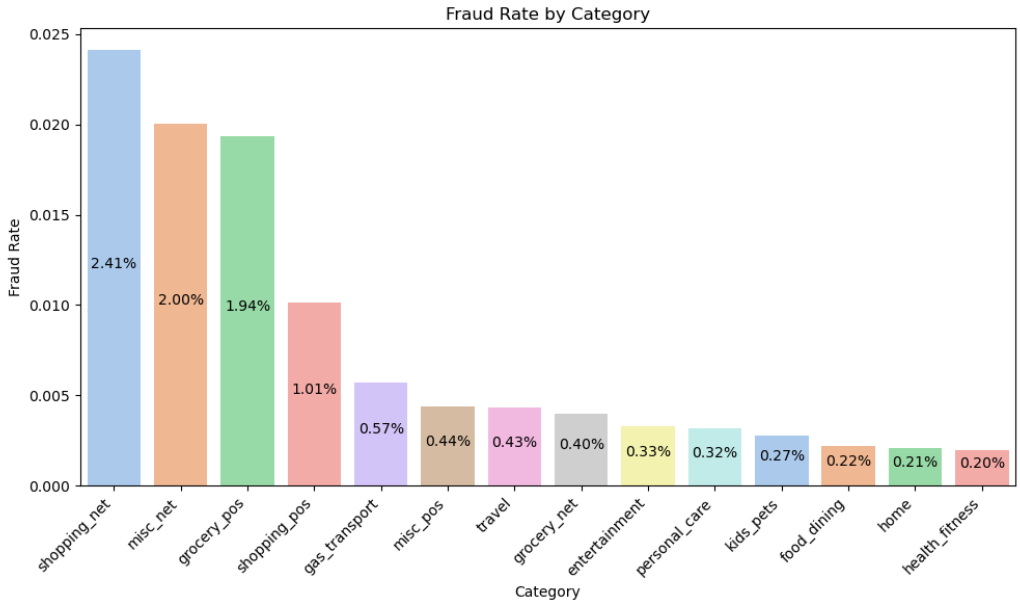


Figure 5.3.13: Fraud Rate by Category

This chart visualises the fraud rate across different transaction categories. The **highest fraud rate is observed in 'shopping_net' (2.41%)**, followed by 'misc_net' (2.00%) and 'grocery_pos' (1.94%), indicating that these three transaction categories are more vulnerable to fraud. Moderate fraud rates are found in categories like 'shopping_pos' (1.01%), 'gas_transport' (0.57%), 'misc_pos' (0.44%), 'travel' (0.43%) and 'grocery_net' (0.40%), suggesting that fraudsters also target essential services and frequent transactions. In contrast, categories such as entertainment, personal care, kids/pets, foods/dining, home and health/fitness exhibit fraud rate below 0.35%.

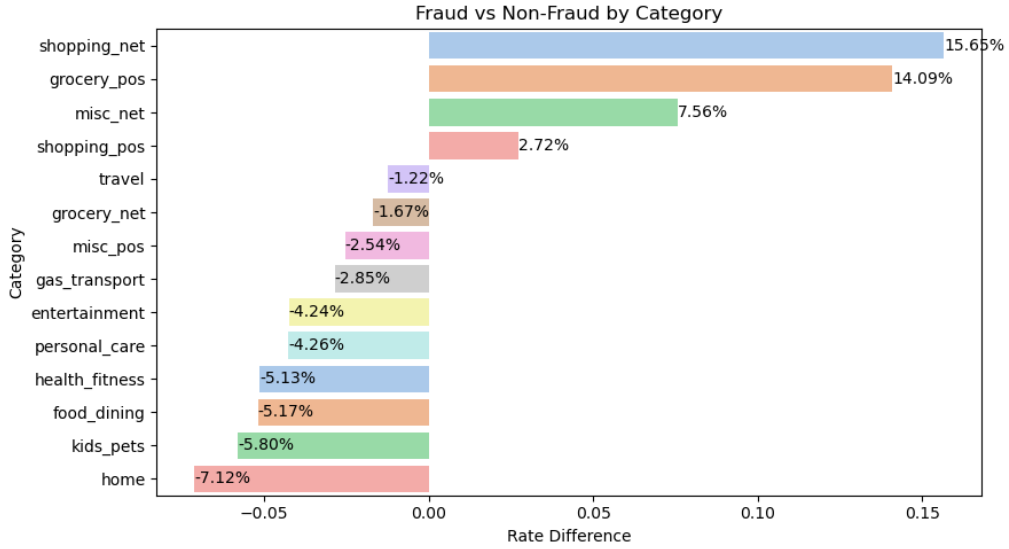


Figure 5.3.14: Rate Difference between Fraud and Non-Fraud Transactions by Category

This chart visualises the difference in fraud rates compared to non-fraud rates across various transaction categories. Categories with positive values indicate a higher probability of fraud, while negative values suggest a lower fraud risk. Category **'shopping_net'** has the **highest rate difference of 15.65%**, followed by **'grocery_pos' (+14.09%)**. Other risky categories include '**misc_net'** and **'shopping_pos'** have positive rate differences of **7.56%** and **2.72%** respectively.

The remaining categories have negative rate differences, indicating that fraud is less common in these types of transactions. The **home** category shows the lowest fraud difference at **-7.12%**, followed by **kids/pets (-5.80%)**, **food/dining (-5.17%)**, and **health/fitness (-5.13%)**. Other categories such as **personal care (-4.26%)** and **entertainment (-4.24%)** also have lower fraud risk, possibly due to transaction verification processes or lower fraud attractiveness.

The dataset distinguishes between online (*Card Not Present, CNP*) and in-store (*Card Present, CP*) transactions through its category labels. Categories ending with "_net" (e.g., shopping_net, grocery_net) are CNP, while those ending with "_pos" (e.g., shopping_pos, grocery_pos) are CP. Since CNP transactions lack physical verification, they are more exposed to fraud and are the main focus of this study.

```
category
grocery_pos       1743
shopping_net      1713
misc_net           915
shopping_pos       843
gas_transport      618
misc_pos           250
kids_pets          239
entertainment      233
personal_care      220
home               198
food_dining        151
grocery_net        134
health_fitness     133
travel             116
Name: count, dtype: int64

Percentage of top 4 categories with the most fraud cases: 69.46%
```

Figure 5.3.15: Fraud Count by Category

This analysis shows that '**grocery_pos'** has the **highest fraud count (1,743 cases)**, followed by *shopping_net* (1,713), *misc_net* (915) and *shopping_pos* (843). However, a high fraud count does not necessarily mean a high fraud rate. Although *grocery_pos* has the highest fraud count, '**shopping_net'** has the **highest fraud rate** (2.41%).

Additionally, the **top 4 fraud-heavy categories** (*grocery_pos, shopping_net, misc_net,* and *shopping_pos*) make up **nearly 70% of all fraud cases**. This means that fraud is mostly happening in just a few transaction types, making them the most critical areas to focus on in fraud detection. The **remaining categories are grouped as 'other'**, as they contribute less to overall fraud.

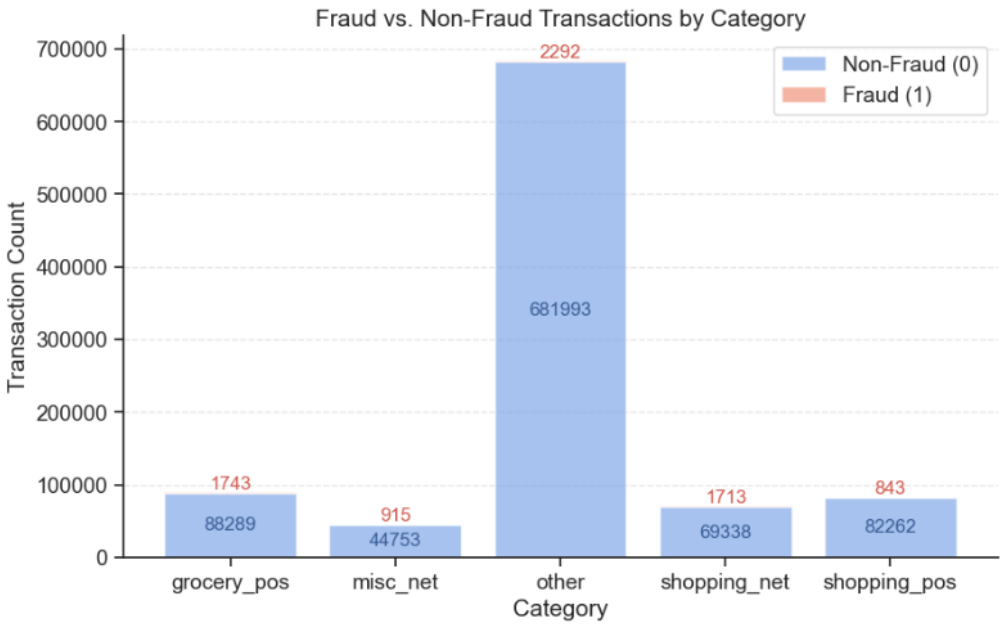


Figure 5.3.16: Fraud and Non-Fraud Transactions by Category

After grouping the transaction categories, the bar chart shows the majority transactions belong to the 'other' category, with 681,993 non-fraudulent transactions and 2,292 fraudulent ones. Other categories like *grocery_pos, misc_net, shopping_net* and *shopping_pos* have significantly fewer transactions in total, but the number of fraudulent transactions is relatively evenly distributed across these categories.

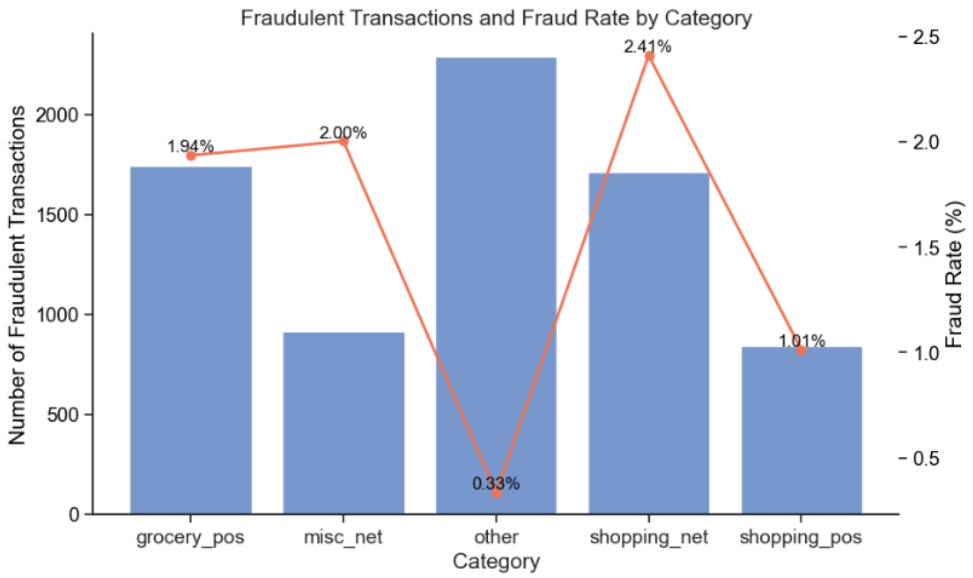


Figure 5.3.17: Fraudulent Transactions and Fraud Rate by Category

This chart provides deeper insights by showing the fraud rate for each transaction category. Although the *'other'* category has the highest number of fraudulent transactions, its **fraud rate is only 0.33%**, meaning fraud is relatively rare compared to the total number of transactions. On the other hand, *shopping_net* has the **highest fraud rate of 2.41%**, indicating that even though there are fewer fraud cases in total transactions, fraudulent activity is more concentrated within this category. Similarly, *misc_net* (2.00%) and *grocery_pos* (1.94%) have high fraud rates, indicating a high concentration of fraudulent transactions. *Shopping_pos* (1.01%) has a lower fraud rate but remains higher than the *'other'* category.

5. Merchant

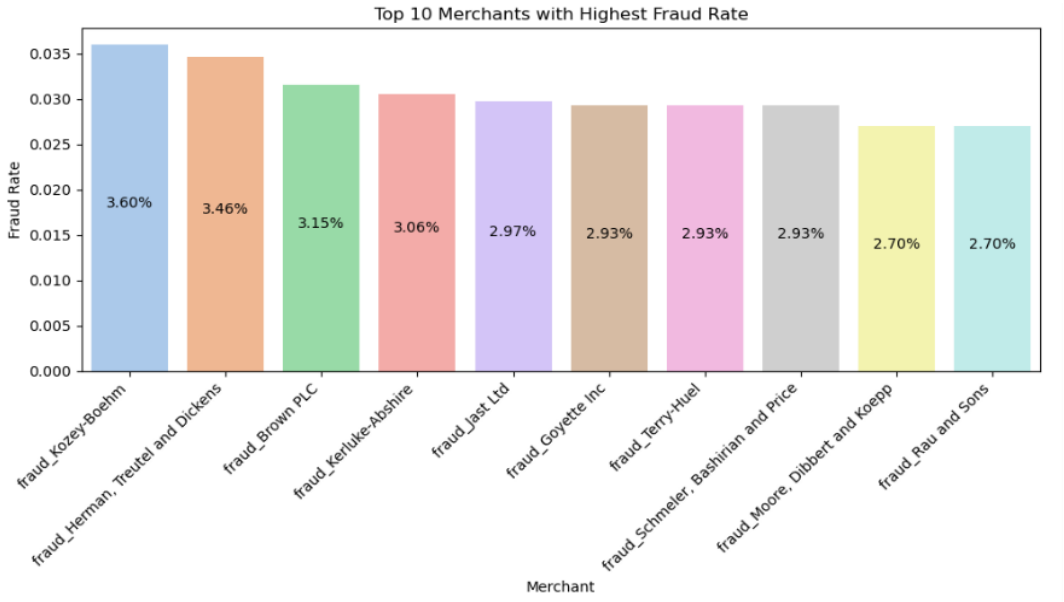


Figure 5.3.18: Top 10 Merchants with the Highest Fraud Rate

The chart shows the **top 10 merchants** with the highest fraud rates, which **range from 2.7% to 3.6%**. The merchant **"fraud_Kozey-Boehm"** has the highest fraud rate at **3.6%**, followed closely by **"fraud_Herman, Treutel and Dickens"** at **3.46%**. These findings suggest that some merchants are more susceptible to fraudulent transactions, either due to weaknesses in their fraud prevention mechanisms or because they are targeted more frequently by fraudsters. Notably, only four merchants have the fraud rates exceeding 3%.

```
# Calculate the value counts for the 'merchant' column for rows where is_fraud = 1
fraud_merchant_counts = df[df['is_fraud'] == 1]['merchant'].value_counts()
fraud_merchant_counts
```

```
merchant
fraud_Rau and Sons        49
fraud_Cormier LLC         48
fraud_Kozey-Boehm         48
fraud_Doyle Ltd           47
fraud_Vandervort-Funk     47
                          ..
fraud_Kuphal-Toy           1
fraud_Eichmann-Kilback     1
fraud_Lynch-Mohr           1
fraud_Tillman LLC          1
fraud_Hills-Olson          1
Name: count, Length: 679, dtype: int64
```

Figure 5.3.19: Fraud Count by Merchant

To complement these findings, the total fraud count per merchant is analysed. The results show that fraud occurs across **679 merchants**. The merchant with the highest fraud count is **"fraud_Rau and Sons"** with 49 fraud cases, followed by **"fraud_Cormier LLC"** and **"fraud_Kozey-Boehm"**, each with 48 cases. However, when evaluating fraud risk, the fraud rate is often more meaningful than the fraud counts because it handles the overall transaction volume at each merchant. A merchant with a high fraud counts but a low fraud rate may simply process a large number of transactions, whereas a high fraud rate indicates a greater likelihood of fraud occurring.

```
# Calculate the sum of the counts for the top 5 merchant
top_5_fraud_sum = fraud_merchant_counts.head(5).sum()
fraud_percentage = (top_5_fraud_sum / fraud_merchant_counts.sum()) * 100
print(f"Percentage of top 5 merchant with the most fraud cases: {fraud_percentage:.2f}%")
```

```
Percentage of top 5 merchant with the most fraud cases: 3.18%
```

Figure 5.3.20: Percentage of Fraud Count for the Top 5 Merchant

Further analysis reveals that the **top five merchants** in term of fraud count only make up **3.18%** of total fraud cases, indicating that fraudulent transactions are widely distributed across a large number of merchants. Since fraud is not highly focused within a small number of merchants, it means that the merchant feature may not a strong indicator of fraud. As a result, this feature is **removed** from the dataset to simplify the model and prevent it from learning patterns that might not generalize well.

6. Job

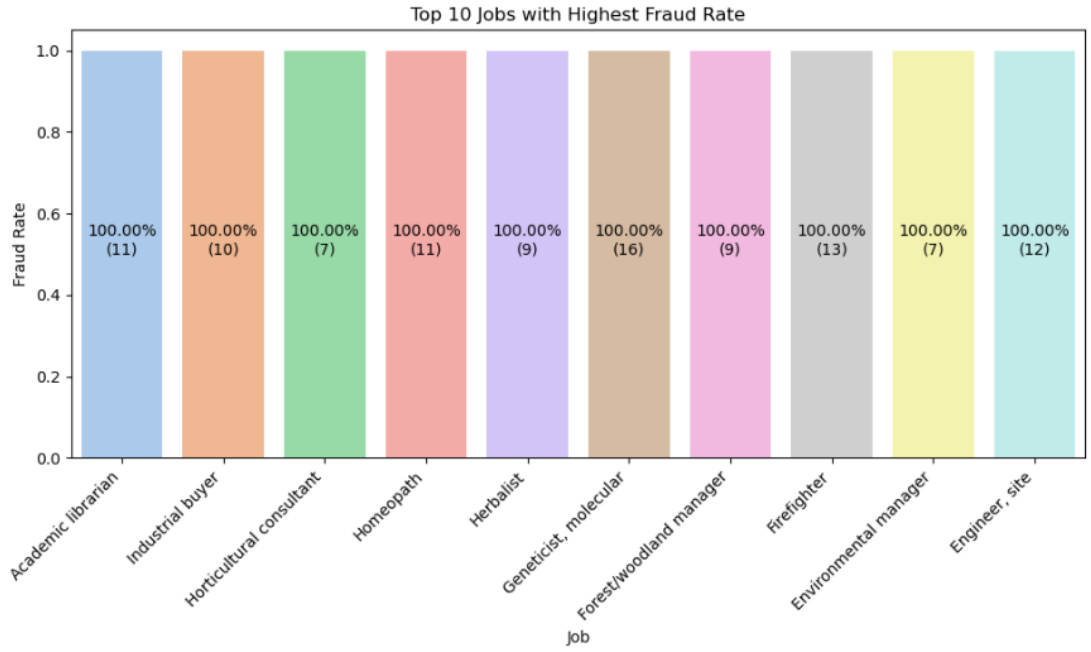


Figure 5.3.21: Top 10 Jobs with the Highest Fraud Rate

The chart highlights the **top 10 jobs** with the highest fraud rates, all of them have a **fraud rate of 100%**. However, the fraud count among these jobs varies, ranging from 7 to 16 cases. This indicates that while every transaction recorded under these job titles was fraudulent, the **total number of fraudulent transactions per job remains relatively low**.

```
# Count jobs where fraud rate is 100%
fraudulent_jobs_count = (job_fraud_rate['fraud_rate'] == 1.0).sum()
print(f"Number of jobs with 100% fraud rate: {fraudulent_jobs_count}")
```

```
Number of jobs with 100% fraud rate: 68
```

```
# Calculate the value counts for the 'job' column for rows where 'is_fraud' equals 1
fraud_job_counts = df[df['is_fraud'] == 1]['job'].value_counts()
fraud_job_counts
```

```
job
Materials engineer                    62
Trading standards officer             56
Naval architect                       53
Exhibition designer                   51
Surveyor, land/geomatics              50
                                      ..
Statistician                           3
Health physicist                       3
Chartered loss adjuster                3
English as a second language teacher   2
Contractor                             2
Name: count, Length: 443, dtype: int64
```

Figure 5.3.22: Jobs with 100% Fraud Rate and Their Counts

The analysis further shows that there are **68 job categories with a 100% fraud rate**, meaning every recorded transaction under these jobs is fraudulent. However, as seen in the fraud count distribution, fraudulent transactions are **spread across 443 different job titles**, with the **highest fraud count** recorded under **"Materials engineer" at 62 cases**, followed by "Trading standards officer" (56) and "Naval architect" (53).

Many job categories show a 100% fraud rate but with low fraud counts. Such findings suggests that these job categories might not be truly high risk, it is likely due to sample size bias, leading to misleading fraud rates. Since fraud cases are widely distributed, job titles alone may not be strong fraud indicators.

```
# Calculate the sum of the counts for the top 10 jobs
top_fraud_sum = fraud_job_counts.head(10).sum()
fraud_percentage = (top_fraud_sum / fraud_category_counts.sum()) * 100
print(f"Percentage of top 10 categories with the most fraud cases: {fraud_percentage:.2f}%")
```

```
Percentage of top 10 categories with the most fraud cases: 6.77%
```

Figure 5.3.23: Percentage of Fraud Count for the Top 10 Jobs

Since the top 10 categories with the most fraud cases make up only 6.77% of the total fraud cases, this suggests that job is not a strong fraud indicator again. The decision to **remove the *job* column** is justified to reduce noise and prevent the model from learning patterns that may not generalize well.

7. Age

The *trans_date_trans_time* and *dob* features are converted to datetime format, then age is extracted by calculating their difference in years.

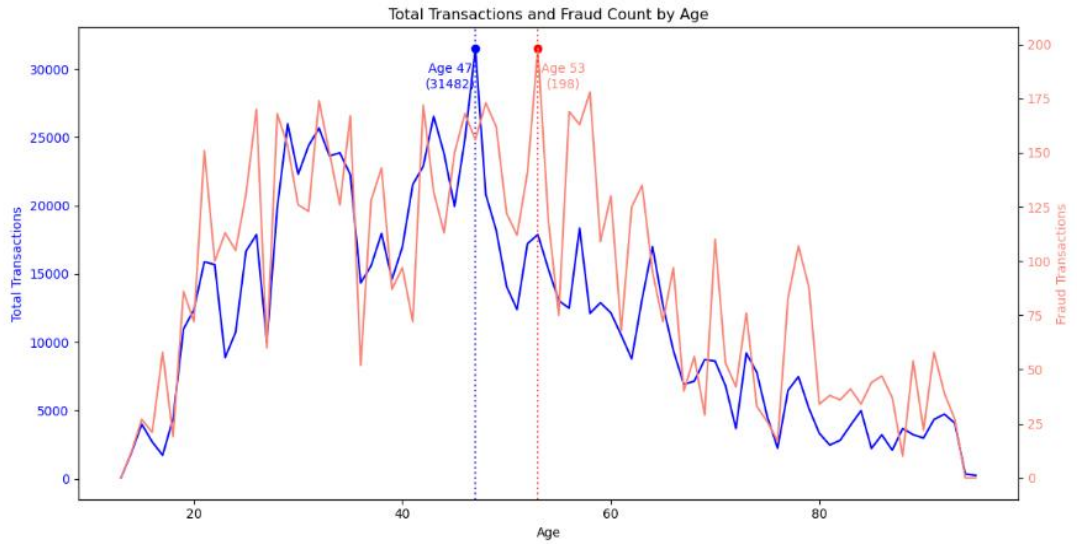


Figure 5.3.24: Total Transactions and Fraud Count by Age

The **highest total transactions** occur at **age 47**, with **31,482 transactions**, while the **highest fraud count** is observed at **age 53**, with **198 fraudulent transactions**. The **fraud transaction trends fluctuate**, often crossing the total transaction line after age 30 and fraudulent transactions tend to be relatively higher beyond age 47.

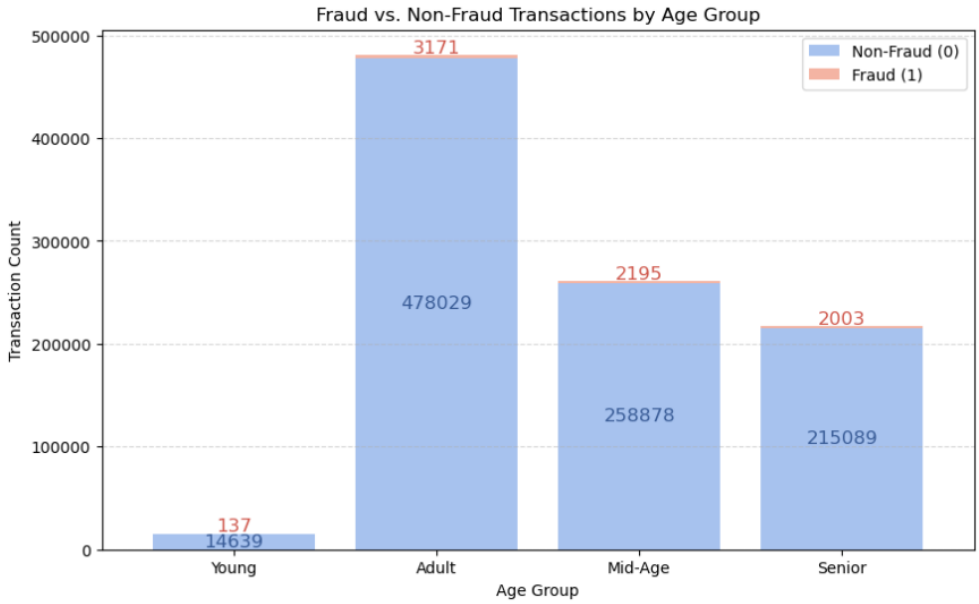


Figure 5.3.25: Fraud and Non-Fraud Transactions by Age Group

Age is categorised into four group: *Young* (0-18), *Adult* (19-44), *Mid-Age* (45-50) and *Senior* (61+), allowing for a more detailed analysis of fraud trends across different life stages. *Adult* group dominates in total transactions, with 478,029 non-fraud and 3,171 fraud cases. The *Mid-Age* and *Senior* groups, though having fewer transactions, still show significant fraud cases, with 2,195 and 2,003 fraud cases respectively. The *Young* group has the lowest non-fraud and fraud count at 14,639 and 137 respectively.

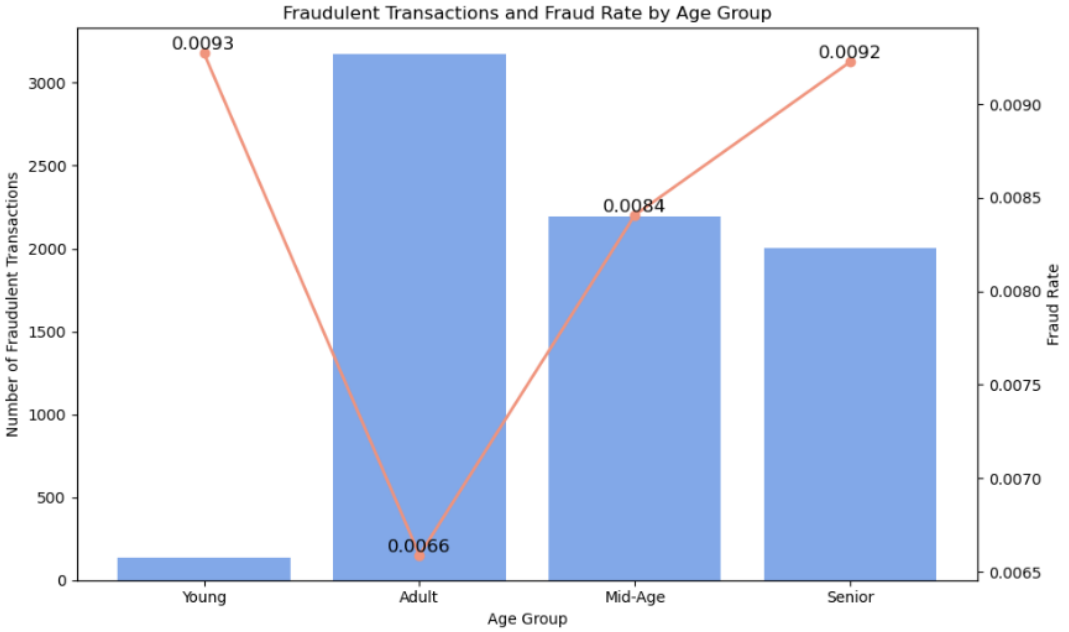


Figure 5.3.26: Fraudulent Transactions and Fraud rate by Age Group

Further analysis of fraud rates reveals that the *Young* group has the highest fraud rate (0.93%), closely followed by Seniors (0.92%). Despite having the highest transaction volume, *Adult* group experiences the lowest fraud rate (0.66%), while the *Mid-Age* group falls in between at 0.84%. Both the Young and Senior groups show an increasing trend in fraudulent transactions, highlighting their vulnerabilities. Therefore, fraud prevention efforts should focus on these groups, even though *Adult* group contributes the highest number of transactions.

8. Hour

The analysis of fraud transactions by hour reveals significant trends in transaction activity and fraudulent behaviour. First, *trans_date_trans_time* is converted into hourly data to examine patterns throughout the day.

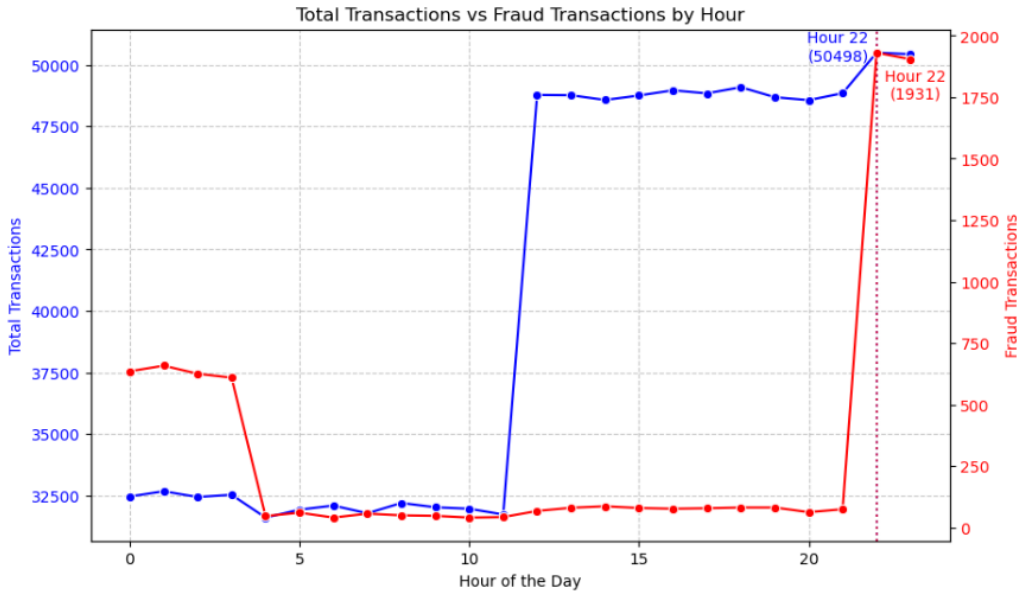


Figure 5.3.27: Total Transactions and Fraud Transactions by Hour

The **peak transaction hour** is identified as **10 PM (Hour 22)** with **50,498 transactions**, which also overlap with the **highest number of fraud cases** at **1,931**. This suggests that fraudsters may be exploiting the high transaction volume during late hours. Fraud activity remains significantly **higher between 10 PM and 3 AM** compared to other hours.

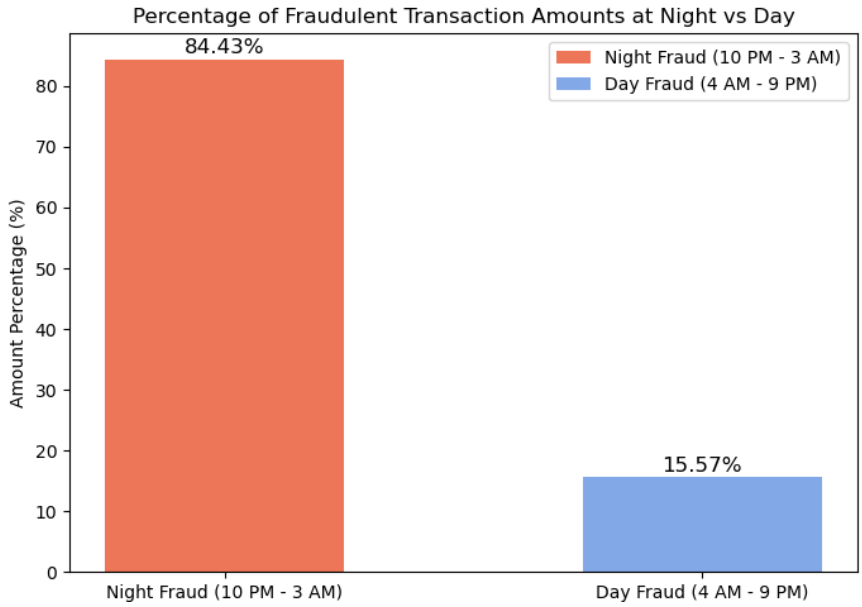


Figure 5.3.28: Percentage of Fraudulent Transaction Amount by Night and Day

Further analysis separated fraud into **night (10 PM - 3 AM)** and **day (4 AM - 9 PM)**, showing that **84.43% of fraudulent transaction** amounts occur at **night**, compared to only **15.57%** during the **day**. This indicates that nighttime is a high-risk period for fraudulent activities, likely due to reduced monitoring and delayed detection. To mitigate these risks, financial institutions and e-commerce platforms should enhance fraud detection measures, particularly during late hours.

```
# Convert hour to is_night (1 for 10 PM - 3 AM, otherwise 0)
df['is_night'] = df['hour'].apply(lambda x: 1 if 22 <= x or x <= 3 else 0)
```

Figure 5.3.29: Conversion of *is_night* Feature

For machine learning purposes, the *is_night* feature is created to indicate whether a transaction occurred during high-risk nighttime hours (10 PM - 3 AM). A value of 1 represents nighttime transactions, while 0 represents transactions during the rest of the day.

9. Day of Week

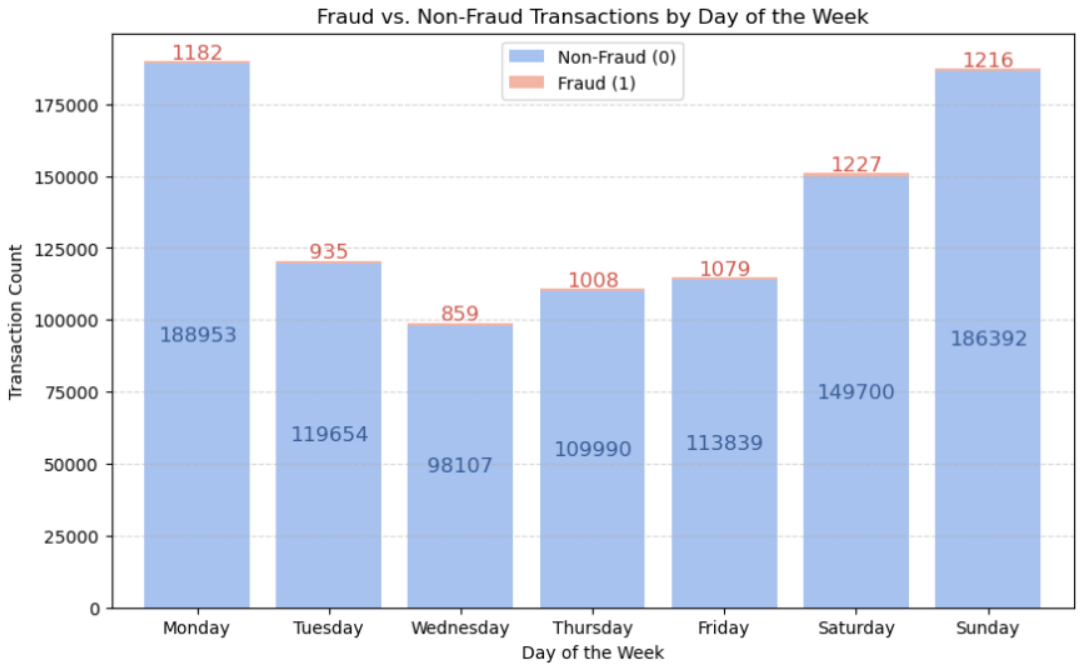


Figure 5.3.30: Fraud and Non-Fraud Transactions by Day of the Week

The chart shows the total transactions for each day, distinguishing between fraud and non-fraud transactions. **Monday** has the **highest total transactions (188,953)**, while **Wednesday** records the **lowest (98,107)**. Fraudulent transactions are present throughout the week, with **peaks on Saturday (1,227)** and **Sunday (1,216)**, suggesting that fraud activity increases over weekends.

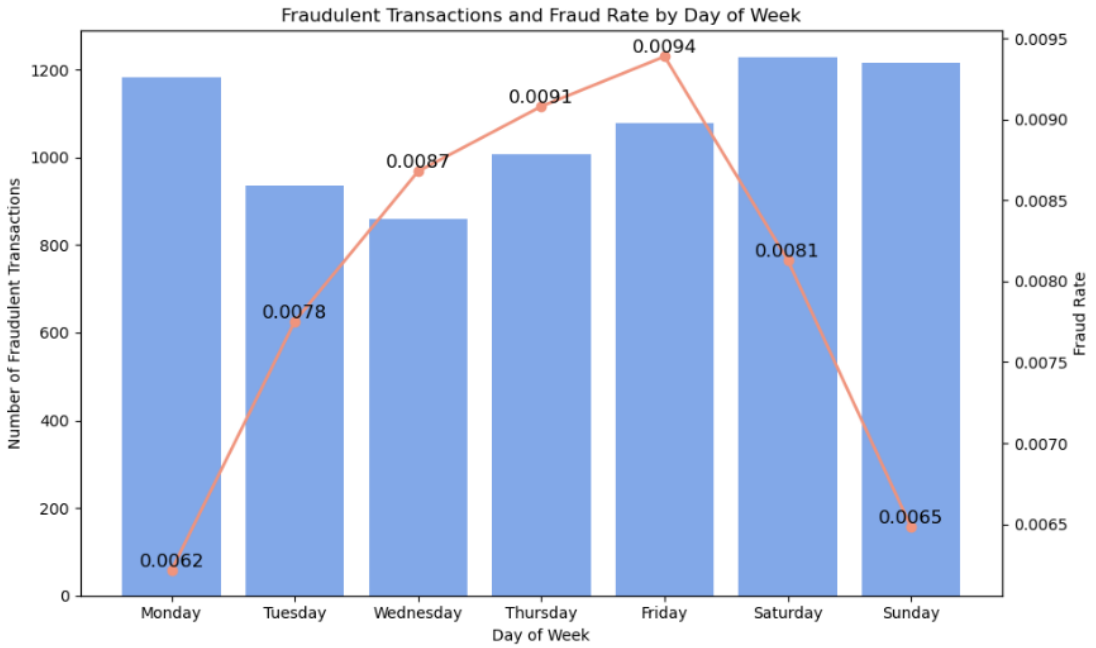


Figure 5.3.31: Fraudulent Transactions and Fraud Rate by Day of the Week

This graph further highlights fraud trends by using the fraud rate. The fraud rate increases from Monday to Friday, with a **peak on Friday** with **0.94%**, then slightly declining over the weekend. Despite a high number of fraud cases on Saturday and Sunday, the fraud rate itself is relatively lower compared to Friday.

This pattern suggests that fraudsters may take advantage of more spending on weekends, but the risk per transaction is slightly lower. The higher fraud rate on Fridays may indicate that fraudsters target end-of-week financial activities. These insights highlight the need for more fraud monitoring on Fridays and weekends to reduce risks.

10. Distance to Merchant

```python
# Convert degrees to radians
def haversine(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
    return 6371 * c  # 6371 is the radius of Earth in kilometers

# Apply Haversine formula to calculate distance
df['distance'] = df.apply(lambda row: haversine(row['lat'], row['long'], row['merch_lat'], row['merch_long']), axis=1)
```

Figure 5.3.32: Distance Calculation Using Haversine Formula

The Haversine formula is applied to calculate the **distance (in km)** between the transaction location and the merchant's location using latitude and longitude. The result is stored in the distance column, helping to identify unusual transactions that may indicate fraud.

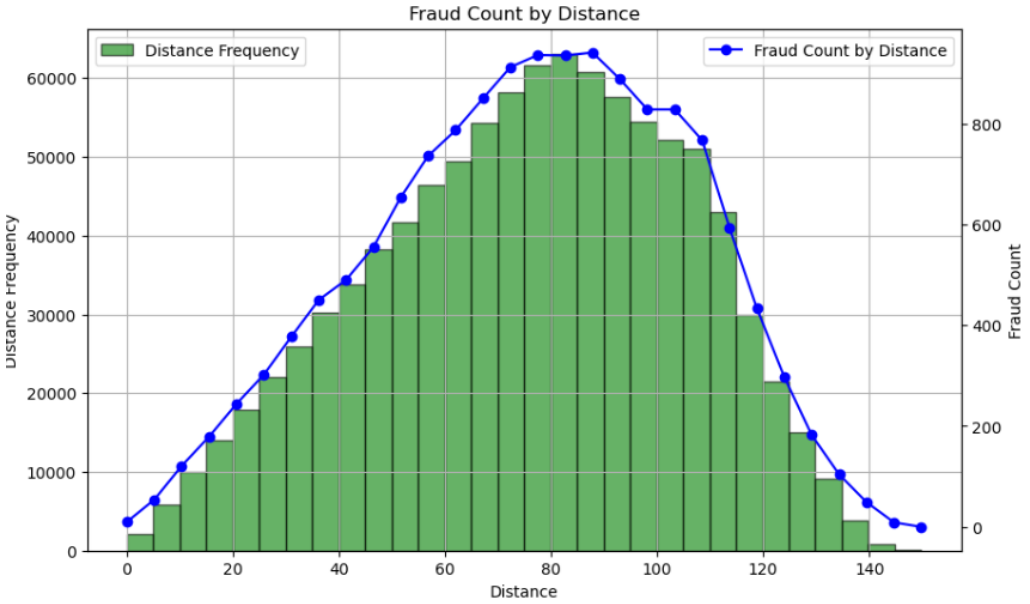


Figure 5.3.33: Transaction and Fraud Count by Distance

This analysis examines the relationship between transaction distance and fraud occurrence. The histogram (green bars) represents the frequency of transactions at different distance ranges, while the blue line shows the number of fraudulent transactions within each distance ranges.

The transaction frequency follows a normal-like distribution, with a peak at around 80 km. Fraud transactions also follow a similar trend, with the highest counts observed in middle range distances (70-90 km).

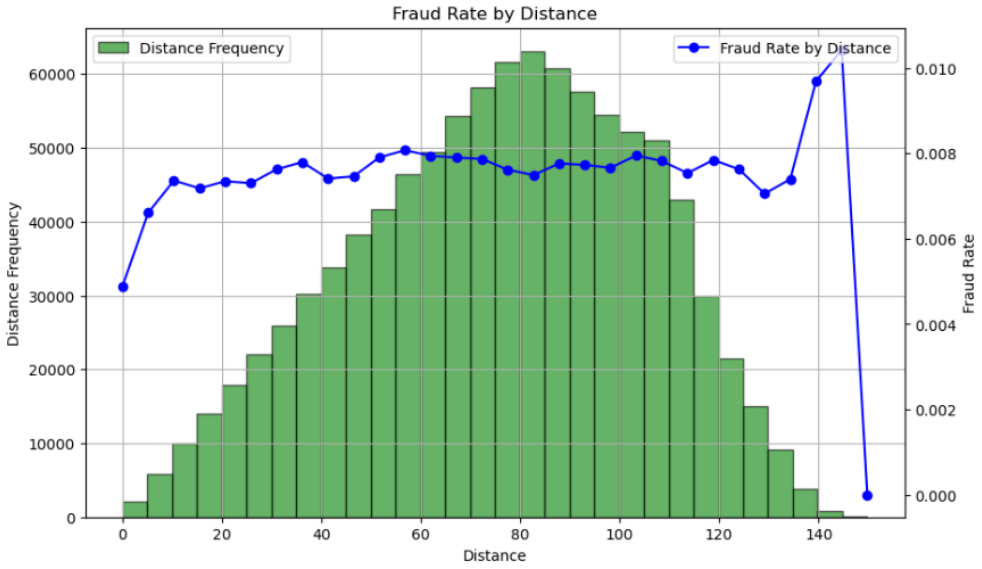


Figure 5.3.34: Transaction and Fraud Rate by Distance

This graph further showed how fraud rates vary with transaction distance. Similar to previous graph, the histogram represents the frequency of transactions at different distances, but the blue line shows the fraud rate across those distances. The fraud rates remain relatively stable across most distances but show an increase at very long distances, with a sharp peak around 140 km. This suggests that fraudsters may exploit extreme distances for fraudulent activities, possibly to bypass location-based security measures.

11. City Population

To understand how city populations are distributed, a box plot analysis is conducted using IQR method to identify the normal range and outliers.

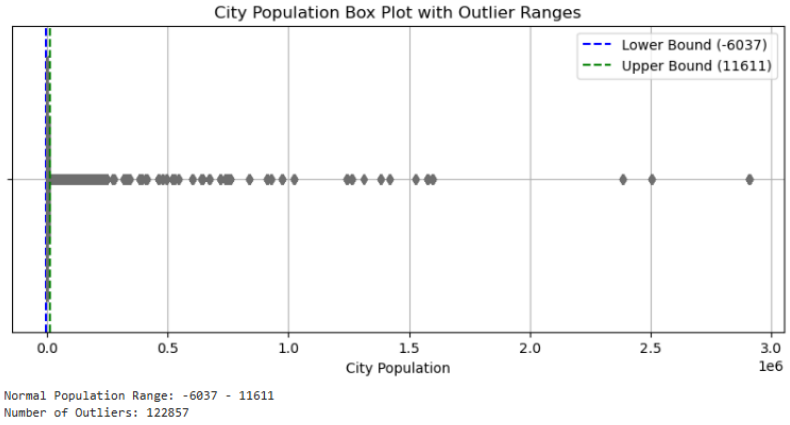


Figure 5.3.35: Boxplot of City Population with Outlier Ranges

In the previous step, outliers with non-fraudulent transactions were removed to focus on meaningful fraud trends. Most cities have populations **below 11,611**, indicating that smaller cities dominate the dataset. However, **122,857 cities** were identified as **outliers**, meaning their populations significantly exceed this threshold.

```
# Check the minimum and maximum values of 'city_pop'
print('Minimum population: ', df['city_pop'].min())
print('Maximum population: ', df['city_pop'].max())
```

```
Minimum population:  23
Maximum population:  2906700
```

```
# Define custom bins and labels
bins = [0, 5000, 10000, 50000, 3000000]  # The range of city populations
labels = ['Small Cities', 'Medium Cities', 'Large Cities', 'Very Large Cities']  # Labels for each range

# Assign categories based on the 'city_pop'
df['pop_group'] = pd.cut(df['city_pop'], bins=bins, labels=labels, right=True)
```

Figure 5.3.36: Assignment of City Population Categories

Since city populations range from 23 to over 2.9 million, they are grouped into four categories: *Small Cities*, *Medium Cities*, *Large Cities*, and *Very Large Cities*. This grouping ensures a meaningful classification, balancing the majority of cities within reasonable population sizes while accounting for larger urban centres.

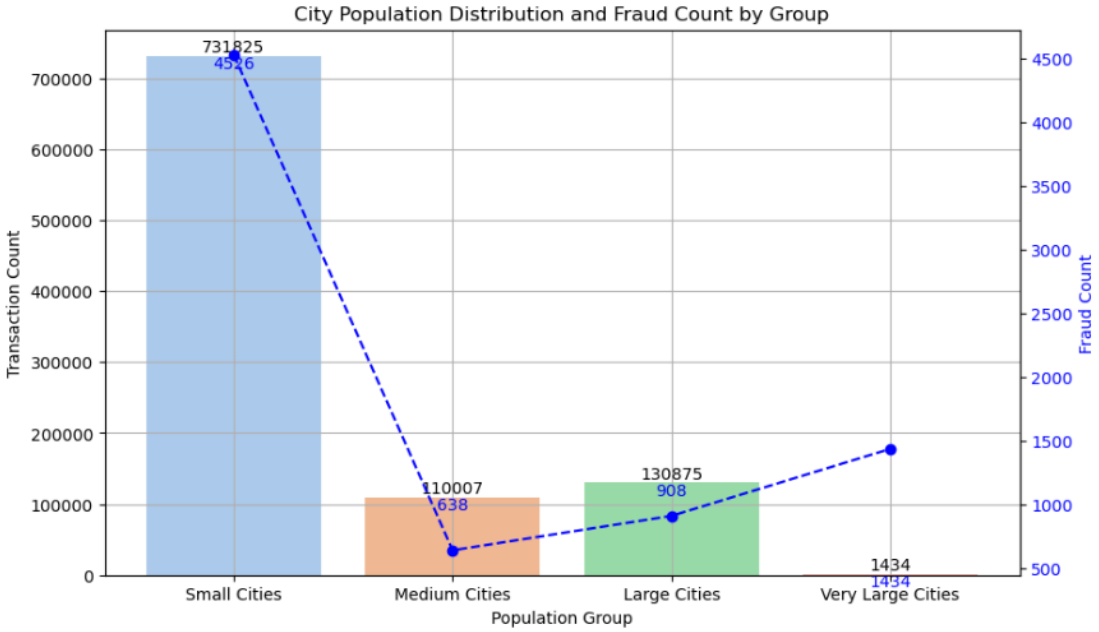


Figure 5.3.37: Transaction and Fraud Count by City Population Group

The bar chart represents the number of transactions across these population groups, while the dotted line represents the corresponding fraud count. **Small cities** have the **highest transaction**

volume, with **731,825 transactions**, followed by large, medium and very large cities. Fraud count follows a similar pattern, with small cities have the most fraud cases (4,526), while medium and large cities have significantly lower fraud counts (638 and 908 respectively). Interestingly, fraud cases rise again in very large cities even though with fewer transactions, where all 1,434 transactions are fraudulent. This observation suggests a potential sampling or reporting bias due to the prior removal of outliers.

12. Correlation Analysis

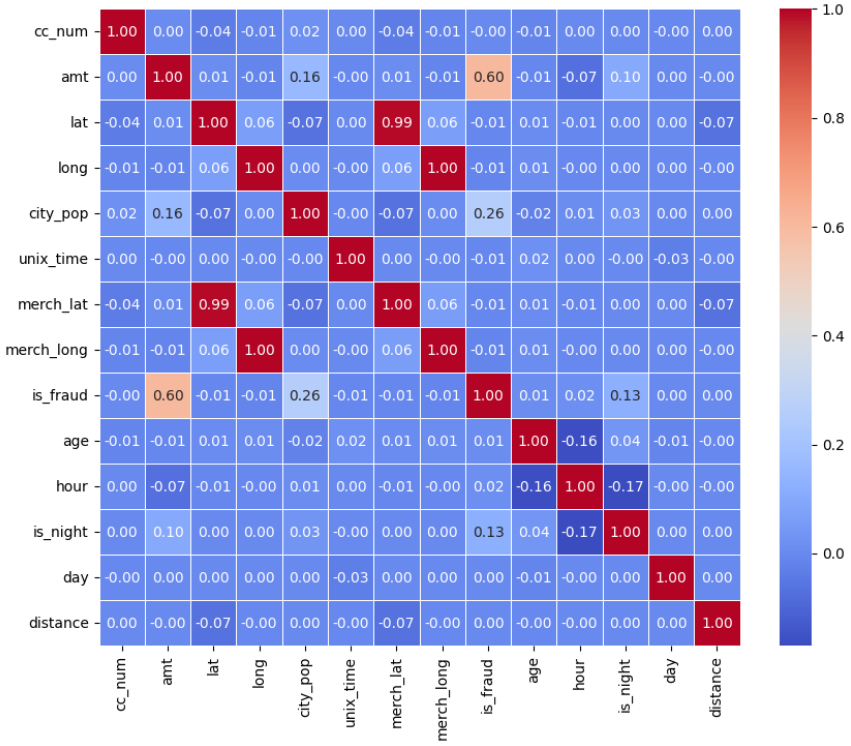


Figure 5.3.38: Heatmap of Numerical Features for Final Dataset

The heatmap visualizes the **correlation between various features** in the dataset, helping to identify potential relationships that may be useful for fraud detection. The correlation values that close to 1 or -1 indicate a strong correlation, values near 0 indicate little to no correlation.

In this case, the **highest correlation with fraud** (*is_fraud*) is observed in the transaction **amount (*amt*)**, with a correlation of **0.6**, suggesting that fraudulent transactions tend to have higher amounts. The city population (*city_pop*) also shows a moderate correlation of 0.26, indicating that fraud is slightly more frequent in larger cities. Additionally, *is_night* has a weaker but still notable correlation of 0.13, implying that fraudulent transactions are more

likely to occur at night. Other features like lat, long, unix_time, merch_lat, merch_long, hour and age have very low correlations, while day and distance showed almost no correlation with fraud.

The analysis revealed a very high correlation between ***lat*** and ***merch_lat*** (0.99) and between ***long*** and ***merch_long*** (1.00). Due to this strong multicollinearity, their individual correlations with *is_fraud* are very low, indicating potential redundancy. Additionally, the features ***trans_date_trans_time, dob*** and ***unix_time*** do not provide direct value for fraud detection, as key information such as *hour* and *age* has already been extracted. To avoid unnecessary features and improve model efficiency, these variables are **removed** from the dataset.

5.3.4 Encoding

```
# Replace 'M' with 1 and 'F' with 0
df['gender'] = df['gender'].replace({'M': 1, 'F': 0})

# Perform one-hot encoding on the 'category' column
df = pd.get_dummies(df, columns=['category'], drop_first=False)
category_columns = [col for col in df.columns if 'category_' in col]
df[category_columns] = df[category_columns].astype(int)

df = pd.get_dummies(df, columns=['age_group'], prefix='age_group')

df = pd.get_dummies(df, columns=['pop_group'], prefix='pop_group')

# Convert all boolean columns to 1 and 0 (int type)
df = df.apply(lambda x: x.astype(int) if x.dtype == 'bool' else x)

# Compute fraud rate per `cc_num` using the entire dataset
cc_fraud_rate = df.groupby('cc_num')['is_fraud'].mean()
df['cc_fraud_rate'] = df['cc_num'].map(cc_fraud_rate)
df = df.drop(columns=['cc_num'])
```

Figure 5.3.39: Binary Encoding, One-hot Encoding and Target Encoding Applied

The encoding process transforms categorical and boolean variables into a format suitable for machine learning. First, the *gender* column is converted into numerical values using **binary encoding**, where 'M' is replaced with 1 and 'F' with 0.

One-hot encoding is applied to categorical variables such as *category, age_group* and *pop_group,* creating separate binary columns for each unique category. These binary columns are then converted into integer values (0 and 1) to ensure compatibility with machine learning models.

Additionally, the **target encoding** is applied to the *cc_num* column by calculating the fraud rate for each unique credit card number based on the mean fraud occurrence. This transformation preserves useful fraud risk information while preventing potential data leakage.

5.3.5 Resampling, Data Splitting and Modelling

1. Initiating the Modelling Process

The process begins by calling the *fit_and_evaluate_model()* function. This function integrates all the necessary steps in the machine learning workflow. It performs the following tasks sequentially: Resampling the Data, Splitting the Data, Initializing and Training the Model and Evaluating the Model.

```
# Evaluate with no resampling for Random Forest
print("Evaluating with No Resampling (Random Forest):")
fit_and_evaluate_model(X, y, model_type='RF', resampling_method=None)
```

Figure 5.3.40: Function Calling for Random Forest without Resampling

```
# Function to fit and evaluate the model with different resampling methods and classifiers
def fit_and_evaluate_model(X, y, model_type='RF', resampling_method=None):
    # Apply resampling
    X_res, y_res = resample_data(X, y, method=resampling_method)

    # Split data (70:30)
    X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test_size=0.3, random_state=42, stratify=y_res)

    # Initialize the model based on the selected type
    if model_type == 'RF':
        model = RandomForestClassifier(random_state=42)
    elif model_type == 'XGBoost':
        model = XGBClassifier(random_state=42)
    elif model_type == 'AdaBoost':
        model = AdaBoostClassifier(random_state=42)

    # Train the model
    model.fit(X_train, y_train)

    # Evaluate the model
    evaluate_model(model, X_test, y_test)
```

Figure 5.3.41: Function Definition for Model Training and Evaluation

2. Resampling the Data

The first operation inside *fit_and_evaluate_model()* addresses the class imbalance issue. The *resample_data()* function is called to apply the specified resampling technique — SMOTE, random oversampling, random undersampling, or no resampling. It returns the resampled

feature set (*X_res)* and label set (*y_res)*. This step ensures that the dataset is balance before the data is split into training and testing sets.

```
# Function to resample data (SMOTE, oversampling, undersampling, or no resampling)
def resample_data(X, y, method=None):
    if method == 'SMOTE':
        smote = SMOTE(random_state=42)
        X_res, y_res = smote.fit_resample(X, y)
    elif method == 'oversample':
        ros = RandomOverSampler(random_state=42)
        X_res, y_res = ros.fit_resample(X, y)
    elif method == 'undersample':
        rus = RandomUnderSampler(random_state=42)
        X_res, y_res = rus.fit_resample(X, y)
    else:
        X_res, y_res = X, y  # No resampling
    return X_res, y_res
```

Figure 5.3.42: Function Definition for Resampling

3. Splitting the Data

The resampled data (*X_res, y_res*) is then split into training and testing datasets using a **70:30 split**. The split is performed using *train_test_split()* with the *stratify=y_res* parameter to maintain consistent class distribution across both sets. This ensures that the model is trained and tested on representative samples.

```
# Split data (70:30)
X_train, X_test, y_train, y_test = train_test_split(X_res, y_res, test_size=0.3, random_state=42, stratify=y_res)
```

Figure 5.3.43: Code for Data Splitting

4. Initializing and Training the Model

Based on the *model_type* parameter, the function initializes one of the selected classifiers: Random Forest, XGBoost, or AdaBoost. The chosen model is then trained using the training dataset (*X_train, y_train*) through the *fit()* method.

```
# Initialize the model based on the selected type
if model_type == 'RF':
    model = RandomForestClassifier(random_state=42)
elif model_type == 'XGBoost':
    model = XGBClassifier(random_state=42)
elif model_type == 'AdaBoost':
    model = AdaBoostClassifier(random_state=42)

# Train the model
model.fit(X_train, y_train)
```

Figure 5.3.44: Code for Model Training

5. Evaluating the model

Once the model is trained, the *evaluate_model()* function is called to assess its performance. This function calculates various performance metrics such as accuracy, recall, precision, F1-score, MCC, and AUC. Additionally, it outputs the confusion matrix and classification report for further insights into how the model performs on testing datasets.

```
def evaluate_model(model, X_test, y_test):
    # Make predictions on the test data
    y_test_pred = model.predict(X_test)
    y_test_prob = model.predict_proba(X_test)[:, 1]

    # Helper function to print and plot the metrics
    def print_metrics(data_type, y_true, y_pred, y_prob):
        print(f"{data_type} Data Metrics:")
        print(f"Accuracy: {accuracy_score(y_true, y_pred):.4f}")
        print(f"Recall: {recall_score(y_true, y_pred):.4f}")
        print(f"Precision: {precision_score(y_true, y_pred):.4f}")
        print(f"F1-Score: {f1_score(y_true, y_pred):.4f}")
        print(f"MCC: {matthews_corrcoef(y_true, y_pred):.4f}")
        print(f"AUC: {roc_auc_score(y_true, y_prob):.4f}")
        print("Classification Report:")
        print(classification_report(y_true, y_pred))

        # Confusion matrix
        cm = confusion_matrix(y_true, y_pred)
        plt.figure(figsize=(5, 4))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                    xticklabels=['Not Fraud', 'Fraud'],
                    yticklabels=['Not Fraud', 'Fraud'])
        plt.xlabel('Predicted Label')
        plt.ylabel('True Label')
        plt.title(f'{data_type} Confusion Matrix')
        plt.tight_layout()
        plt.show()

    # Evaluate testing data
    print_metrics("Testing", y_test, y_test_pred, y_test_prob)
```

Figure 5.3.45: Function Definition for Model Evaluation

6. Repeating for All Combinations

Steps 1 to 5 are repeated for all combinations of resampling methods (SMOTE, oversampling, under-sampling and no resampling) and machine learning models (Random Forest, XGBoost and AdaBoost). This allows for a comprehensive comparison of how different model-resampling pairs perform in detecting fraud, enabling the selection of the best-performing configuration.

5.3.6 Model Evaluation and Comparison

Pipeline 1: Target Encoding → Resampling → Data Splitting

1. Performance Metrics

	Random Forest	XGBoost	AdaBoost
No Resampling	Accuracy: 0.9988 Recall: 0.8441 Precision: 0.9969 F1-Score: 0.9142 MCC: 0.9168 AUC: 0.9830	Accuracy: 0.9987 Recall: 0.8552 Precision: 0.9688 F1-Score: 0.9085 MCC: 0.9096 AUC: 0.9978	Accuracy: 0.9986 Recall: 0.8157 Precision: 1.0000 F1-Score: 0.8985 MCC: 0.9025 AUC: 0.9951
SMOTE	Accuracy: 0.9990 Recall: 0.9992 Precision: 0.9989 F1-Score: 0.9990 MCC: 0.9981 AUC: 1.0000	Accuracy: 0.9929 Recall: 0.9919 Precision: 0.9939 F1-Score: 0.9929 MCC: 0.9858 AUC: 0.9998	Accuracy: 0.9618 Recall: 0.9418 Precision: 0.9811 F1-Score: 0.9610 MCC: 0.9244 AUC: 0.9930
Over-sampling	Accuracy: 0.9999 Recall: 1.0000 Precision: 0.9998 F1-Score: 0.9999 MCC: 0.9998 AUC: 1.0000	Accuracy: 0.9959 Recall: 0.9999 Precision: 0.9920 F1-Score: 0.9959 MCC: 0.9919 AUC: 0.9998	Accuracy: 0.9638 Recall: 0.9497 Precision: 0.9772 F1-Score: 0.9632 MCC: 0.9279 AUC: 0.9935
Under-sampling	Accuracy: 0.9691 Recall: 0.9627 Precision: 0.9753 F1-Score: 0.9689 MCC: 0.9384 AUC: 0.9955	Accuracy: 0.9758 Recall: 0.9774 Precision: 0.9743 F1-Score: 0.9758 MCC: 0.9516 AUC: 0.9973	Accuracy: 0.9585 Recall: 0.9507 Precision: 0.9657 F1-Score: 0.9582 MCC: 0.9171 AUC: 0.9941

Table 5.3.2: Evaluation Metrics of Random Forest, XGBoost and AdaBoost with Different Resampling Techniques in Fraud Detection

Without resampling, Random Forest achieves a high Precision of 0.9969, but the Recall drops to 0.8441, suggesting poor handling of the minority class. With the use of Oversampling and SMOTE, Random Forest achieves perfect and near-perfect Recall (1.000 and 0.9992) and F1-Score (0.9999 and 0.9990). This suggests Random Forest excellent in fraud detection but raise the concerns about overfitting. For Under-sampling, the performance drops significantly with the lowest accuracy of 0.9691, among all Random Forest configurations, showing its inefficiency for highly imbalanced dataset.

For XGBoost, Oversampling achieves the best performance, with an outstanding Recall of 0.9999, F1-Score of 0.9959 and MCC of 0.9919. SMOTE also performs well, with slightly lower scores than Oversampling, but it offers a higher Precision of 0.9939. Without resampling, XGBoost achieves the highest Accuracy of 0.9997, but its Recall drops significantly to 0.8552, which is similar to Random Forest. This significant drop suggests poor handling of the model with minority class. Under-sampling shows a decline across all performance metrics, further supporting the conclusion that it is not an effective resampling technique for this dataset.

AdaBoost shows weaker performance compared to other models across all resampling techniques. Its best results are achieved with the Oversampling, reaching a F1-Score of 0.9632 and an AUC of 0.9935. However, the worst performance occurs without resampling, where the Recall is only 0.8157, even though achieving a perfect Precision. This suggests that AdaBoost is too focus on minimising false alarms when no resampling technique is applied. This means that the model has strong bias toward the majority class. As a result, it tends to misclassify the minority class, leading to imbalanced predictions and poor detection of fraud cases.

In comparing the different resampling techniques, it becomes clear that no resampling leads to high Precision but poor Recall. It is not suitable for imbalanced datasets, especially in fraud detection where Recall is very important. SMOTE significantly improves Recall, F1-Score and MCC, especially for Random Forest and XGBoost, showing the better in generalization. Oversampling performs similarly or slightly better than SMOTE, achieving near-perfect scores in both Random Forest and XGBoost. Conversely, Under-sampling consistently reduces performance across all models, likely due to the loss of valuable information from the majority class.

To find the best-performing model, the combination of model and resampling technique that consistently achieving the highest metrics should be considered. Based on the analysis, Random Forest with Oversampling archives perfect or near perfect scores across all metrics, indicating high generalization. Random Forest with SMOTE follows closely behind, with slightly lower overall performance compared to that of Oversampling. XGBoost with Oversampling also performs excellently, with slightly lower Precision and MCC than Random Forest but offering more balanced performance.

In short, Random Forest and XGBoost consistently deliver the best performance. Random Forest and XGBoost have the similar performance. This is because both of them are tree-based ensemble models. This type of models inherently handles class imbalance through feature selection and strong pattern learning. However, since they share similar strengths, direct comparison may not be valuable unless tested under more challenging conditions. Resampling methods like SMOTE and Oversampling help improving model performance by addressing imbalanced dataset, generating synthetic samples or duplicating existing ones to better classify minority class instances. In contrast, Under-sampling tends to reduce effectiveness, especially for XGBoost and AdaBoost.

2. Classification Report

	Random Forest				XGBoost				AdaBoost				
No Resampling		precision	recall	f1-score	support		precision	recall	f1-score	support		precision	recall	f1-score	support
	0	1.00	1.00	1.00	289991	0	1.00	1.00	1.00	289991	0	1.00	1.00	1.00	289991
	1	1.00	0.84	0.91	2252	1	0.97	0.86	0.91	2252	1	1.00	0.82	0.90	2252
	accuracy			1.00	292243	accuracy			1.00	292243	accuracy			1.00	292243
	macro avg	1.00	0.92	0.96	292243	macro avg	0.98	0.93	0.95	292243	macro avg	1.00	0.91	0.95	292243
	weighted avg	1.00	1.00	1.00	292243	weighted avg	1.00	1.00	1.00	292243	weighted avg	1.00	1.00	1.00	292243
SMOTE		precision	recall	f1-score	support		precision	recall	f1-score	support		precision	recall	f1-score	support
	0	1.00	1.00	1.00	289991	0	0.99	0.99	0.99	289991	0	0.94	0.98	0.96	289991
	1	1.00	1.00	1.00	289990	1	0.99	0.99	0.99	289990	1	0.98	0.94	0.96	289990
	accuracy			1.00	579981	accuracy			0.99	579981	accuracy			0.96	579981
	macro avg	1.00	1.00	1.00	579981	macro avg	0.99	0.99	0.99	579981	macro avg	0.96	0.96	0.96	579981
	weighted avg	1.00	1.00	1.00	579981	weighted avg	0.99	0.99	0.99	579981	weighted avg	0.96	0.96	0.96	579981
Over-sampling		precision	recall	f1-score	support		precision	recall	f1-score	support		precision	recall	f1-score	support
	0	1.00	1.00	1.00	289991	0	1.00	0.99	1.00	289991	0	0.95	0.98	0.96	289991
	1	1.00	1.00	1.00	289990	1	0.99	1.00	1.00	289990	1	0.98	0.95	0.96	289990
	accuracy			1.00	579981	accuracy			1.00	579981	accuracy			0.96	579981
	macro avg	1.00	1.00	1.00	579981	macro avg	1.00	1.00	1.00	579981	macro avg	0.96	0.96	0.96	579981
	weighted avg	1.00	1.00	1.00	579981	weighted avg	1.00	1.00	1.00	579981	weighted avg	0.96	0.96	0.96	579981
Under-sampling		precision	recall	f1-score	support		precision	recall	f1-score	support		precision	recall	f1-score	support
	0	0.96	0.98	0.97	2252	0	0.98	0.97	0.98	2252	0	0.95	0.97	0.96	2252
	1	0.98	0.96	0.97	2252	1	0.97	0.98	0.98	2252	1	0.97	0.95	0.96	2252
	accuracy			0.97	4504	accuracy			0.98	4504	accuracy			0.96	4504
	macro avg	0.97	0.97	0.97	4504	macro avg	0.98	0.98	0.98	4504	macro avg	0.96	0.96	0.96	4504
	weighted avg	0.97	0.97	0.97	4504	weighted avg	0.98	0.98	0.98	4504	weighted avg	0.96	0.96	0.96	4504

Table 5.3.3: Classification Reports of Random Forest, XGBoost and AdaBoost with Different Resampling Techniques in Fraud Detection

A classification report is important to determine how well a model performs on different classes, providing metrics like Precision, Recall, F1-Score and Support for each class. Unlike performance metrics which summarise performance in a single value, a classification report provides a breakdown for each class, enabling for deeper insights into how the model behaves, especially in imbalanced datasets where the majority class often dominates performance. Each combination of model and resampling method was evaluated using a classification report to determine how well the models handle both majority class (class 0) and minority class (class 1).

Without resampling, all models showed a bias toward the majority class (Class 0), especially with AdaBoost achieves the lowest Recall of 0.82 for Class 1, meaning that it missed 18% of minority-class instances, despite its Precision is perfect. Although XGBoost performs slightly better than Random Forest and AdaBoost in Recall with 0.86, but it still shows limitation with minority-class identification. Without resampling leads to high Precision but poor Recall, which is a critical issue in fraud detection, where missing positive fraud cases is costly.

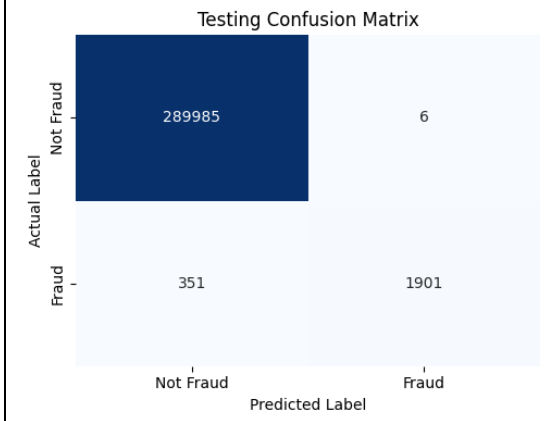
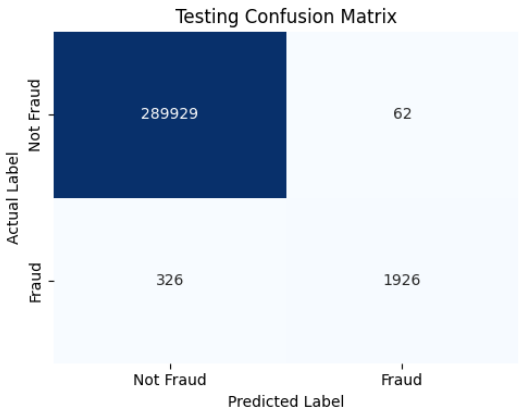
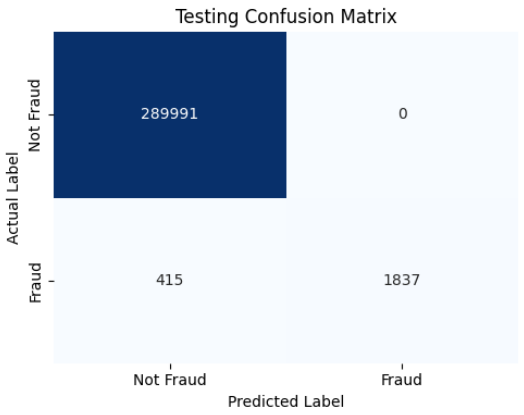
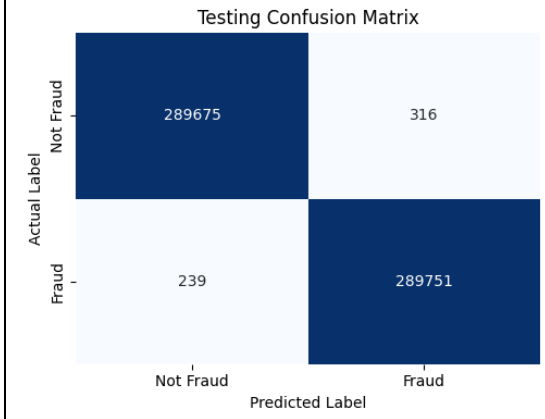
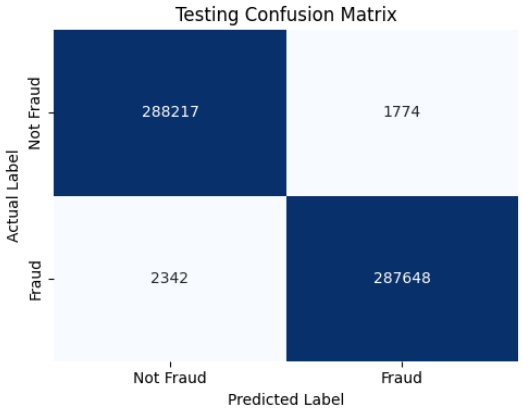
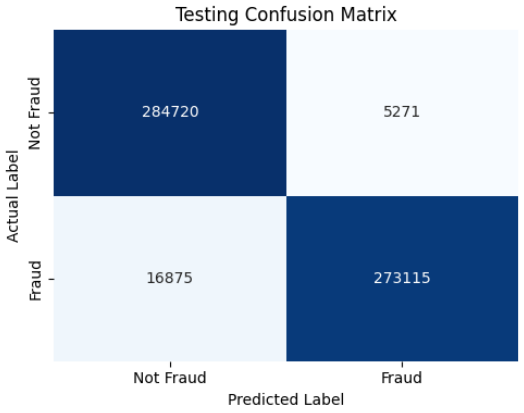
The use of SMOTE and Oversampling shows a large improvement in performance across all models. Random Forest and XGBoost achieves nearly perfect Precision, Recall and F1-Score for Class 1 under these resampling methods. AdaBoost also improved significantly, with a high F1-Score of 0.96 in Class 1, even though it still behind the other two models. However, perfect or near-perfect scores may indicate overfitting, especially for Oversampling, This is because the same minority samples are repeated too often, the model has potentially to memorise patterns in the oversampled data rather than generalising to unseen data.

With the use of Under-sampling, the performance of models drops compared to SMOTE and Oversampling because a portion of data is removed. Despite this, the results remain strong, with XGBoost achieving the highest F1-Score of 0.98 for Class 1, followed by Random Forest of 0.97 and AdaBoost of 0.96. These results show that these models can perform well even with a smaller and balanced dataset. However, Under-sampling comes with risk of discarding valuable information, which could hinder generalisation on unseen data. This trade-off becomes even more critical when considering the original class distribution, where Class 0 includes 289,991 transactions, while Class 1 consists of only 2,552, meaning that 99%

of the majority class is discarded. Given this imbalance, Under-sampling may not be the most suitable approach, especially when data retention is important.

When comparing models overall, XGBoost is proven to be the most robust and consistent across all resampling strategies, showing strong recall and F1-Scores even without using resampling technique. Random forest shows excellent performance when combining with SMOTE and Oversampling, but its Recall dropped more significantly without resampling. AdaBoost tends to underperform slightly while it is still competitive, especially with imbalanced data.

3. Confusion Matrix

	Random Forest	XGBoost	AdaBoost
No Resampling			
SMOTE			

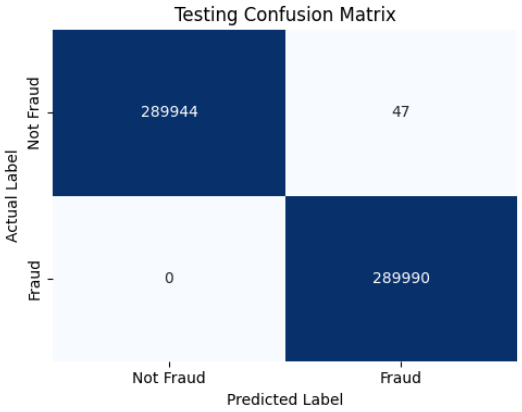
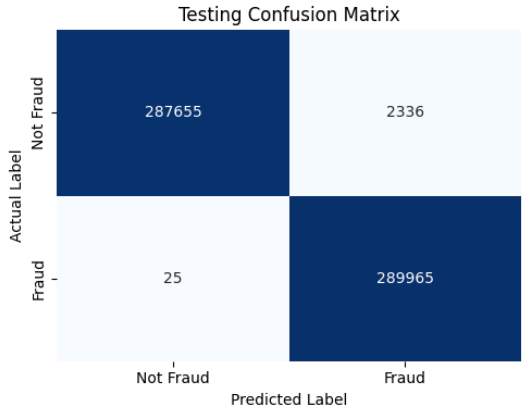
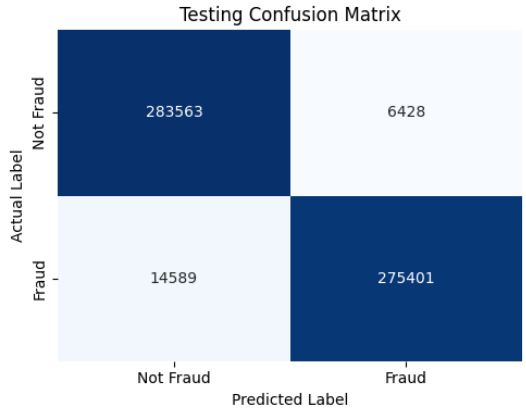
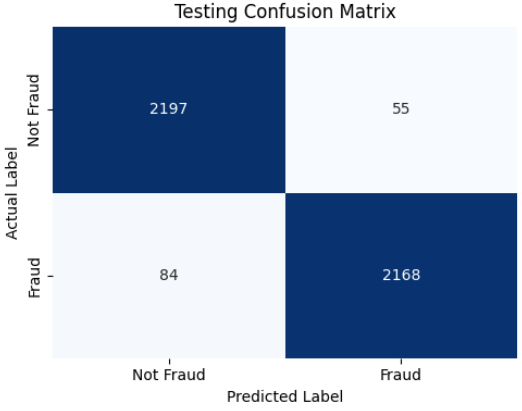
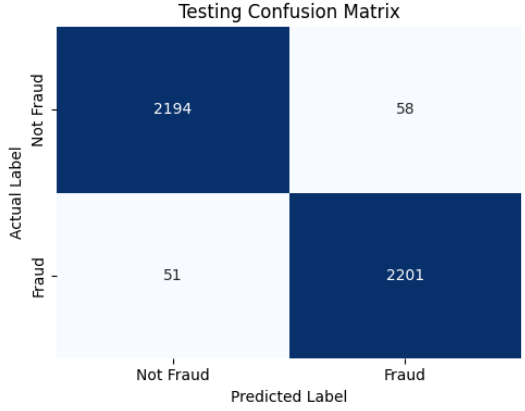
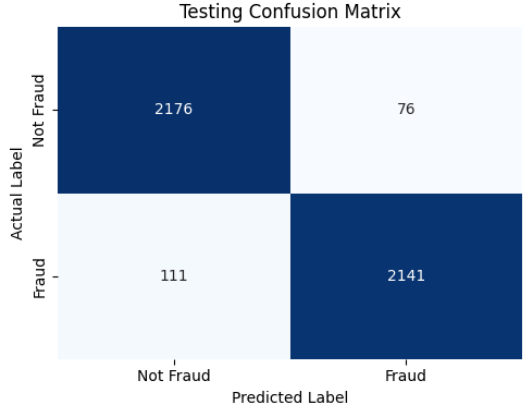
Over-sampling			
Under-sampling			

Table 5.3.4: Confusion Matrixes of Random Forest, XGBoost and AdaBoost with Different Resampling Techniques in Fraud Detection

The confusion matrix provides clear breakdown into how models classify fraud (class 1) and non-fraud (class 0) transactions by showing exact numbers of correct and incorrect predictions. This helps to identify specific errors that cannot be directly found in performance metrics or classification report.

When training and testing on the original imbalanced dataset with 289,991 non-fraud cases and 2,552 fraud cases, all three models show somewhat difficult to identify fraud cases correctly. Random Forest and XGBoost show relatively better performance, correctly identifying 1,901 and 1,926 fraudulent transactions, but still missing 351 and 326 respectively. AdaBoost performs the worst without resampling, detecting only 1,837 frauds while missing 415 frauds, which shows a bias towards the majority class. This is a normal problem in imbalanced datasets where minority class is underrepresented. Surprisingly, it produces no False Positive, achieving perfect Precision for fraud detection.

The models' ability to detect fraud is improved by using SMOTE. Random Forest with SMOTE shows a very strong performance with only 239 False Negative and 316 False Positives. This means that among all actual fraud cases (289,990), only 239 was predicted incorrectly as non-fraud, while 316 legitimate transactions are wrongly flagged as fraud. XGBoost also performs well under SMOTE with a False Negative of 2,342 and False Negative of 1,774, although higher than those of Random Forest. AdaBoost benefits the least from SMOTE and misclassifying a large number of both frauds (16,875 False Negatives) and non-frauds (5,271 False Positives), which indicate difficulty in capturing patterns in synthetically balanced data as SMOTE may introduce noise for some models.

Oversampling reach nearly perfect results for Random Forest, which detects all 289,990 fraudulent transactions with only 47 False Positives. This may indicate potential overfitting as such high scores may not generalise well to unseen data. XGBoost also performs strongly, only 25 frauds are missed, and 2,336 legitimate transactions are misclassified as fraud. AdaBoost shows an improvement in False Negative if compared to using SMOTE, with a number of 14,589 fraud missed, but the False Positive is increase to 6,428. AdaBoost still remain the weakest model among all.

Under-sampling reduces the dataset to a balanced but smaller size, retaining only 2,252 non-fraudulent out of original of 289,991. This leads to a slightly lower overall performance.

However, the results remain good, with XGBoost achieves the best balance here with only 51 False Negatives and 58 False Positives. Random Forest follows closely with 84 False Negatives and 55 False Positives, while AdaBoost misclassifies the most again with 111 frauds missed and 76 legitimate transactions flagged incorrectly. Despite loss of data and patterns from majority class, the models still generalise quite well, which shows that they can work on smaller datasets.

5.3.7 Performance Across Different Pipelines

Pipeline 2: Resampling → Data Splitting → Target Encoding

Random Forest

	No Resampling	SMOTE	Oversampling	Under-sampling
Accuracy	0.9987	0.9986	0.9999	0.9658
Recall	0.8406	0.9981	1.0000	0.9636
Precision	0.9963	0.9991	0.9998	0.9679
F1-Score	0.9118	0.9986	0.9999	0.9657
MCC	0.9146	0.9971	0.9998	0.9316
AUC	0.9816	1.0000	1.0000	0.9942

Table 5.3.5: Performance of Random Forest with Pipeline 2

XGBoost

	No Resampling	SMOTE	Oversampling	Under-sampling
Accuracy	0.9986	0.9932	0.9955	0.9720
Recall	0.8477	0.9907	0.9996	0.9738
Precision	0.9690	0.9957	0.9914	0.9704
F1-Score	0.9043	0.9932	0.9955	0.9721
MCC	0.9057	0.9865	0.9910	0.9441
AUC	0.9976	0.9998	0.9998	0.9967

Table 5.3.6: Performance of XGBoost with Pipeline 2

AdaBoost

	No Resampling	SMOTE	Oversampling	Under-sampling
Accuracy	0.9986	0.9508	0.9635	0.9594
Recall	0.8157	0.9174	0.9490	0.9480
Precision	1.0000	0.9831	0.9773	0.9700
F1-Score	0.8985	0.9491	0.9630	0.9589
MCC	0.9025	0.9036	0.9274	0.919
AUC	0.9943	0.9894	0.9935	0.992

Table 5.3.7: Performance of AdaBoost with Pipeline 2

In the previous evaluation of Pipeline 1, target encoding was applied before the data splitting, followed by resampling and model training. However, this sequence introduces a potential issue of data leakage. Target encoding replaces the original feature values (*cc_num*

in this case) with their derived values (fraud rate per cc_num). If this encoding is done before the dataset splitting into training and testing sets, information from the entire dataset, including the testing part will contributes to the encoded values. As a result, the model unintentionally accesses to target information from testing set during training. This will lead to artificially high-performance metrics and poor generalizability to unseen data. So, alternative modelling sequence was tested where target encoding is applied after data splitting. This allows for a comparison between the two approaches and helps evaluate the impact of potential data leakage on model performance.

When comparing Random Forest across both modelling pipelines, the performance is slightly dropped when No Resampling, SMOTE or Under-sampling applied, as shown *Table 5.3.5*. However, with Oversampling, there is no difference in results between both pipelines. This indicates that Oversampling is robust to changes in processing sequence, likely because it replicates existing samples without introducing synthetic patterns based on target variable. The slightly drop in other methods suggests that when target encoding is applied before data splitting, the model may unintentionally refer to the target distribution across the whole dataset, causing data leakage. This leakage improves the model performance during training, which may not generalise well to unseen data, this is why the scores are slightly lower in this pipeline.

For XGBoost, the trend is quite similar to Random Forest. The performance dropped slightly across most metrics when switching to Pipeline 2, which gain supports the presence of mild data leakage in Pipeline 1. However, SMOTE stands out with slightly improvements in Accuracy, Precision, F1-Score and MCC in Pipeline 2 as shown in *Table 5.3.6*, highlighting that XGBoost may better utilise the balanced structure introduce by SMOTE once data leakage is controlled. In contrast, no resampling and Under-sampling result in more noticeable performance drops, by which F1-Score for no resampling drops from 0.9085 to 0.9043 and for SMOTE drops from 0.9085 to 0.9043. This highlights that these two resampling methods are less effective and possibly more dependent on pipeline ordering for maintaining performance. Basically, XGBoost performs best when the data is balanced. If the data is not balanced or too much information lost, it does not perform as well, especially in a proper setup without data leakage.

AdaBoost shows a different pattern. With no resampling method applied, its performance is stable between both pipelines, only AUC drops marginally from 0.9951 to

0.9943, as shown in *Table 5.3.7*. This suggests that AdaBoost is relatively insensitive to pipeline changes when no resampling is applied. This might be due to its sequential boosting nature, which can correct individual misclassifications without relying heavily on target-encoded data.

In contrast, SMOTE causes a more noticeable drop in metrics, where Accuracy drops from 0.9618 to.0.9508, Recall drops from 0.9418 to 0.9174, F1-Score drops from 0.9610 to 0.9491 and MCC from 0.9244 to 0.9036. Interestingly, Precision and AUC increase to 0.9831 and 0.9894 respectively. This may mean that the model becomes more cautious and gives more false alarms, but it is confident and accurate when it does predict fraud, thus raising AUC and Precision. This reflects the changes in decision threshold or learning pattern due to the synthetic samples introduced by SMOTE.

In short, the order of steps in the pipeline is important. If target encoding is done before splitting the data, it can leak information from the labels into training. This makes the model seem better than it actually is. This is especially a problem when using resampling methods like SMOTE that create fake data. Pipeline 2 is more realistic because it follows how things would work in real life. Random Forest and XGBoost are slightly affected by this change, but AdaBoost is more sensitive to whether the data is balanced than to the sequence of steps, which makes it worth looking into further.

Pipeline 3: Target Encoding → Data Splitting → Resampling

Random Forest

	No Resampling	SMOTE	Oversampling	Under-sampling
Accuracy	0.9988	0.9983	0.9988	0.9786
Recall	0.8441	0.8726	0.8512	0.9645
Precision	0.9969	0.9055	0.9851	0.2601
F1-Score	0.9142	0.8887	0.9133	0.4097
MCC	0.9168	0.8880	0.9151	0.4950
AUC	0.9830	0.9896	0.9860	0.9956

Table 5.3.8: Performance of Random Forest with Pipeline 3

XGBoost

	No Resampling	SMOTE	Oversampling	Under-sampling
Accuracy	0.9987	0.9944	0.9926	0.9755
Recall	0.8552	0.9094	0.9418	0.9711
Precision	0.9688	0.586	0.5118	0.2357
F1-Score	0.9085	0.7127	0.6632	0.3794
MCC	0.9096	0.7276	0.6914	0.4722
AUC	0.9978	0.9947	0.9972	0.9969

Table 5.3.9: Performance of XGBoost with Pipeline 3

AdaBoost

	No Resampling	SMOTE	Oversampling	Under-sampling
Accuracy	0.9986	0.9783	0.9759	0.9749
Recall	0.8157	0.9294	0.9529	0.9547
Precision	1.0000	0.2532	0.2362	0.2294
F1-Score	0.8985	0.398	0.3786	0.3699
MCC	0.9025	0.479	0.4680	0.4615
AUC	0.9951	0.9904	0.9942	0.9943

Table 5.3.10: Performance of AdaBoost with Pipeline 3

In Pipeline 3, the sequence of operations starts from target encoding, followed by splitting dataset into training and testing sets and finally applying resampling techniques only to the training set. This approach is different from Pipeline 1 and Pipeline 2, where resampling is applied before data splitting. As a result, Pipeline 3 introduces a key problem, which

resampling is applied in isolation to only part of the data, leading to an incomplete correction of class imbalance and potential distribution mismatch between training and testing sets.

For Random Forest, the best performance is when no resampling applied. It achieves a high F1-Score of 0.9142, Precision of 0.9969 and MCC of 0.9168, as shown in *Table 5.3.8*. However, when resampling techniques like SMOTE and Oversampling are applied after splitting, the performance does not improve in some cases worsened. For example, SMOTE achieves a better Recall of 0.8726, but its Precision drops to 0.9055 and F1-Score drops to 0.8887. Under-sampling has significantly improved Recall to 0.9645 but caused a drastic drop in Precision to only 0.2601, which led to a sharp decrease in F1-Score to 0.4097. This indicates that the model is aggressively predicting positives, even though many of them are incorrect. This is due to the skewed training distribution created by Under-sampling a small subset of the majority class. The low Precision and F1-Score confirm that although the model can detect many fraud cases (high Recall), it also misclassified many normal transactions as fraudulent (low Precision).

XGBoost follows a similar trend. It performs well without any resampling, achieving Accuracy of 0.9987, F1-Score of 0.9085, MCC of 0.9096 and AUC of 0.9978, as shown in *Table 5.3.9*. When resampling techniques are applied after splitting, especially SMOTE and Oversampling, there is a significant decline in Precision, where it drops to 0.586 and 0.5118 respectively. The decline is even bigger with under-sampling, where precision falls to just 0.2357. These low Precision values significantly reduce the F1-Score to 0.7127, 0.6636 and 0.3794 respectively, although the Recall is high. This imbalance indicates that resampling methods fail to generalise well on the unseen test data because they only applied to the training set. This can be attributed to overfitting on the resampled training set, where synthetic minority samples or duplicated minority observations skewed the learning patterns. The model learns the resampled training data too much instead of learning real patterns, so it fails on new data.

AdaBoost shows even worse performance degradation. Without resampling applied, it achieves a modest F1-Score of 0.8985, even lower than Random Forest and XGBoost, as shown in *Table 5.3.10*. While using SMOTE, Oversampling and Under-sampling, Precision falls to just 23%-25%, while F1-Score and MCC are below 0.4 and 0.5 respectively. This suggests that AdaBoost is more sensitive to noise from synthetic or duplicated samples generated during

resampling. Under-sampling introduces another problem where the model becomes overly biased toward the minority class, sacrificing Precision and generalizability in the process.

Pipeline 3 generally not performs well because of applying resampling methods after data splitting. This sequence of operations is not recommended because the model learns from a training set whose class distribution has been artificially changed, but the testing set remains imbalanced. This creates a mismatch between what the model learns and what it sees during testing. As a result, the model performs well during training but fails to generalise on new data. It often gives a high Recall but low Precision.

In contrast, Pipeline 1 and Pipeline 2 fix this issue by resampling before data splitting. This makes sure both training and testing sets have a consistent class distribution. It helps the model to learn the patterns better and provides more reliable results. In short, Pipeline 3 with data splitting done first before resampling is not suitable for imbalanced classification tasks like fraud detection, as it creates mismatches in data distribution, increases the risk of overfitting and reduce ability of model to generalise well to unseen data.

5.3.8 Hyperparameter Tuning

Model performance is highly influenced by hyperparameters, which control complexity, regularization, and decision rules. Instead of relying on defaults, systematic tuning can improve generalization and help balance recall, precision, and stability—especially in imbalanced tasks like fraud detection.

Two widely used approaches for hyperparameter optimization are **Randomized Search** and **Grid Search**. Randomized Search samples parameter combinations at random from defined ranges. It is faster and more efficient when the parameter space is large. Grid Search, in contrast, evaluates all possible combinations within a smaller, targeted space. While slower, it ensures thorough exploration of promising values.

In this study, Randomized Search was limited to **20 iterations (n_iter=20)** with **3-fold cross-validation (cv=3)** to balance efficiency and robustness. Grid Search also used **3-fold cross-validation**, but systematically explored a smaller, more focused hyperparameter space. This ensured comparability between the two methods, while keeping computational cost manageable.

Both methods were tested under four resampling strategies: **no resampling, SMOTE, oversampling,** and **under-sampling**. This allowed comparison of whether performance gains came from algorithm tuning, data balancing, or both. The same approach was applied across **Random Forest, XGBoost,** and **AdaBoost** for consistency.

Random Forest

Randomised Search Hyperparameter Space	Grid Search Hyperparameter Space
`random_params = { 'n_estimators': [50, 100, 150, 200], 'max_depth': [None, 5, 10, 15, 20], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4], 'max_features': ['sqrt', 'log2'] }`	`grid_params = { 'n_estimators': [100, 200], 'max_depth': [None, 10, 20], 'min_samples_split': [2, 5], 'min_samples_leaf': [1, 2], 'max_features': ['sqrt', 'log2'] }`

Table 5.3.11: Random Forest Hyperparameter space settings

For **Random Forest**, the hyperparameter space was designed to balance model complexity, generalization, and computational cost, as shown in *Table 5.3.11*. The **number of estimators (n_estimators)** was set between 50 and 200 in Randomized Search to explore both smaller and larger ensembles, while Grid Search focused on 100 and 200 as practical defaults that provide stability without excessive computation. The **maximum depth (max_depth)** parameter included both unrestricted trees (None) and constrained depths (5, 10, 15, 20) in Randomized Search to test how limiting tree growth impacts overfitting, whereas Grid Search narrowed this to None, 10, and 20 for targeted optimization. The **minimum samples required to split an internal node (min_samples_split)** and **minimum samples required at a leaf (min_samples_leaf)** were varied across small values (2, 5, 10 for split; 1, 2, 4 for leaf) to regulate how finely trees partition the data, with Grid Search refining this range to 2 and 5 for splits and 1 and 2 for leaves for efficiency. Finally, the **max_features** parameter was restricted to 'sqrt' and 'log2', two common strategies in Random Forests that promote diversity among trees and help reduce correlation between them

Without hyperparameter tuning	Randomised search	Grid search
No resampling + Random Forest Accuracy: 0.9987 Recall: 0.8406 Precision: 0.9963 F1-Score: 0.9118 MCC: 0.9146 AUC: 0.9816	Best Accuracy (CV): 0.9987 Best Params: {'n_estimators': 150, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': None} Accuracy: 0.9988 Recall: 0.8401 Precision: 0.9995 F1-Score: 0.9129 MCC: 0.9158 AUC: 0.9892	Best Accuracy (CV): 0.9987 Best Params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200} Accuracy: 0.9988 Recall: 0.8424 Precision: 0.9963 F1-Score: 0.9129 MCC: 0.9155 AUC: 0.9872
SMOTE + Random Forest Accuracy: 0.9986 Recall: 0.9981 Precision: 0.9991 F1-Score: 0.9986 MCC: 0.9971 AUC: 1.0000	Best Accuracy (CV): 0.9982 Best Params: {'n_estimators': 150, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': None} Accuracy: 0.9984 Recall: 0.9977 Precision: 0.9991 F1-Score: 0.9984 MCC: 0.9968 AUC: 1.0000	Best Accuracy (CV): 0.9985 Best Params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200} Accuracy: 0.9986 Recall: 0.9981 Precision: 0.9990 F1-Score: 0.9986 MCC: 0.9971 AUC: 1.0000
Oversampling + Random Forest Accuracy: 0.9999 Recall: 1.0000 Precision: 0.9998 F1-Score: 0.9999 MCC: 0.9998 AUC: 1.0000	Best Accuracy (CV): 0.9999 Best Params: {'n_estimators': 150, 'min_samples_split': 5, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': None} Accuracy: 0.9999 Recall: 1.0000 Precision: 0.9998 F1-Score: 0.9999 MCC: 0.9998 AUC: 1.0000	Best Accuracy (CV): 0.9999 Best Params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200} Accuracy: 0.9999 Recall: 1.0000 Precision: 0.9998 F1-Score: 0.9999 MCC: 0.9998 AUC: 1.0000
Undersampling + Random Forest Accuracy: 0.9658 Recall: 0.9636 Precision: 0.9679 F1-Score: 0.9657 MCC: 0.9316 AUC: 0.9942	Best Accuracy (CV): 0.9673 Best Params: {'n_estimators': 200, 'min_samples_split': 2, 'min_samples_leaf': 2, 'max_features': 'sqrt', 'max_depth': None} Accuracy: 0.9660 Recall: 0.9618 Precision: 0.9700 F1-Score: 0.9659 MCC: 0.9321 AUC: 0.9944	Best Accuracy (CV): 0.9677 Best Params: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 5, 'n_estimators': 200} Accuracy: 0.9658 Recall: 0.9609 Precision: 0.9704 F1-Score: 0.9656 MCC: 0.9317 AUC: 0.9945

Table 5.3.12: Random Forest Hyperparameter Tuning Results

Random Forest Hyperparameter Tuning Results Interpretation

When **no resampling** was applied, the results indicated that Random Search and Grid Search produced nearly identical performance. Both configurations achieved training and testing accuracies close to 0.9987 and 0.9988 respectively, with MCC values around 0.915. This shows that the default parameters of Random Forest were already highly effective, and fine-tuning did not bring significant improvements. Grid Search slightly improved recall, while Random Search provided better precision, but the difference was negligible. Compared to the model performance before fine-tuning, both approaches showed almost no improvement, confirming that the default Random Forest parameters were already highly effective. This suggests that in the absence of resampling, Random Forest is robust enough to perform well without the need for extensive parameter optimization.

When **SMOTE** was applied, Random Forest performance remained almost the same before fine-tuning, with only very small changes across metrics since the model was already performing at a high level. Both Random Search and Grid Search produced almost identical results, with training and testing accuracies remaining above 0.998 and MCC values close to 0.997. Grid Search showed slightly higher recall, reflecting a marginally stronger ability to identify minority class cases, while Random Search provided slightly better performance in precision. The main difference in parameter settings between the two approaches was that Grid Search favoured a smaller split size, which tends to improve recall, whereas Random Search leaned toward settings that maintained stronger precision. Compared to the model before fine-tuning, the differences in performance were minimal, showing only a very slight drop or gain across metrics. This confirms that the fine-tuning process did not significantly alter the effectiveness of Random Forest under SMOTE, and that the real performance improvement came from the resampling itself rather than hyperparameter adjustments.

When **oversampling** was applied, Random Forest achieved the strongest performance, with both Random Search and Grid Search producing identical outcomes. Training and testing accuracies were nearly perfect and MCC values reached 0.9998, indicating near-perfect classification. Although the two search methods selected different parameter settings, these differences had no effect on the results, as both models converged to the same performance level. Compared to the model before fine-tuning, there was no real improvement, since Random Forest already performed at its maximum under oversampling. This shows that

oversampling was the key factor driving the near-perfect results, while fine-tuning brought no measurable gains despite the different parameter choices.

When **under-sampling** was applied, Random Forest performance was lower compared to other resampling methods and fine-tuning did not provide meaningful improvements. Training and testing accuracies were around 0.966, while MCC values were slightly above 0.93. Compared to before fine-tuning, recall dropped a bit, while precision improved slightly, showing that the model became more conservative in identifying minority class cases. Between the two search approaches, Grid Search produced slightly better precision but at the cost of lower recall, while Random Search maintained a more balanced performance with a marginally higher MCC and F1-score. Overall, fine-tuning under under-sampling did not enhance performance and in fact introduced a small trade-off between recall and precision, confirming that Random Forest remains strong in its default form and that under-sampling itself is the main factor behind the reduced accuracy.

In short, hyperparameter tuning Random Forest through Random Search and Grid Search showed only minimal differences compared to the default settings, with **no meaningful improvement** across resampling methods. **Grid Search** generally offered **slightly higher recall**, while Random Search gave marginally better precision, though recall dropped a bit. The overall impact of fine-tuning was negligible, confirming that Random Forest is **already strong with default parameters**. The choice of resampling strategy was far more important, with **oversampling giving the best results**, followed by SMOTE, under-sampling, and finally no resampling.

XGBoost

Randomised Search Hyperparameter Space	Grid Search Hyperparameter Space

`random_params = {`
`'n_estimators': [50, 100, 150, 200, 300],`
`'learning_rate': [0.01, 0.05, 0.1, 0.3],`
`'max_depth': [3, 4, 5, 6, 8, 10],`
`'subsample': [0.6, 0.8, 1.0],`
`'colsample_bytree': [0.6, 0.8, 1.0],`
`'gamma': [0, 1, 3, 5],`
`'reg_alpha': [0, 0.01, 0.1, 1],`
`'reg_lambda': [0.5, 1, 1.5, 2],`
`}`
	`grid_params = {`
	`'n_estimators': [100, 200],`
	`'learning_rate': [0.05, 0.1, 0.3],`
	`'max_depth': [4, 6, 8],`
	`'subsample': [0.8, 1.0],`
	`'colsample_bytree': [0.8, 1.0],`
	`'gamma': [0, 1, 3],`
	`'reg_alpha': [0, 0.1],`
	`'reg_lambda': [0.5, 1]`
	`}`

Table 5.3.13: XGBoost Hyperparameter space settings

For **XGBoost**, the hyperparameter search space was carefully designed to balance model complexity, regularization, and ensemble diversity, as shown in Table *5.3.13*. The **number of estimators (n_estimators)** was varied more broadly in Randomized Search (50–300) to capture both faster, lightweight models and deeper, more stable ensembles, while Grid Search narrowed this to 100 and 200 for efficiency. The **learning_rate** was tuned between 0.01 and 0.3, where smaller rates allow gradual learning with more trees, and higher rates converge faster but risk overfitting; Grid Search focused on 0.05–0.3 for more practical optimization. **max_depth** was explored between 3 and 10 in Randomized Search, enabling both shallow trees for generalization and deeper ones to capture complex fraud patterns, with Grid Search emphasizing moderate depths (4, 6, 8) for stability. To handle overfitting, **subsample** (row sampling) and **colsample_bytree** (feature sampling) were tuned between 0.6 and 1.0, enforcing randomness that increases robustness, with Grid Search restricting to 0.8 and 1.0 for more reliable evaluation. The **gamma** parameter was included to control split creation, ranging from 0 (more splits allowed) to 5 (restrictive), reducing noise-driven patterns in Randomized Search while Grid Search focused on smaller values (0, 1, 3). For regularization, **reg_alpha (L1)** and **reg_lambda (L2)** were tuned to improve generalization and reduce overfitting; Randomized Search tested wider ranges (0–1 for L1, 0.5–2 for L2), while Grid Search restricted to fewer values (0, 0.1 for L1 and 0.5, 1 for L2) for computational efficiency.

Without hyperparameter tuning	Randomised search	Grid search
No Resampling + XGBoost Accuracy: 0.9986 Recall: 0.8512 Precision: 0.9682 F1-Score: 0.9060 MCC: 0.9072 AUC: 0.9970	Best Accuracy (CV): 0.9986 Best Params: {'subsample': 1.0, 'reg_lambda': 0.5, 'reg_alpha': 0.1, 'n_estimators': 300, 'max_depth': 8, 'learning_rate': 0.05, 'gamma': 0, 'colsample_bytree': 0.8} Accuracy: 0.9986 Recall: 0.8428 Precision: 0.9753 F1-Score: 0.9042 MCC: 0.9060 AUC: 0.9978	Best Accuracy (CV): 0.9986 Best Params: {'colsample_bytree': 0.8, 'gamma': 0, 'learning_rate': 0.1, 'max_depth': 8, 'n_estimators': 100, 'reg_alpha': 0.1, 'reg_lambda': 0.5, 'subsample': 1.0} Accuracy: 0.9986 Recall: 0.8401 Precision: 0.9738 F1-Score: 0.9020 MCC: 0.9038 AUC: 0.9975
SMOTE + XGBoost Accuracy: 0.9932 Recall: 0.9907 Precision: 0.9957 F1-Score: 0.9932 MCC: 0.9865 AUC: 0.9998	Best Accuracy (CV): 0.9982 Best Params: {'subsample': 0.8, 'reg_lambda': 0.5, 'reg_alpha': 1, 'n_estimators': 150, 'max_depth': 10, 'learning_rate': 0.3, 'gamma': 0, 'colsample_bytree': 0.8} Accuracy: 0.9982 Recall: 0.9982 Precision: 0.9983 F1-Score: 0.9982 MCC: 0.9964 AUC: 1.0000	Best Accuracy (CV): 0.9981 Best Params: {'colsample_bytree': 1.0, 'gamma': 0, 'learning_rate': 0.3, 'max_depth': 8, 'n_estimators': 200, 'reg_alpha': 0.1, 'reg_lambda': 0.5, 'subsample': 0.8} Accuracy: 0.9981 Recall: 0.9980 Precision: 0.9982 F1-Score: 0.9981 MCC: 0.9962 AUC: 1.0000
Oversampling + XGBoost Accuracy: 0.9955 Recall: 0.9996 Precision: 0.9914 F1-Score: 0.9955 MCC: 0.9910 AUC: 0.9998	Best Accuracy (CV): 0.9993 Best Params: {'subsample': 0.8, 'reg_lambda': 0.5, 'reg_alpha': 1, 'n_estimators': 150, 'max_depth': 10, 'learning_rate': 0.3, 'gamma': 0, 'colsample_bytree': 0.8} Accuracy: 0.9995 Recall: 1.0000 Precision: 0.9989 F1-Score: 0.9995 MCC: 0.9989 AUC: 1.0000	Best Accuracy (CV): 0.9993 Best Params: {'colsample_bytree': 1.0, 'gamma': 0, 'learning_rate': 0.3, 'max_depth': 8, 'n_estimators': 200, 'reg_alpha': 0.1, 'reg_lambda': 0.5, 'subsample': 0.8} Accuracy: 0.9994 Recall: 1.0000 Precision: 0.9989 F1-Score: 0.9994 MCC: 0.9989 AUC: 1.0000
Undersampling + XGBoost Accuracy: 0.9720 Recall: 0.9738 Precision: 0.9704 F1-Score: 0.9721 MCC: 0.9441 AUC: 0.9967	Best Accuracy (CV): 0.9735 Best Params: {'subsample': 1.0, 'reg_lambda': 0.5, 'reg_alpha': 0.1, 'n_estimators': 300, 'max_depth': 8, 'learning_rate': 0.05, 'gamma': 0, 'colsample_bytree': 0.8} Accuracy: 0.9736 Recall: 0.9756 Precision: 0.9717 F1-Score: 0.9736 MCC: 0.9472 AUC: 0.9971	Best Accuracy (CV): 0.9730 Best Params: {'colsample_bytree': 0.8, 'gamma': 0, 'learning_rate': 0.1, 'max_depth': 8, 'n_estimators': 200, 'reg_alpha': 0, 'reg_lambda': 0.5, 'subsample': 0.8} Accuracy: 0.9725 Recall: 0.9738 Precision: 0.9712 F1-Score: 0.9725 MCC: 0.9449 AUC: 0.9971

Table 5.3.14: XGBoost Hyperparameter Tuning Results

XGBoost Hyperparameter Tuning Results Interpretation

When **no resampling** was applied, both Random Search and Grid Search produced very similar outcomes. Training and testing accuracies remained around 0.9986, which was essentially unchanged from before fine-tuning. Random Search provided overall better performance, delivering slightly higher recall, precision, F1, and MCC. However, compared to the untuned baseline, recall, F1, and MCC dropped slightly, while precision and AUC increased. This indicates that fine-tuning brought no real benefit and, in some metrics, even slightly reduced performance.

When **SMOTE** was applied, performance improved compared to the baseline, with training and testing accuracies above 0.998 and MCC values above 0.996. Both Random Search and Grid Search produced near-identical results, with Random Search maintaining a small edge across all metrics. The gain compared to before fine-tuning was very minor (less than 0.01), but after fine-tuning, the model achieved a perfect AUC of 1.0. This shows that the real performance gain came from resampling rather than parameter optimization.

With **oversampling**, XGBoost achieved its strongest performance. Training and testing accuracies reached nearly 0.9995, and MCC values were close to 0.999. Both Random Search and Grid Search converged to almost same outcomes, with Random Search again showing slightly better balance across accuracy and F1-score. Recall and AUC both reached 1.000, reflecting near-perfect classification. The effect of fine-tuning was negligible, as oversampling alone allowed the model to reach its optimal level.

When **under-sampling** was applied, performance declined compared to SMOTE and oversampling but results still remained solid. Training and testing accuracies dropped to around 0.973, while MCC values were about 0.945. Both Random Search and Grid Search gave comparable outcomes, but Random Search consistently provided a slight advantage across all metrics. Compared to before fine-tuning, improvements were minimal, suggesting that the lower results were caused by information loss from undersampling rather than hyperparameter tuning.

In summary, fine-tuning XGBoost through Random Search and Grid Search did **not provide substantial improvements** compared to the default settings. **Random Search consistently performed better** across all resampling methods, although in the case of no resampling, the

untuned baseline outperformed both Random Search and Grid Search. Performance mainly depended on the resampling strategy, with **oversampling giving the best results**, followed by SMOTE, under-sampling, and no resampling, which was the same order as Random Forest.

AdaBoost

Randomised Search Hyperparameter Space	Grid Search Hyperparameter Space

`random_params_2 = {`
`'n_estimators': [50, 100, 150, 200, 300],`
`'learning_rate': [0.01, 0.05, 0.1, 0.5, 1.0],`
`'estimator__max_depth': [1, 2, 3],`
`'estimator__min_samples_split': [2, 5, 10],`
`'estimator__min_samples_leaf': [1, 2, 4],`
`}`
	`grid_params = {`
	`'n_estimators': [100, 200, 300],`
	`'learning_rate': [0.1, 0.5, 1.0],`
	`'estimator__max_depth': [2, 3],`
	`'estimator__min_samples_split': [2, 5],`
	`'estimator__min_samples_leaf': [2, 4],`
	`}`

Table 15.3.15: AdaBoost Hyperparameter space settings

For AdaBoost, *Table 15.3.15* showed the hyperparameter space settings. The **number of estimators (n_estimators)** was varied widely in Randomized Search (50–300) to capture models ranging from lightweight ensembles to deeper boosting chains, while Grid Search narrowed this to 100, 200, and 300 for more focused evaluation. The **learning_rate** was tuned between 0.01 and 1.0 in Randomized Search, where smaller values ensure gradual updates for stability and larger values accelerate convergence but may risk overfitting; Grid Search emphasized practical ranges of 0.1, 0.5, and 1.0. Since AdaBoost typically uses shallow trees as weak learners, the **base estimator's max_depth** was limited to small values (1–3) to maintain weak but diverse learners, with Grid Search focusing on slightly deeper splits (2 and 3) for stronger base classifiers. To further refine the tree structure, **min_samples_split** (2, 5, 10 in Randomized Search) and **min_samples_leaf** (1, 2, 4 in Randomized Search) were included to regulate overfitting, ensuring nodes split only when sufficient data supports the division. Grid Search refined these to fewer combinations (min_samples_split: 2, 5; min_samples_leaf: 2, 4) for efficiency and interpretability.

Without hyperparameter tuning	Randomised search	Grid search
No Resampling + Adaboost Accuracy: 0.9986 Recall: 0.8157 Precision: 1.0000 F1-Score: 0.8985 MCC: 0.9025 AUC: 0.9943	Best Accuracy (CV): 0.9987 Best Params: {'n_estimators': 200, 'learning_rate': 1.0, 'estimator__min_samples_split': 10, 'estimator__min_samples_leaf': 4, 'estimator__max_depth': 3} Accuracy: 0.9987 Recall: 0.8424 Precision: 0.9860 F1-Score: 0.9085 MCC: 0.9107 AUC: 0.9971	Best Accuracy (CV): 0.9987 Best Params: {'estimator__max_depth': 3, 'estimator__min_samples_leaf': 2, 'estimator__min_samples_split': 2, 'learning_rate': 1.0, 'n_estimators': 100} Accuracy: 0.9987 Recall: 0.8344 Precision: 0.9921 F1-Score: 0.9064 MCC: 0.9092 AUC: 0.9968
SMOTE + Adaboost Accuracy: 0.9508 Recall: 0.9174 Precision: 0.9831 F1-Score: 0.9491 MCC: 0.9036 AUC: 0.9894	Best Accuracy (CV): 0.9820 Best Params: {'n_estimators': 200, 'learning_rate': 1.0, 'estimator__min_samples_split': 10, 'estimator__min_samples_leaf': 4, 'estimator__max_depth': 3} Accuracy: 0.9810 Recall: 0.9731 Precision: 0.9888 F1-Score: 0.9809 MCC: 0.9622 AUC: 0.9980	Best Accuracy (CV): 0.9833 Best Params: {'estimator__max_depth': 3, 'estimator__min_samples_leaf': 4, 'estimator__min_samples_split': 2, 'learning_rate': 1.0, 'n_estimators': 300} Accuracy: 0.9818 Recall: 0.9743 Precision: 0.9891 F1-Score: 0.9816 MCC: 0.9636 AUC: 0.9982
Oversampling + Adaboost Accuracy: 0.9635 Recall: 0.9490 Precision: 0.9773 F1-Score: 0.9630 MCC: 0.9274 AUC: 0.9935	Best Accuracy (CV): 0.9778 Best Params: {'n_estimators': 200, 'learning_rate': 1.0, 'estimator__min_samples_split': 10, 'estimator__min_samples_leaf': 4, 'estimator__max_depth': 3} Accuracy: 0.9774 Recall: 0.9750 Precision: 0.9797 F1-Score: 0.9773 MCC: 0.9548 AUC: 0.9983	Best Accuracy (CV): 0.9791 Best Params: {'estimator__max_depth': 3, 'estimator__min_samples_leaf': 2, 'estimator__min_samples_split': 2, 'learning_rate': 1.0, 'n_estimators': 300} Accuracy: 0.9790 Recall: 0.9760 Precision: 0.9818 F1-Score: 0.9789 MCC: 0.9579 AUC: 0.9987
Undersampling + Adaboost Accuracy: 0.9594 Recall: 0.9480 Precision: 0.9700 F1-Score: 0.9589 MCC: 0.9190 AUC: 0.9920	Best Accuracy (CV): 0.9706 Best Params: {'n_estimators': 300, 'learning_rate': 0.5, 'estimator__min_samples_split': 5, 'estimator__min_samples_leaf': 4, 'estimator__max_depth': 3} Accuracy: 0.9680 Recall: 0.9689 Precision: 0.9672 F1-Score: 0.9681 MCC: 0.9361 AUC: 0.9962	Best Accuracy (CV): 0.9714 Best Params: {'estimator__max_depth': 3, 'estimator__min_samples_leaf': 2, 'estimator__min_samples_split': 5, 'learning_rate': 0.5, 'n_estimators': 300} Accuracy: 0.9700 Recall: 0.9711 Precision: 0.9690 F1-Score: 0.9701 MCC: 0.9401 AUC: 0.9960

Table 5.3.16: AdaBoost Hyperparameter Tuning Results

AdaBoost Hyperparameter Tuning Results Interpretation

When **no resampling** was applied, AdaBoost performed well under both Random Search and Grid Search, with training and testing accuracies of 0.998. Random Search achieved a slightly higher MCC (0.9107) compared to Grid Search (0.9092), indicating a better balance across F1 and MCC. Grid Search, however, achieved marginally higher precision. Compared to before fine-tuning, both methods showed small gains: recall increased from 0.8157 to 0.8424 (Random Search) and 0.8344 (Grid Search), while F1 rose to just above 0.90. Precision dropped slightly from near 1.0 to the 0.98–0.99 range, but overall the model remained strong, suggesting only minor benefits from fine-tuning.

When **SMOTE** was applied, AdaBoost achieved its greatest improvement. Training and testing accuracies rose to 0.982–0.983, and MCC reached 0.963 under Grid Search, the highest among all resampling methods. Grid Search performed slightly better than Random Search across all metrics. The improvement compared to before fine-tuning was also the most significant under SMOTE, confirming it as the most effective resampling strategy for AdaBoost.

With **oversampling**, AdaBoost also improved compared to no resampling. Training and testing accuracies reached around 0.978–0.979. Grid Search again performed slightly better than Random Search across all metrics, while Random Search maintained a reasonable balance across metrics. However, the improvements compared to the untuned baseline were more modest than those achieved with SMOTE.

When **under-sampling** was applied, AdaBoost performance dropped compared to SMOTE and oversampling, though it still improved compared to the baseline. Training and testing accuracies were around 0.968–0.970, while MCC values ranged from 0.936 (Random Search) to 0.940 (Grid Search). Both search methods performed well, but Grid Search provided better MCC and recall, making it slightly superior in this setting. However, the lower results compared to other resampling strategies were largely due to information loss from reducing the dataset size.

In summary, fine-tuning AdaBoost **led to overall improvements** compared to the baseline, though precision dropped slightly under no resampling and under-sampling. Random Search performed better without resampling, but **Grid Search outperformed Random Search** in **SMOTE, oversampling, and under-sampling** by delivering higher recall, MCC, and F1-

scores. The best results were achieved with SMOTE, which gave the highest F1 and MCC values, making it the best-performing resampling strategy for AdaBoost. The ranking of effectiveness was: **SMOTE > Oversampling > Under-sampling > No resampling**. Compared to Random Forest and XGBoost, **AdaBoost showed clearer benefits from fine-tuning** under SMOTE, whereas Random Forest and XGBoost were less sensitive to tuning and relied more heavily on resampling strategies to reach their peak performance.

5.3.9 Final Model Choice

The final choice for deployment was **Random Forest combined with oversampling using default parameters**. Random Forest consistently delivered strong performance across all settings, with its default configuration already achieving near-optimal results. Fine-tuning through Random Search or Grid Search brought almost no measurable improvement, confirming that Random Forest is naturally robust and well-suited to the dataset without the need for extensive parameter optimization.

Among the resampling strategies, **oversampling** provided the best overall outcomes. It produced nearly perfect accuracies and F1-score, highlighting its ability to balance the dataset effectively and improve the detection of minority class cases. This result demonstrated that the real performance gains came from the resampling strategy rather than hyperparameter adjustments, with oversampling standing out as the most effective method.

In addition to superior results, this configuration offers simplicity and stability for deployment. Using the default Random Forest parameters reduces computational cost, avoids overfitting risks from over-tuning, and ensures reproducibility. At the same time, oversampling retains all available data while addressing class imbalance, making it more reliable than under-sampling. Random Forest's robustness across scenarios, combined with its ease of integration into platforms like Power BI through joblib export, makes it an efficient and practical solution for real-world deployment.

5.3.10 Synthetic Data Generation

Version	Key Adjustments	Training Setup	Accuracy	Recall	Precision	F1	MCC	AUC
1	- Dropped unique identifiers & **high-cardinality** columns - Removed **cc_num** then restored (70/30 split) - **Normalized** numeric cols - Random restoration of city and datetime	CTGAN, 50 epochs, batch=100	0.8987	0.1114	0.3506	0.1690	0.1550	0.6566
2	- **Fraud oversampled ×3** in focus categories - Reduced sample size (200k→100k) - Same CTGAN config	CTGAN, 50 epochs, batch=100	0.9119	0.7345	0.6395	0.6837	0.6348	0.8736
3	- Extracted **time features** (day, hour, night) - Stratified sampling for training data - Datetime rebuilt from hour - Increase epochs (longer training)	CTGAN, 300 epochs, batch=100, pac=10	0.9295	0.7809	0.5529	0.6474	0.6207	0.9062
4	- Added **distance** feature (instead of raw lat/long) - Derived **age & age_group** (instead of raw dob) - Reduced Adult frauds 50%	CTGAN, 300 epochs, batch=100, pac=10	**0.9433**	0.9149	0.5800	0.7100	0.7020	0.9636
5	- **Fraud ratio target = 15**% (instead of ×3) - Metadata defined column types - Reduce epochs (shorter training), remove pac	CTGAN, 200 epochs, batch=100	0.8810	0.8193	0.6722	0.7385	0.6679	0.8990
6	- Model switch: **CTGAN → TVAE**	TVAE, 200 epochs, batch=100	0.9307	0.9451	0.6426	0.7650	0.7447	0.9683
7	- Increase sample size (100k→500k) - Epochs = 100	TVAE, 100 epochs, batch=100	0.9400	**0.9646**	**0.7133**	**0.8202**	**0.7980**	**0.9808**

Table 5.3.17: Random Forest Evaluation Results on Different Synthetic Dataset Version

Synthetic Data 1

The first synthetic dataset was created as a **baseline experiment** to evaluate whether a straightforward CTGAN-based approach could generate structurally valid synthetic data suitable for fraud detection. The dataset was prepared by **first removing several high-cardinality and unique identifier columns**, such as trans_num, merchant, job, city, lat, long, and trans_date_trans_time, as shown in *Figure 5.3.46*. These features were excluded because they contained either too many unique values or strong identifiers as shown in *Figure 5.3.47*, which would cause the CTGAN to memorize them rather than learn meaningful fraud patterns.

```
trans_date_trans_time: 522111 unique values
merchant: 693 unique values
category: 14 unique values
gender: 2 unique values
city: 861 unique values
job: 486 unique values
dob: 922 unique values
trans_num: 974141 unique values
```

Figure 5.3.46: High cardinality columns

```
Dropped 'trans_num' from training data
Dropped columns: ['merchant', 'job', 'city', 'lat', 'long']
Dropped 'trans_date_trans_time' from training data
Dropped 'cc_num' for CTGAN training (will re-add later)
Final categorical columns for CTGAN: ['category', 'gender', 'dob', 'is_fraud']
Training data shape: (974141, 9)
```

Figure 5.3.47: Dropping High-Cardinality Columns

Special attention was given to the credit card number (cc_num). Unlike transaction IDs, which are completely unique, the same cc_num can be linked to many transactions. This makes it a **quasi-identifier**: **high cardinality but still useful for describing customer behavior**. However, training GANs directly on raw 16-digit card numbers is not meaningful, since the digits themselves do not encode useful fraud signals. Therefore, **cc_num was dropped** during training to avoid noise, as shown in *Figure 5.3.47*. After generation, it was restored with a hybrid approach: **70%** of the synthetic dataset used **existing values sampled from the original dataset** to preserve repeat usage patterns, while the remaining **30% were filled with randomly generated** Visa-like 16-digit numbers to introduce diversity, as shown in *Figure 5.3.48*. This approach retained some behavioural realism while preventing the CTGAN from overfitting on raw card numbers.

```python
# 2. cc_num: 70% existing, 30% new
unique_ccs = df_ref['cc_num'].dropna().unique()
num_existing = int(n_rows * 0.7)
num_new = n_rows - num_existing

# Sample existing
existing_sample = np.random.choice(unique_ccs, size=num_existing, replace=True)

# Generate 16-digit new values sequentially after max
start_new = int(unique_ccs.max()) + 1
new_sample = np.arange(start_new, start_new + num_new, dtype=np.int64)

# Combine and shuffle
cc_nums = np.concatenate([existing_sample, new_sample])

# Safety check
if len(cc_nums) != n_rows:
    shortfall = n_rows - len(cc_nums)
    print(f"⚠️ Fixing shortfall of {shortfall} cc_num values")
    extra = np.random.choice(unique_ccs, size=shortfall, replace=True)
    cc_nums = np.concatenate([cc_nums, extra])

np.random.shuffle(cc_nums)
synthetic_final['cc_num'] = cc_nums
```

Figure 5.3.48: Hybrid Generation of Synthetic Credit Card Numbers

To stabilize training, all numerical columns were **normalized using a Min Max Scaler** so that values fell within the range of 0 to 1, as shown in *Figure 5.3.49*. This ensured that large-value features, such as transaction amounts or population counts, did not dominate smaller-value features.

	category	amt	gender	city_pop	dob	unix_time	merch_lat	merch_long	is_fraud
0	grocery_net	0.085579	M	-0.000027	20/2/2000	0.201639	0.459222	0.851071	0
1	personal_care	0.027663	F	0.000673	17/12/1989	0.696373	0.488982	0.745235	0
2	travel	0.007365	M	0.000011	7/5/2003	0.703763	0.223322	0.860756	0
3	food_dining	0.035856	M	-0.000702	8/10/1992	0.007253	0.524684	0.896162	0
4	misc_net	0.100022	F	0.003382	13/10/1952	0.599713	0.506355	0.718086	0

Figure 5.3.49: Normalized Data using Min-Max Scaling

The CTGAN model was then **trained on a 200,000-row sample** of the dataset, with training conducted for **50 epochs** using a **batch size of 100** and default hyperparameter settings, as shown in *Figure 5.3.50*. This relatively short training run was deliberately chosen as a **baseline configuration**, serving as a reference point for comparison against later experiments that used longer training durations, advanced sampling strategies, or additional model adjustments. By starting with a moderate sample size and limited epochs, the goal was to establish a clear performance benchmark before progressively scaling complexity in subsequent datasets.

```python
# Step 3: Train CTGAN
# -----------------
df_sampled = df_cleaned.sample(200000, random_state=42)
print("Training sample shape:", df_sampled.shape)

ctgan = CTGAN(epochs=50, batch_size=100, verbose=True)
ctgan.fit(df_sampled, discrete_columns=categorical_cols)
```

Figure 5.3.50: CTGAN Training Configuration

After generation, post-processing was applied to **restore the synthetic dataset to match the original schema,** as shown in *Figure 5.3.51*. Numeric columns were **inverse transformed to their original scales**: transaction amounts (amt) were rounded and forced positive, city populations (city_pop) clipped to ⩾1, and unix_time rounded back to integers.

```
Preview after restoration:
      amt  city_pop   unix_time  merch_lat  merch_long
0  118.67        56  1334740283  41.506878  -81.784206
1   39.04      1980  1357716150  42.937873  -92.325440
2   11.13        55  1358059312  30.163575  -80.819573
3   50.30      2016  1325712860  44.654639  -77.293125
4  138.53      9853  1353227154  43.773268  -95.029547
```

Figure 5.3.51: Numeric Data after Restoration

Figure 5.3.52 showed implementation of post-processing for synthetic dataset restoration. **Identifiers were reconstructed**. A unique trans_num was assigned (e.g., T0000001), and credit card numbers (cc_num) were created using a 70/30 mix of existing and new accounts, balancing realism with diversity. **City, latitude, longitude, and population were resampled** together as blocks to maintain consistency, while transaction timestamps were randomly sampled from the original dataset to ensure temporal coverage.

```
# 1. trans_num unique ID
synthetic_final['trans_num'] = [
    f"T{str(i+1).zfill(7)}" for i in range(n_rows)
]

# 2. cc_num: 70% existing, 30% new
unique_ccs = df_ref['cc_num'].dropna().unique()
num_existing = int(n_rows * 0.7)
num_new = n_rows - num_existing

# Sample existing
existing_sample = np.random.choice(unique_ccs, size=num_existing, replace=True)

# Generate 16-digit new values sequentially after max
start_new = int(unique_ccs.max()) + 1
new_sample = np.arange(start_new, start_new + num_new, dtype=np.int64)

# Combine and shuffle
cc_nums = np.concatenate([existing_sample, new_sample])

# Safety check
if len(cc_nums) != n_rows:
    shortfall = n_rows - len(cc_nums)
    print(f"⚠️ Fixing shortfall of {shortfall} cc_num values")
    extra = np.random.choice(unique_ccs, size=shortfall, replace=True)
    cc_nums = np.concatenate([cc_nums, extra])

np.random.shuffle(cc_nums)
synthetic_final['cc_num'] = cc_nums
```

```
# 3. Restore city, lat, long, city_pop together
city_block = df_ref[['city', 'lat', 'long', 'city_pop']].sample(
    n=n_rows, replace=True, random_state=42
).reset_index(drop=True)

synthetic_final[['city', 'lat', 'long', 'city_pop']] = city_block

# Ensure city_pop positive integer >= 1
synthetic_final['city_pop'] = synthetic_final['city_pop'].apply(lambda x: max(int(round(abs(x))), 1))

# 4. Restore trans_date_trans_time
synthetic_final['trans_date_trans_time'] = np.random.choice(
    df_ref['trans_date_trans_time'], size=n_rows, replace=True
)

# 5. Format amt to 2 decimal places and ensure positive
if 'amt' in synthetic_final.columns:
    synthetic_final['amt'] = synthetic_final['amt'].apply(lambda x: round(abs(float(x)), 2))
```

Figure 5.3.52: Implementation of Post-Processing for Synthetic Dataset 1 Restoration

```
✅ Final synthetic dataset shape: (200000, 15)
       category     amt gender  city_pop         dob   unix_time  merch_lat  \
0    grocery_net  118.67      M     18182   20/2/2000  1334740283  41.506878
1  personal_care   39.04      F       836  17/12/1989  1357716150  42.937873
2         travel   11.13      M       743    7/5/2003  1358059312  30.163575
3   food_dining   50.30      M        91    8/10/1992  1325712860  44.654639
4       misc_net  138.53      F       645  13/10/1952  1353227154  43.773268

   merch_long  is_fraud trans_num        cc_num        city      lat  \
0  -81.784206         0  T0000001  3.587961e+15    Belgrade  45.7801
1  -92.325440         0  T0000002  2.131860e+14      Thomas  39.1505
2  -80.819573         0  T0000003  3.744980e+14      Orient  41.1437
3  -77.293125         0  T0000004  4.992346e+18  Eagarville  39.1118
4  -95.029547         0  T0000005  3.506041e+15   Montandon  40.9661

       long trans_date_trans_time
0 -111.1439       6/12/2019 19:31
1  -79.5030       27/5/2019 18:57
2  -72.2879      15/10/2019 16:17
3  -89.7855      16/12/2019 23:05
4  -76.8575        26/5/2019 9:51
```

Figure 5.3.53: Synthetic Dataset 1 after Post-Processing

While this restored structure and realism, it introduced a weakness: random resampling of location and time broke natural fraud patterns. In real data, fraud often happens more in certain cities or late-night hours, but random resampling spread these patterns out. The dataset still looked correct, but it no longer reflected fraud behaviour as clearly.

The evaluation metrics confirmed these limitations. The synthetic dataset achieved an **accuracy of 0.8987**, but **recall dropped sharply to 0.1114**, meaning the model failed to detect the majority of fraud cases. **Precision** was moderate at **0.3506**, and both the **F1 score and MCC remained very low at 0.1690** and **0.1550** respectively. Similarly, the **AUC** value of **0.6566** indicated poor separation between fraud and non-fraud classes.

In short, Synthetic Data 1 successfully established a structural baseline for synthetic data generation but **failed to capture meaningful fraud patterns**. The absence of **class imbalance** handling and the **reliance on random resampling** of critical features such as time and location resulted in very poor fraud detection performance. These findings highlighted the need for more targeted adjustments in later datasets, including fraud oversampling, feature engineering (extract distance, time, age), and refined training strategies to improve the model's ability to replicate real fraud patterns.

Synthetic Data 2

Synthetic Data 2 was developed as the first major improvement over the baseline. The main motivation was to address the **severe class imbalance** that undermined fraud detection in Synthetic Data 1. In the real dataset, fraudulent transactions were extremely rare compared to non-fraudulent ones, and CTGAN had learned to mostly generate non-fraud samples, leading to very poor recall. To counter this, a targeted **oversampling strategy** was introduced.

Fraudulent samples belonging to specific high-risk categories—grocery_pos, shopping_net, misc_net, and shopping_pos—were oversampled three times. These categories were chosen because they are historically associated with higher fraud rate in e-commerce contexts. Before resampling, the number of fraud cases in these focus categories was **5,214**, which increased to **15,642** after oversampling as shown in *Figure 5.3.54*. By oversampling frauds in these categories, the training distribution became more balanced and provided CTGAN with stronger fraud-related signals to learn from. This approach was designed to improve the generator's ability to model fraud patterns without overwhelming the training process with noise.

```
# Step 2.5: Oversample FRAUD in Focus Categories
focus_categories = ['grocery_pos', 'shopping_net', 'misc_net', 'shopping_pos']

# Split data
df_focus_fraud = df_cleaned[
    (df_cleaned['category'].isin(focus_categories)) & (df_cleaned['is_fraud'] == 1)
]
df_non_focus = df_cleaned[
    ~((df_cleaned['category'].isin(focus_categories)) & (df_cleaned['is_fraud'] == 1))
]

print("Fraud in focus categories before oversampling:", len(df_focus_fraud))

# Oversample fraud in focus categories (e.g., 3x)
df_focus_fraud_oversampled = df_focus_fraud.sample(
    n=len(df_focus_fraud) * 3, replace=True, random_state=42
)

# Combine
df_balanced = pd.concat([df_focus_fraud_oversampled, df_non_focus]).reset_index(drop=True)
```

```
Fraud in focus categories before oversampling: 5214
```

```
print("Fraud in focus categories after oversampling:",
      df_balanced[(df_balanced['category'].isin(focus_categories)) &
                  (df_balanced['is_fraud'] == 1)].shape[0])
```

```
Fraud in focus categories after oversampling: 15642
```

Figure 5.3.54: Oversampling of Fraudulent Transactions in Focus Categories

Another change from Data 1 was the adjustment of the training sample size. Instead of training CTGAN on 200,000 records, the dataset was reduced to 100,000 records before oversampling to reduce computational time and resource usage, as shown in *Figure 5.3.55*. The reduction was intentional as CTGAN often struggles to converge with very large datasets unless carefully

tuned, which also makes training much slower. A smaller but more balanced dataset made the training more efficient and allowed the model to better capture fraud-related patterns. The CTGAN configuration remained unchanged at 50 epochs with a batch size of 100, keeping the focus of this iteration on the effects of resampling rather than tuning.

Figure 5.3.55: Training Process of CTGAN with Reduced Dataset

The results of Synthetic Data 2 demonstrated the significant impact of oversampling. Table *5.3.17* showed that **recall increased dramatically** from **0.1114 to 0.7345**, indicating that the model could now detect the majority of fraud cases. **Precision** also **improved to 0.6395**, which indicated that a much larger proportion of fraud predictions were correct compared to Data 1. As a result, the **F1 score** rose to **0.6837** and **MCC** rose to **0.6348**, reflecting a stronger balance between recall and precision. The AUC also increased sharply to 0.8736, showing better separation between fraud and non-fraud classes. **Accuracy increased only slightly**, from **0.8987 to 0.9119**, which was expected. In imbalanced datasets, accuracy can be misleading, catching more frauds often adds some false positives, but this trade-off actually improves the dataset's value for fraud detection.

In short, Synthetic Data 2 marked a turning point in the synthetic generation process. By explicitly correcting the class imbalance through targeted oversampling of high-risk fraud categories, the dataset enabled the CTGAN to generate synthetic samples that were far more effective for fraud detection. While still limited in its feature representation, this version demonstrated that **resampling is a critical adjustment to improve recall and AUC**, directly addressing the weaknesses of the baseline Synthetic Data 1.

Synthetic Data 3

Synthetic Data 3 built upon the improvements of Data 2 and focused on **enhancing the feature representation** available to CTGAN. While oversampling had successfully improved fraud detection, the synthetic model still lacked access to temporal patterns, which are crucial in fraud behaviour. Fraudulent activity often happens at unusual times (e.g., late at night, certain days of the week), but in the baseline datasets, temporal information was either dropped or randomly restored, weakening its predictive value.

To address this, Data 3 introduced several engineered temporal features derived from trans_date_trans_time, as shown in *Figure 5.3.56*. Specifically, new variables were created for day of the week, hour of the day, and a binary flag is_night (set to 1 for transactions between 10 PM and 3 AM). These features provided CTGAN with structured representations of temporal context, allowing it to learn fraud-related timing patterns directly rather than relying on noisy random resampling of datetime.

```
# Extract time features BEFORE dropping datetime
if 'trans_date_trans_time' in df_cleaned.columns:
    df_cleaned['trans_date_trans_time'] = pd.to_datetime(
    df_cleaned['trans_date_trans_time'],
    format='%d/%m/%Y %H:%M',
    errors='coerce')
    df_cleaned['day_of_week'] = df_cleaned['trans_date_trans_time'].dt.dayofweek
    df_cleaned['hour'] = df_cleaned['trans_date_trans_time'].dt.hour
    df_cleaned['is_night'] = df_cleaned['hour'].apply(lambda h: 1 if h in [22,23,0,1,2,3] else 0)
    df_cleaned = df_cleaned.drop(columns=['trans_date_trans_time'])
```

Figure 5.3.56: Feature Engineering of Temporal Attributes

Another improvement in Data 3 was the use of stratified sampling when preparing the training data as shown in *Figure 5.3.57*. Instead of selecting random subsets, the sampling process preserved the fraud-to-non-fraud ratio across training splits. This adjustment ensured that CTGAN was consistently exposed to representative fraud patterns during training, further mitigating imbalance issues.

```
# 5. Stratified Sampling for CTGAN Training
train_df, _ = train_test_split(
    df_balanced,
    train_size=100000,  # sample size for CTGAN
    stratify=df_balanced['is_fraud'],
    random_state=42
)

print("Training sample shape:", train_df.shape)
print("Fraud ratio in training sample:", train_df['is_fraud'].mean())
```

```
Training sample shape: (100000, 12)
Fraud ratio in training sample: 0.01822
```

Figure 5.3.57: Use of Stratified Sampling for Balanced Training Data

The CTGAN training itself was also strengthened. Compared to the short 50-epoch run previously, Data 3 was trained for 300 epochs with pac=10 (PacGAN setting), as shown in *Figure 5.3.58*. The longer training duration gave the generator more opportunities to refine the synthetic data distribution, while the pac adjustment helped reduce mode collapse—a common problem where GANs generate overly similar samples instead of diverse patterns.

```
# 6. Train CTGAN
ctgan = CTGAN(epochs=300, batch_size=100, pac=10, verbose=True)
ctgan.fit(train_df, discrete_columns=categorical_cols)
```

Figure 5.3.58: CTGAN Training Settings in Dataset 3

The performance results highlighted both strengths and trade-offs. **Accuracy improved to 0.9295**, and **recall increased further to 0.7809**, showing that the inclusion of temporal features helped the model detect more fraud cases. However, **precision dropped to 0.5529** compared to Data 2, meaning the model generated more false positives alongside the true positives. This imbalance caused the **F1 score and MCC to fall slightly to 0.6474** and **0.6207** respectively, even though the overall **AUC rose to 0.9062**. In other words, the dataset became better at finding fraud but at the cost of sometimes mislabelling legitimate transactions.

In short, Synthetic Data 3 demonstrated the value of feature engineering, particularly the inclusion of temporal attributes. The results confirmed that fraud patterns are often time-dependent, and capturing this structure helped improve recall and AUC. The trade-off was a decline in precision, as the generator became more sensitive to potential fraud but less selective. This version showed that while oversampling was essential (Data 2), adding informative features (Data 3) was equally critical to move closer toward realistic and effective fraud detection.

Synthetic Data 4

SHAP Analysis before training for Synthetic Data 4

Before developing Synthetic Data 4, SHAP analysis was conducted to compare feature importance between the original training dataset and the synthetic dataset from Data 3 As shown in *Figure 5.3.59*, the two datasets displayed a very strong alignment, with a **Pearson correlation of 0.998** and a **cosine similarity of 0.998**. This indicates that the synthetic data broadly reproduced the same fraud-related signals as the real data. Key predictors such as transaction amount (amt), temporal indicators (is_night, hour, day), city population (city_pop), and age consistently ranked among the most influential features, with differences of **less than 10%,** as shown in *Table 5.3.18*.

```
Pearson correlation: 0.998
Cosine similarity: 0.998
```

Figure 5.3.59: Pearson Correlation and Cosine Similarity of Synthetic Dataset 3

	Feature	Real Data Importance	Synthetic Data Importance	Difference	Diff %
0	amt	0.2858	0.3005	-0.0147	5.14
5	is_night	0.0632	0.0574	0.0057	-9.05
21	cc_num_fraud_rate	0.0538	0.0669	-0.0131	24.42
4	hour	0.0349	0.0320	0.0029	-8.42
2	city_pop	0.0287	0.0285	0.0002	-0.79
10	category_other	0.0213	0.0236	-0.0023	10.77
7	distance	0.0136	0.0250	-0.0114	83.55
20	pop_group_Very Large Cities	0.0115	0.0113	0.0002	-1.50
3	age	0.0111	0.0107	0.0004	-3.98
6	day	0.0073	0.0075	-0.0001	1.61
8	category_grocery_pos	0.0057	0.0063	-0.0006	10.62
11	category_shopping_net	0.0047	0.0048	-0.0002	4.02
17	pop_group_Small Cities	0.0041	0.0038	0.0002	-5.69
1	gender	0.0028	0.0037	-0.0010	35.95
14	age_group_Adult	0.0018	0.0026	-0.0008	42.00

Table 5.3.18: SHAP Comparison Between Train Dataset and Synthetic Dataset 3

However, the SHAP comparison shown in *Table 5.3.18* also revealed important discrepancies. The distance feature, derived from latitude and longitude, showed overemphasized in synthetic data (+83.6%), suggesting CTGAN struggled to model raw geographic variables reliably. Demographic features such as Adult age group (+42%) and gender (+36%) also appeared more influential in the synthetic dataset than in the real one, raising concerns about demographic

bias. These findings showed that while Data 3 broadly captured fraud behaviour, it distorted or misrepresented certain signals, particularly around location and demographics.

Train for Synthetic Data 4

To address these limitations, Synthetic Data 4 introduced several key adjustments. The key improvement in Synthetic Data 4 was the introduction of the **distance feature**, calculated using the Haversine formula between customer and merchant coordinates and **capped at 160 km**. This cap was important because raw latitude and longitude are continuous, noisy, and difficult for CTGAN to model, often leading to unrealistic outputs. In fact, even small coordinate shifts could lead to extremely large distances—sometimes over 7,000 km, as observed in Synthetic Data 3 (see *Figure 5.3.60*), which clearly did not reflect real transaction behaviour. By replacing raw coordinates with capped distances during training and later recomputing them in post-processing, the dataset kept geographic patterns realistic while avoiding distorted outliers.

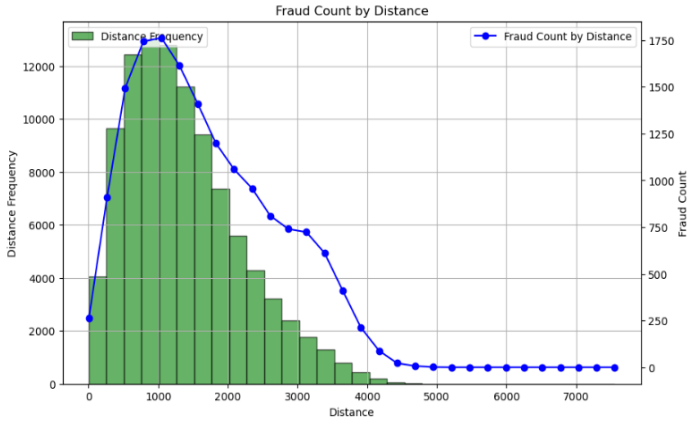


Figure 5.3.60: Distribution of Distance in Synthetic Dataset 3

```
# --- Extract distance (Haversine) ---
if {'lat', 'long', 'merch_lat', 'merch_long'}.issubset(df_cleaned_original.columns):
    def haversine(lat1, lon1, lat2, lon2):
        lat1, lon1, lat2, lon2 = map(np.radians, [lat1, lon1, lat2, lon2])
        dlat = lat2 - lat1
        dlon = lon2 - lon1
        a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
        c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
        return 6371 * c

    df_cleaned['distance'] = df_cleaned_original.apply(
        lambda row: haversine(row['lat'], row['long'], row['merch_lat'], row['merch_long']), axis=1
    )

    # Cap distance at 160 km
    df_cleaned['distance'] = df_cleaned['distance'].clip(upper=160)
```

Figure 5.3.61: Replacement of Raw Coordinates with Capped Distances During Training

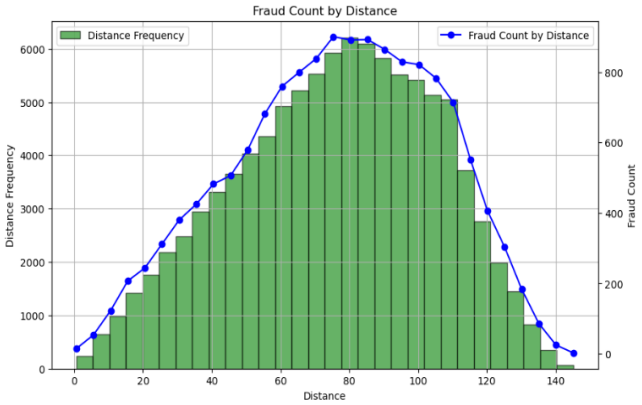


Figure 5.3.62: Distribution of Distance in Synthetic Dataset 4

Another major adjustment was the introduction of **age and age group,** shown in *Figure 5.3.63*. Instead of training on raw date of birth (dob), which introduces unnecessary complexity and difficult for CTGAN, the age was calculated at the transaction time and grouped into four categories: Young (0–18), Adult (19–44), Mid-Age (45–59), and Senior (60+). This transformation provided CTGAN with interpretable demographic features while reducing noise from exact dates. Furthermore, since SHAP analysis showed an overemphasis on Adults, fraudulent transactions in this group were reduced by 50% to prevent the generator from disproportionately modelling "Adult = fraud" behaviour. This balancing prevented the generator from disproportionately modelling Adult fraud at the expense of other age groups.

```
# Extract age & reduce Adult fraud ---
if 'dob' in df_cleaned_original.columns:
    df_cleaned = df_cleaned.reset_index(drop=True)
    df_cleaned_original = df_cleaned_original.reset_index(drop=True)

    df_cleaned['dob'] = pd.to_datetime(df_cleaned_original['dob'], format='%d/%m/%Y', errors='coerce')
    df_cleaned['age'] = (df_cleaned['trans_date_trans_time'] - df_cleaned['dob']).dt.days // 365
    median_age = df_cleaned['age'].median()
    df_cleaned['age'].fillna(median_age, inplace=True)

    age_bins = [0, 19, 45, 60, 100]
    age_labels = ['Young', 'Adult', 'Mid-Age', 'Senior']
    df_cleaned['age_group'] = pd.cut(df_cleaned['age'], bins=age_bins, labels=age_labels, right=False)
    df_cleaned['age_group'] = df_cleaned['age_group'].cat.add_categories('Unknown').fillna('Unknown')

    # Reduce fraud for Adults (keep 50%)
    adult_fraud = df_cleaned[(df_cleaned['age_group'] == 'Adult') & (df_cleaned['is_fraud'] == 1)]
    adult_fraud_reduced = adult_fraud.sample(frac=0.5, random_state=42)

    df_cleaned = pd.concat([
        df_cleaned[~((df_cleaned['age_group'] == 'Adult') & (df_cleaned['is_fraud'] == 1))],
        adult_fraud_reduced
    ]).reset_index(drop=True)
```

Figure 5.3.63: Code for Age Group Creation and Adult Fraud Balancing

After training, extensive post-processing was performed to restore dropped or transformed variables. The **distance feature** was recomputed in post-processing using the Haversine formula on customer and merchant coordinates and **capped at 160 km** to avoid unrealistic travel values, as shown in *Figure 5.3.64*. However, since distance, age, and age_group were

engineered only to guide CTGAN training, they were dropped from the final synthetic output. This allowed the dataset to retain the **same schema as the real data**, while still benefiting from the extra fraud-related signals during training.

```
# Safety check for distance if present
if 'distance' in synthetic_data.columns:
    max_distance = synthetic_data['distance'].max()
    print(f"⚠️ Max synthetic distance: {max_distance:.2f} km")
    if max_distance > 160:
        print("❌ Warning: Distance exceeded 160 km!")

⚠️ Max synthetic distance: 146.49 km
```

Figure 5.3.64: Maximum Synthetic Distance After Recalculation and 160 km Cap

SHAP Analysis after training for Synthetic Data 4

After training Synthetic Data 4, SHAP analysis showed that it achieved near-perfect alignment with the real dataset (Pearson = 1.0, Cosine = 1.0) as shown in *Figure 5.3.65*. Key features such as amount, is_night, cc_num_fraud_rate, hour, and city_pop closely matched their real-data importance (all within ±10%) as shown in *Table 5.3.19*.

```
🔗 Pearson correlation: 1.0
🔗 Cosine similarity: 1.0
```

Figure 5.3.65: Pearson Correlation and Cosine Similarity of Synthetic Dataset 4

	Feature	Real Data Importance	Synthetic Data Importance	Difference	Diff %
0	amt	0.2858	0.3072	-0.0214	7.50
5	is_night	0.0632	0.0619	0.0012	-1.95
21	cc_num_fraud_rate	0.0538	0.0608	-0.0070	13.01
4	hour	0.0349	0.0330	0.0019	-5.44
2	city_pop	0.0287	0.0301	-0.0014	4.73
10	category_other	0.0213	0.0228	-0.0015	7.27
7	distance	0.0136	0.0147	-0.0011	8.09
20	pop_group_Very Large Cities	0.0115	0.0114	0.0001	-0.57
3	age	0.0111	0.0118	-0.0007	6.23
6	day	0.0073	0.0080	-0.0006	8.47
8	category_grocery_pos	0.0057	0.0063	-0.0006	10.76
11	category_shopping_net	0.0047	0.0052	-0.0006	12.19
17	pop_group_Small Cities	0.0041	0.0041	-0.0000	1.17
1	gender	0.0028	0.0038	-0.0011	39.10
14	age_group_Adult	0.0018	0.0028	-0.0010	53.08

Table 5.3.19: SHAP Comparison Between Train Dataset and Synthetic Dataset 4

The **distance** feature, which was unstable in Data 3, was now **well-aligned (+8.1%)**, confirming the effectiveness of capping and recomputation. The main remaining differences were in **gender (+39%)** and **age_group_Adult (+53%)**, but their overall contribution to fraud detection was small.

Overall, the SHAP analysis confirmed that **Synthetic Data 4 represented a turning point**: by introducing spatial (distance) and demographic (age/age group) features, the model not only improved predictive performance but also **preserved the real-world importance structure of fraud signals** with near-perfect alignment. This provided strong evidence that the engineered features guided the CTGAN to capture fraud behaviour more realistically.

Results

The results showed a further step forward in balancing fraud detection. **Accuracy improved to 0.9433**, **recall increased** dramatically to **0.9140**, and **AUC reached 0.9636**, the highest achieved so far as shown in *Table 5.3.17*. **Precision**, however, **remained at 0.5800**, lower than Data 2 but slightly higher than Data 3, reflecting a middle ground between recall and precision. The **F1 score improved to 0.7100** and the **MCC rose to 0.7020**, confirming that the synthetic dataset captured fraud patterns with much stronger balance and reliability. These results confirmed that spatial and demographic features contributed significant predictive value, especially in helping the model distinguish fraudulent from non-fraudulent patterns in a more realistic way.

In short, Synthetic Data 4 highlighted the importance of feature engineering beyond time variables. By introducing distance and age-related attributes, the dataset captured crucial fraud signals that improved recall and AUC without overwhelming precision. Balancing Adult fraud cases further ensured a more representative dataset, avoiding demographic bias. This version established a strong foundation by showing that integrating temporal, spatial, and demographic features together with resampling strategies yields more robust synthetic data for fraud detection.

Synthetic Data 5

Synthetic Data 5 introduced two major refinements over the previous version: fraud ratio control and explicit metadata guidance.

The first adjustment was **fraud ratio control**. Unlike Data 4, which oversampled fraud using a fixed multiplier, Data 5 explicitly targeted a **15% fraud prevalence** in the training set as shown in *Figure 5.3.66*. This rate is much higher than in real life but still believable. The motivation was to **improve precision** by reducing the overwhelming dominance of non-fraud cases while still giving CTGAN enough fraud examples to learn meaningful patterns. Fraud rows—especially from high-risk categories such as *grocery_pos, shopping_net, misc_net,* and *shopping_pos*—were oversampled dynamically until the 15% threshold was reached.

```
# --- Step 3: Oversample fraud ---
focus_categories = ['grocery_pos', 'shopping_net', 'misc_net', 'shopping_pos']
df_focus_fraud = df_cleaned[
    (df_cleaned['category'].isin(focus_categories)) & (df_cleaned['is_fraud'] == 1)
]
df_non_focus = df_cleaned[
    ~((df_cleaned['category'].isin(focus_categories)) & (df_cleaned['is_fraud'] == 1))
]

target_ratio = 0.15
num_non_fraud = len(df_non_focus)
num_fraud_needed = int((target_ratio / (1 - target_ratio)) * num_non_fraud)

df_focus_fraud_oversampled = df_focus_fraud.sample(
    n=num_fraud_needed, replace=True, random_state=42
)

df_balanced = pd.concat([df_focus_fraud_oversampled, df_non_focus]).reset_index(drop=True)
print("Fraud ratio after oversampling:", df_balanced['is_fraud'].mean())
```

```
Fraud ratio after oversampling: 0.15150145273561968
```

Figure 5.3.66: Training Data Distribution After Adjusting Fraud Rate to 15%

The second adjustment was the use of SingleTableMetadata to **explicitly define feature types** as shown in *Figure 5.3.67*. Earlier versions occasionally misclassified variables (e.g., treating city_pop as categorical), which led CTGAN to generate misaligned distributions. By correcting these definitions, CTGAN was able to interpret numerical, categorical, and datetime features more reliably, improving stability.

```
# --- Step 4: Build Metadata for CTGAN ---
metadata = SingleTableMetadata()
metadata.detect_from_dataframe(data=train_df)
metadata.update_column("city_pop", sdtype="numerical")
metadata.save_to_json('/content/drive/MyDrive/Colab Notebooks/syn 7/metadata_ctgan.json')
```

Figure 5.3.67: Defining Feature Types Using SingleTableMetadata

Other aspects from Data 4 were retained as standard practice: reducing Adult fraud cases by 50% to mitigate demographic bias, restoring schema consistency in post-processing, and using CTGAN for training. However, **training epochs were reduced to 200** (compared to 300 in Data 4), as shown in *Figure 5.3.68.* This is because improved metadata stability lessened the need for extended runs or PacGAN regularization.

```
# --- Step 5: Train CTGAN ---
ctgan = CTGANSynthesizer(metadata, epochs=200, batch_size=100, verbose=True)
ctgan.fit(train_df)
```

```
/usr/local/lib/python3.11/dist-packages/sdv/single_table/base.py:163: FutureWarning: The
'SingleTableMetadata' is deprecated. Please use the new 'Metadata' class for synthesizers.
  warnings.warn(DEPRECATION_MSG, FutureWarning)
Gen. (0.85) | Discrim. (0.11): 100%|██████████| 200/200 [2:28:53<00:00, 44.67s/it]
```

Figure 5.3.68: CTGAN Training Configuration in Data 5

The evaluation of Synthetic Data 5 confirmed that the main improvement was in **precision**, which rose to **0.6722**, the highest across all versions. This meant that a larger share of detected fraud cases were truly fraudulent, reducing false alarms compared to earlier datasets. However, this gain came with a **trade-off in recall**, which decreased to **0.8193** (from 0.9140 in Data 4), showing that the model caught slightly fewer fraud cases overall. Despite this, the F1 score improved to **0.7385**, showing that the dataset produced a better balance between catching fraud (recall) and avoiding false alarms (precision). The **MCC of 0.6679** further demonstrated a robust overall correlation between predictions and true labels, confirming that the improvements were not one-sided. On the other hand, **accuracy dropped to 0.8810** and **AUC declined to 0.8990**, indicating weaker modelling of normal transaction behaviour.

In summary, Synthetic Data 5 showed that controlling fraud ratio and improving feature handling can shift performance trade-offs. It gave the best fraud detection so far, though with weaker overall transaction realism (lower AUC).

Synthetic Data 6

Synthetic Data 6 marked the first shift from GAN-based models to a variational autoencoder approach, **replacing CTGAN with TVAE** (Tabular Variational Autoencoder). This change was motivated by the limitations of GANs in modelling continuous numerical features such as transaction amount (amt), distance, and age. CTGAN had shown improvements with feature engineering and oversampling, but it was unstable and sometimes collapsed, producing less diverse data. In contrast, TVAE learns a smooth internal representation of the data, making it more stable and better at handling both numbers and categories.

For training, TVAE was configured with 200 epochs and a batch size of 100 as shown in *Figure 5.3.69*. Unlike GAN-based models, TVAE did not require PacGAN or extended epochs to prevent collapse, since its architecture naturally models both fraud and non-fraud cases in a smooth latent space. This makes training more stable and efficient, avoiding the adversarial competition present in CTGAN, which often requires extra epochs or stabilizing tricks to converge. As a result, TVAE converged reliably within fewer epochs and completed training faster than CTGAN, reducing computational time while maintaining high-quality synthetic data.

```
tvae = TVAESynthesizer(metadata, epochs=200, batch_size=100, verbose=True)
tvae.fit(train_df)
```

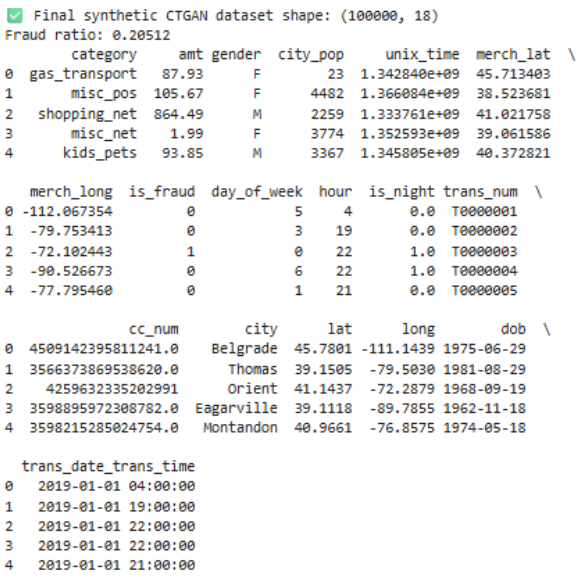
```
C:\Users\Dell\anaconda3\Lib\site-packages\sdv\single_table\base.py:163: FutureWarning:

The 'SingleTableMetadata' is deprecated. Please use the new 'Metadata' class for synthesizers.

Loss: -2.947: 100%|██████████| 200/200 [49:03<00:00, 14.72s/it]
```

Figure 5.3.68: TVAE Training Configuration in Data 6

Post-processing followed the same steps as in Data 5. After generation, features such as dob, merchant coordinates, and card numbers (cc_num) were restored from the original dataset. Distance was recomputed using restored coordinates, and engineered variables (distance, age, age_group) were dropped to maintain schema consistency. This ensured that the final synthetic dataset aligned structurally with the raw dataset while retaining the benefits of engineered features during training.



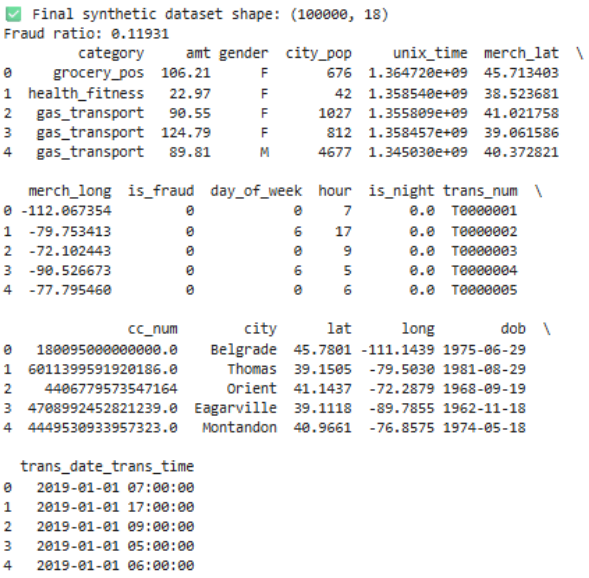
```
✅ Final synthetic CTGAN dataset shape: (100000, 18)
Fraud ratio: 0.20512
       category     amt gender  city_pop     unix_time  merch_lat  \
0  gas_transport   87.93      F        23  1.342840e+09  45.713403
1      misc_pos  105.67      F      4482  1.366084e+09  38.523681
2   shopping_net  864.49      M      2259  1.333761e+09  41.021758
3      misc_net    1.99      F      3774  1.352593e+09  39.061586
4     kids_pets   93.85      M      3367  1.345805e+09  40.372821

   merch_long  is_fraud  day_of_week  hour  is_night trans_num  \
0 -112.067354         0            5     4       0.0  T0000001
1  -79.753413         0            3    19       0.0  T0000002
2  -72.102443         1            0    22       1.0  T0000003
3  -90.526673         0            6    22       1.0  T0000004
4  -77.795460         0            1    21       0.0  T0000005

              cc_num        city      lat      long        dob  \
0  4509142395811241.0    Belgrade  45.7801 -111.1439 1975-06-29
1  3566373869538620.0      Thomas  39.1505  -79.5030 1981-08-29
2    4259632335202991      Orient  41.1437  -72.2879 1968-09-19
3  3598895972308782.0  Eagarville  39.1118  -89.7855 1962-11-18
4  3598215285024754.0   Montandon  40.9661  -76.8575 1974-05-18

  trans_date_trans_time
0   2019-01-01 04:00:00
1   2019-01-01 19:00:00
2   2019-01-01 22:00:00
3   2019-01-01 22:00:00
4   2019-01-01 21:00:00
```

Figure 5.3.70: Final Synthetic Dataset Generated by CTGAN in Data 5



```
✅ Final synthetic dataset shape: (100000, 18)
Fraud ratio: 0.11931
        category     amt gender  city_pop     unix_time  merch_lat  \
0     grocery_pos  106.21      F       676  1.364720e+09  45.713403
1  health_fitness   22.97      F        42  1.358540e+09  38.523681
2   gas_transport   90.55      F      1027  1.355809e+09  41.021758
3   gas_transport  124.79      F       812  1.358457e+09  39.061586
4   gas_transport   89.81      M      4677  1.345030e+09  40.372821

   merch_long  is_fraud  day_of_week  hour  is_night trans_num  \
0 -112.067354         0            0     7       0.0  T0000001
1  -79.753413         0            6    17       0.0  T0000002
2  -72.102443         0            0     9       0.0  T0000003
3  -90.526673         0            6     5       0.0  T0000004
4  -77.795460         0            0     6       0.0  T0000005

              cc_num        city      lat      long        dob  \
0   180095000000000.0    Belgrade  45.7801 -111.1439 1975-06-29
1  6011399591920186.0      Thomas  39.1505  -79.5030 1981-08-29
2    4406779573547164      Orient  41.1437  -72.2879 1968-09-19
3  4708992452821239.0  Eagarville  39.1118  -89.7855 1962-11-18
4  4449530933957323.0   Montandon  40.9661  -76.8575 1974-05-18

  trans_date_trans_time
0   2019-01-01 07:00:00
1   2019-01-01 17:00:00
2   2019-01-01 09:00:00
3   2019-01-01 05:00:00
4   2019-01-01 06:00:00
```

Figure 5.3.71: Final Synthetic Dataset Generated by TVAE in Data 6

The results demonstrated a big improvement in performance. **Recall reached 0.9451**, the highest of all datasets so far, showing that TVAE was effective at detecting fraudulent transactions. **AUC also improved to 0.9683**, setting a new benchmark for discrimination between fraud and non-fraud. **Precision, at 0.6426**, was slightly lower than Data 5 but still competitive, while the **F1 score rose to 0.7650**, reflecting a stronger balance between recall and precision. **Accuracy also improved to 0.9307**, recovering much of the loss seen in Data 5.

In short, Synthetic Data 6 highlighted the advantages of model architecture choice in synthetic data generation. By moving from CTGAN to TVAE, the dataset gained much stronger generalization over continuous features, leading to dramatic improvements in recall and AUC. Although some precision was sacrificed compared to CTGAN, the overall balance (F1) and discriminative power (AUC) were superior. This version demonstrated that TVAE was better suited for fraud detection tasks where continuous variables play a critical role in distinguishing fraudulent behaviour.

Synthetic Data 7

Synthetic Data 7 expanded upon the previous TVAE experiment by scaling up the training process. While Data 6 had already demonstrated the stability and representational power of TVAE over CTGAN, its training was limited to 100,000 rows. Data 7 increased the training sample size to 500,000 rows as shown in *Figure 5.3.72*, providing TVAE with significantly more examples of both fraud and non-fraud patterns to learn from. The objective of this adjustment was to test whether a larger training base would enable TVAE to generate more realistic synthetic data, especially in capturing rare fraud patterns.

The TVAE configuration differed from Data 6 by reducing the epochs from 200 to 100. Despite the shorter training duration, the much larger dataset ensured that TVAE was exposed to a richer and more diverse set of features.

```
train_df, _ = train_test_split(
    df_balanced,
    train_size=500000,
    stratify=df_balanced['is_fraud'],
    random_state=42
)

tvae = TVAESynthesizer(metadata, epochs=100, batch_size=100, verbose=True)
tvae.fit(train_df)
```

Figure 5.3.72: TVAE Training Configuration in Data 7

The results confirmed the benefit of scaling up training size. **Recall increased** further to **0.9646**, demonstrating that TVAE trained on more data was even more effective at detecting fraud. **Precision** also improved markedly to **0.7133**, reducing false alarms compared to Data 6. These gains translated into an **F1 score of 0.8202**, an **MCC of 0.7980**, and an **AUC of 0.9808**—the highest values achieved so far. **Accuracy** also rose to **0.9400**, showing consistent overall improvements across all evaluation metrics.

In summary, Synthetic Data 7 highlighted the importance of training size in synthetic data generation. By giving TVAE five times more training examples, the model was able to generalize better across both fraud and non-fraud transactions. Compared to Data 6, this version achieved stronger recall without sacrificing precision, leading to better balance (F1 score), correlation (MCC), and discrimination (AUC). This confirmed that, beyond choosing the right model, using a larger training sample size is a key factor in producing synthetic data that improves fraud detection model performance.

5.3.11 Model and Pipeline Export

Before deploying the model into Power BI, the trained **Random Forest model (oversampling + default settings)** and its preprocessing pipeline were exported as shown in *Figure 5.3.73*. During model development, several transformations were applied, including gender encoding, one-hot encoding for transaction categories, and target encoding for credit card numbers. To ensure these transformations would be applied consistently during deployment, both the pipeline and the trained model were serialized and saved using the Joblib library. By exporting the pipeline alongside the model, the system guarantees that identical feature engineering and encoding steps are performed on any new data, reducing the risk of inconsistency and ensuring portability across different environments.

```
# Create the pipeline with your custom preprocessor
encoder = Pipeline([
    ('custom_preprocessing', CustomPreprocessor())
])
# Fit the pipeline (if needed, e.g., before transforming)
encoder.fit(df)
# Transform the data
df = encoder.transform(df)
# Save the pipeline
joblib.dump(encoder, 'custom_encoding_pipeline.joblib')
```

```
# ---- BUILD PIPELINE ----
pipeline = Pipeline(steps=[
    ('target_encoder', TargetEncoder(col='cc_num', target_col='is_fraud')),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Fit the pipeline on training data
pipeline.fit(X_train, y_train)

# Save pipeline
joblib.dump(pipeline, "target_encoding_rf_oversampling_pipeline.joblib")
```

Figure 5.3.73: Saving the preprocessing pipeline and Random Forest model using Joblib

5.3.12 Power BI Deployment

Data Source Connection

Following the model export, the **data source** was connected to Power BI. Transaction data was retrieved directly via a OneDrive link as shown in *Figure 5.3.74*, allowing the dashboard to work with updated records when the dataset is **refreshed** in Power BI. For this purpose, the **Kaggle test set** was sampled to 100,000 records. This dataset size was chosen to strike a balance between providing sufficient representation of fraud cases and ensuring efficient processing within Power BI's Python execution environment. The use of the Kaggle test set in Power BI is further justified in Chapter 6, where it is compared against synthetic alternatives.

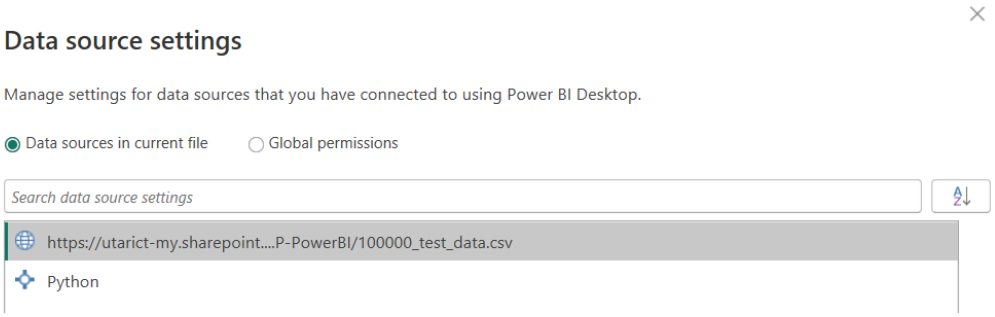


Figure 5.3.74: Power BI data source connection from OneDrive

Python Script Integration

Once the data connection was established, the dataset could be accessed and inspected via **Transform Data (Power Query Editor)** in Power BI. A Python script was embedded in Power BI to load both the **preprocessing pipeline** and **the trained model**. When new data is imported, the script applies the same preprocessing transformations as during training, including categorical encoding and target encoding. The processed features are then aligned to the expected order used by the model, as shown in *Figure 5.3.75*.

```
# Load dataset
df = dataset.copy()

# Load pipeline
preprocessor = joblib.load(r'C:\Users\Dell\OneDrive - Universiti Tunku Abdul Rahman\FYP-PowerBI\custom_encoding_pipeline.joblib')
model_pipeline = joblib.load(r'C:\Users\Dell\OneDrive - Universiti Tunku Abdul Rahman\FYP-PowerBI\target_encoding_rf_oversampling_pipeline.joblib')

# Preprocess data
X_encoded = preprocessor.transform(df)
y = X_encoded['is_fraud'] if 'is_fraud' in X_encoded.columns else None
X = X_encoded.drop(columns=['is_fraud'], errors='ignore')
```

Figure 5.3.75: Python script preprocessing new transaction data

After this step, the Random Forest model predicts and generates two outputs for each transaction: a **binary classification (predicted_fraud)** indicating whether the transaction is fraudulent, and **a probability score (fraud_prob)** representing the likelihood of fraud, as shown in *Figure 5.3.76*.

```
# Predict class labels
predictions = model_pipeline.predict(X)
df['predicted_fraud'] = predictions
# dataset['predicted_fraud'] = predictions

y_pred_proba = model_pipeline.predict_proba(X)[:, 1]
df['fraud_prob'] = y_pred_proba
```

Figure 5.3.76: Python script generating predictions

Finally, the script produces **evaluation results**. If the dataset contains true fraud labels (is_fraud), **performance metrics** such as accuracy, recall, precision, F1-score, Matthews Correlation Coefficient (MCC), and Area Under the Curve (AUC) are computed and presented in tabular form within Power BI. If the dataset does not contain labels, the system outputs only fraud predictions and probability scores, as shown in *Figure 5.3.77*. The generated outputs are then ready for visualization, as shown in *Figure 5.3.78*, forming the foundation for the dashboard development stage of deployment.

```
if y is not None:
    # Calculate metrics
    accuracy = accuracy_score(y, predictions)
    recall = recall_score(y, predictions)
    precision = precision_score(y, predictions)
    f1 = f1_score(y, predictions)
    mcc = matthews_corrcoef(y, predictions)
    auc_score = roc_auc_score(y, y_pred_proba)

    # Format metrics into a table
    metrics_df = pd.DataFrame({
        "Accuracy": [round(accuracy, 4)],
        "Recall": [round(recall, 4)],
        "Precision": [round(precision, 4)],
        "F1 Score": [round(f1, 4)],
        "MCC": [round(mcc, 4)],
        "AUC": [round(auc_score, 4)]
    })

    # Output metrics table
    output = metrics_df
else:
    # If no labels are available, return dataset with predictions
    df['predicted_fraud'] = predictions
    df['fraud_prob'] = y_pred_proba
    output = df
```

Figure 5.3.77: Python script generating evaluation metrics

1.2 Accuracy	1.2 Recall	1.2 Precision	1.2 F1 Score	1.2 MCC	1.2 AUC
0.9991	0.8473	0.9597	0.9	0.9013	0.9828

Figure 5.3.78: Output generated by Python script

5.3.13 Dashboard Development

The Power BI dashboard was developed as the final stage of the system implementation, providing the interactive interface through which users can monitor and analyse fraud detection outcomes. The design was guided by the wireframe created earlier (see Chapter 4), ensuring a consistent structure and smooth navigation across different analytical perspectives. Navigation follows a **hierarchical structure**, with the **Homepage** and **Overview Page** serving as **central hubs**, with other pages accessed through them for a clear, structured flow.

The **Homepage** acts as the entry point, containing interactive buttons with tooltips that direct users to the respective analysis pages.

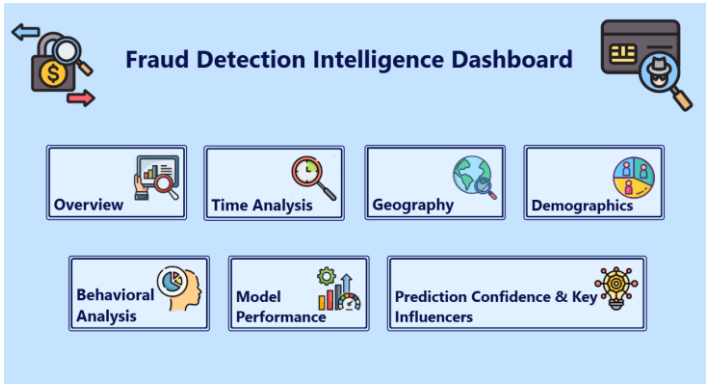


Figure 5.3.79: Homepage

The **Overview Page** presents key performance indicators (KPIs) through card visuals, summarising overall transaction activity, fraud statistics, and model performance. To complement the KPIs, trend charts illustrate changes in fraud amount and fraud rate over time, providing a quick view of both scale and temporal behaviour.

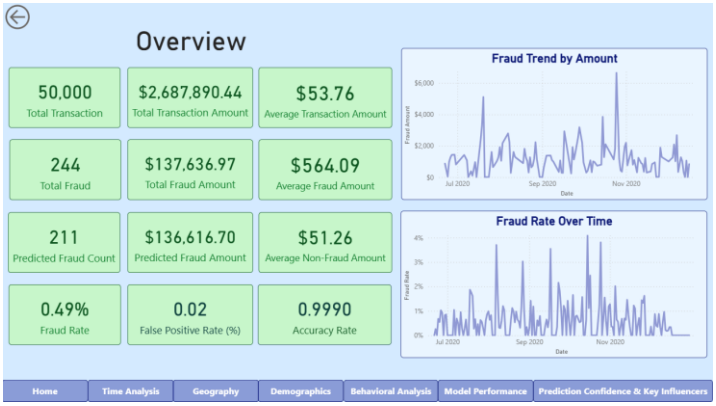


Figure 5.3.80: Overview Page

The **Time Analysis Page** examines fraud patterns by day, hour, and month, with additional views comparing daytime versus nighttime fraud activity and fraud trends across days in a month.

Figure 5.3.81: Time Analysis Page

The **Geography Page** highlights fraud distribution across city sizes and transaction distances, supplemented with an interactive hotspot map.

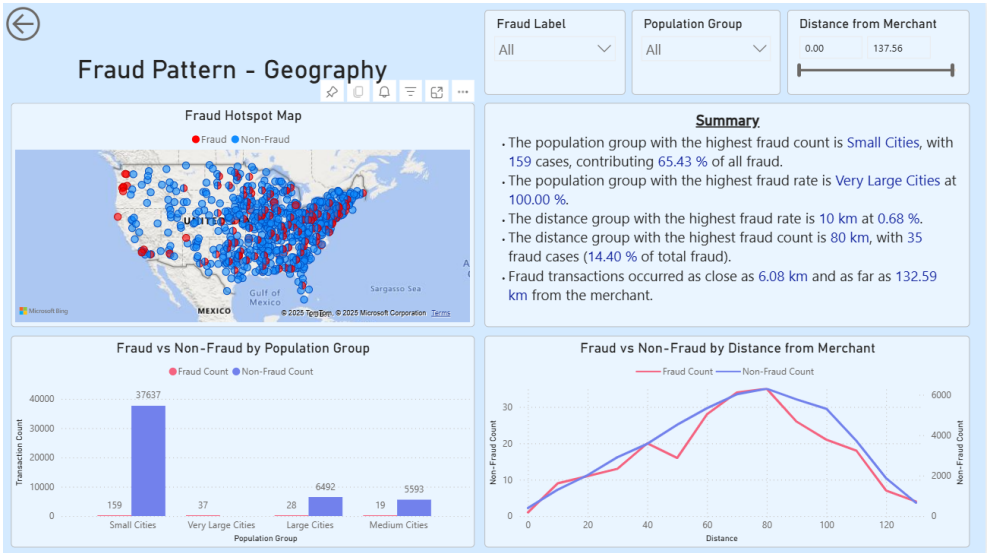


Figure 5.3.82: Geography Page

The **Demographics Page** analyses fraud by age and gender, including a heatmap to show intersections, and supports drill-through to card-level details for deeper investigation.

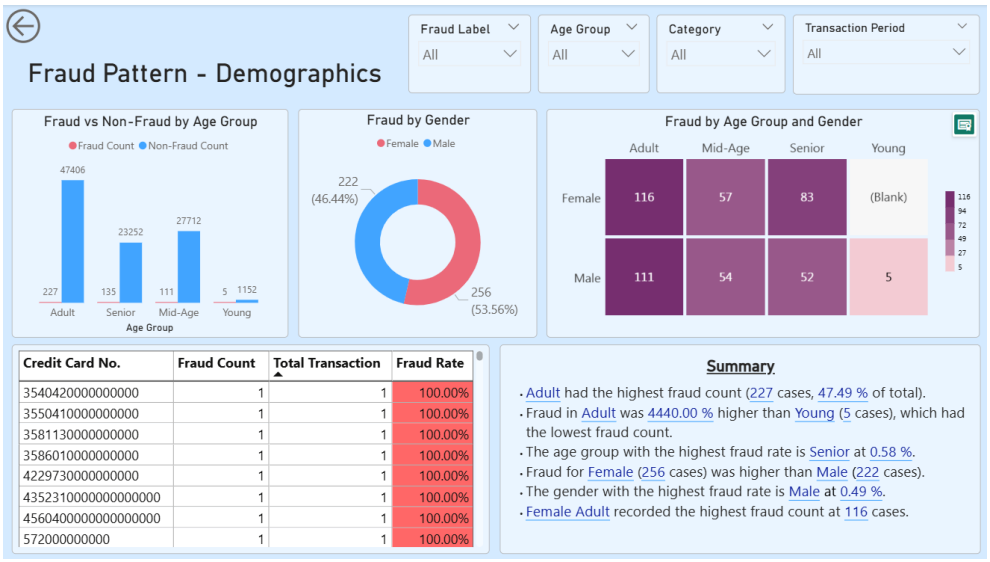


Figure 5.3.83: Demographics Page

Fraud behaviour by category and spending patterns is analysed in the **Behavioural Analysis Page**, which compares different category transactions and evaluates fraud by total and average amounts.

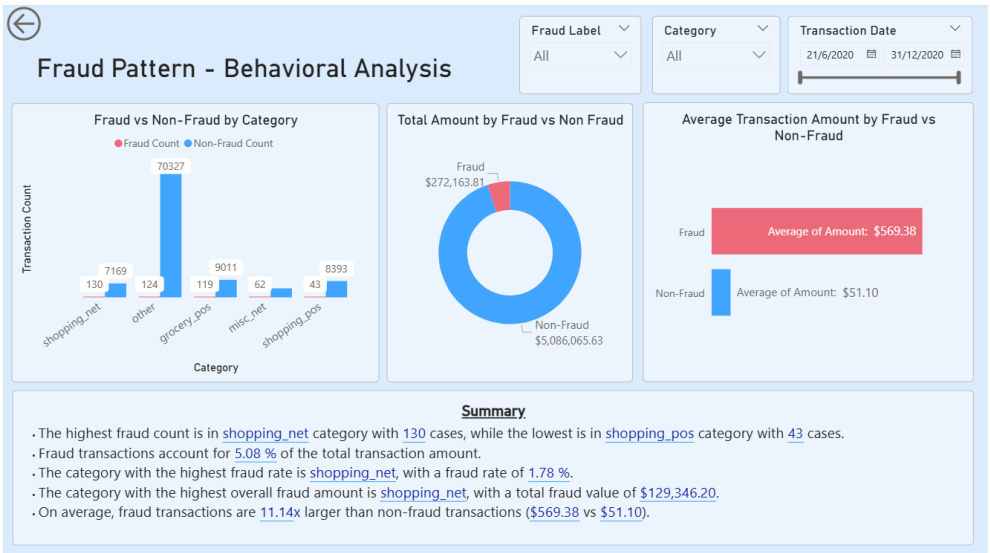


Figure 5.3.84: Behavioural Analysis Page

Model performance is assessed in the **Model Performance Page**, which includes a confusion matrix, performance metric cards, and misclassification breakdowns by category and demographic group. **Conditional formatting** is applied to performance metrics: values below 0.9 are highlighted in yellow, and values below 0.75 are highlighted in red.

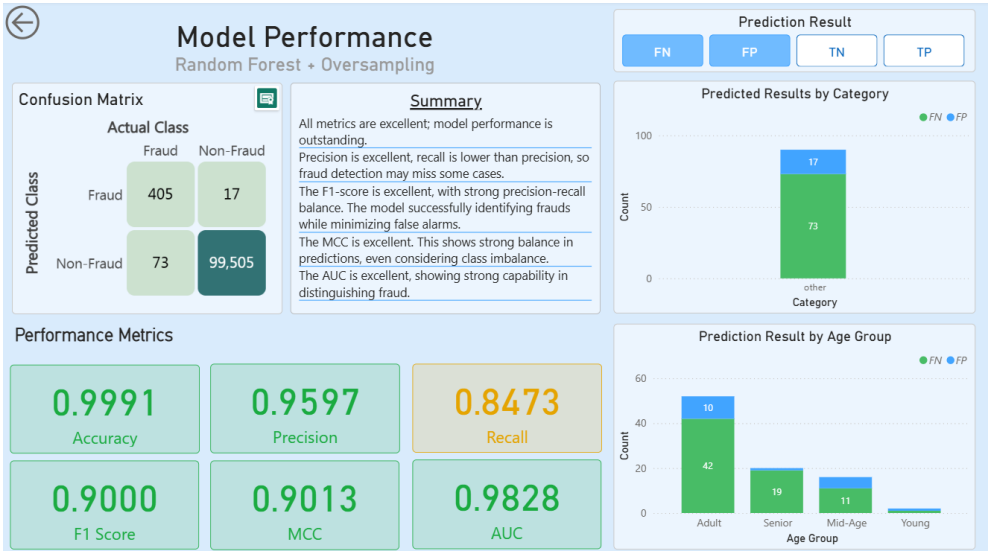


Figure 5.3.85: Model Performance Page

The **Prediction Confidence & Key Influencers Page** shows model certainty, key factors driving fraud predictions, and a table of transactions—highlighting incorrect predictions while leaving fraud cases unhighlighted. Users can drill through to view each transaction.

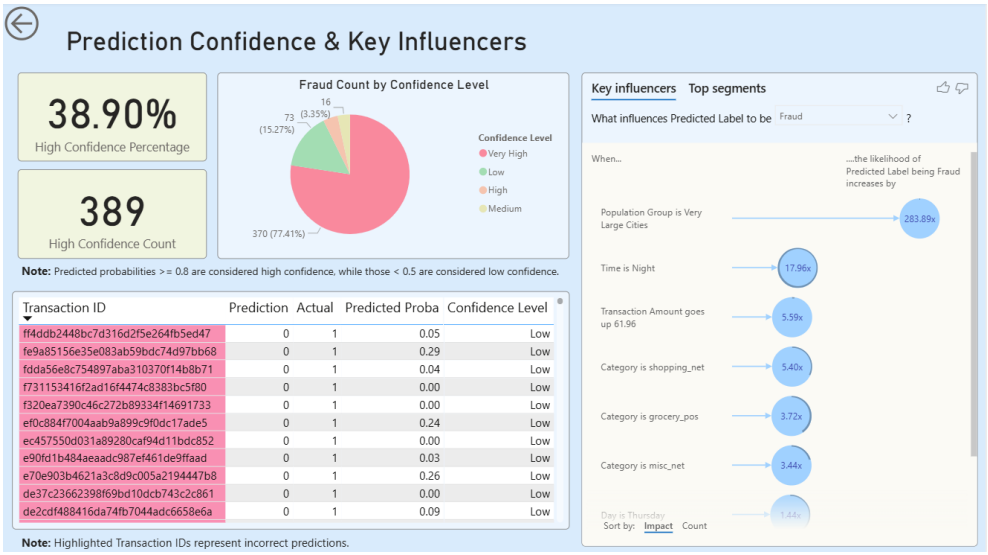


Figure 5.3.86: Prediction Confidence & Key Influencers Page

In addition to the main analysis pages, two **drill-through pages** were developed to support detailed investigation, providing a seamless workflow from aggregated views to case-level insights, supporting in-depth fraud investigation.

The **Credit Card Transactions Page** allows users to examine all transactions linked to a specific card, supported by fraud metrics, transaction history, and visual trend analysis. **KPI cards with a non-zero fraud count** are highlighted to draw attention. In the transaction history table, **fraud cases** and transactions with **incorrect predictions** are highlighted for easy identification.

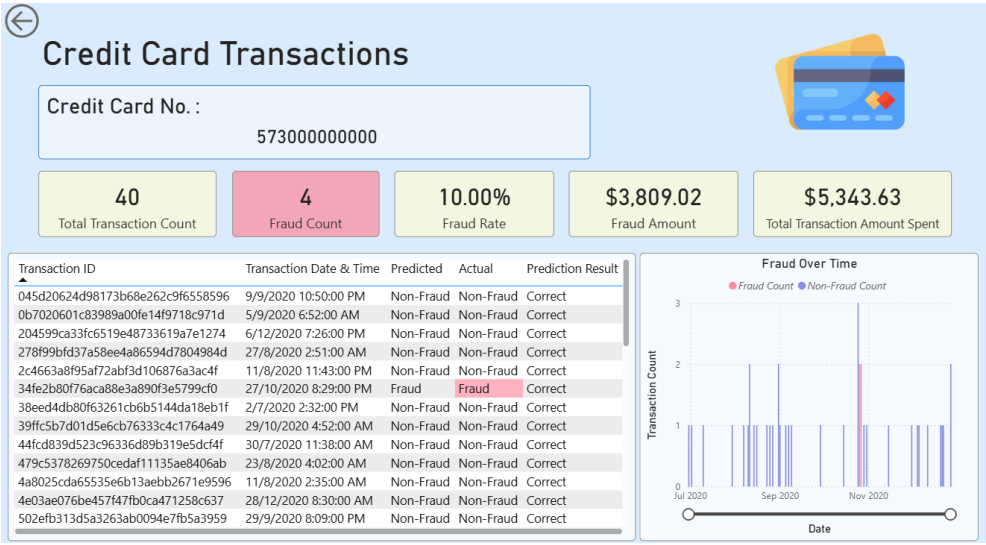


Figure 5.3.87: Credit Card Transactions Page (drill through from Demographics Page)

From there, users can navigate to the **Transaction Details Page**, which provides full information for a single transaction, including prediction details, related credit card data, and cardholder demographics. For the KPI cards, **conditional formatting is applied to highlight key information**: actual or predicted fraud labels are shown in red, fraud probability uses a gradient from green (0%) to red (100%), and the predicted result is coloured green if correct and red if incorrect.

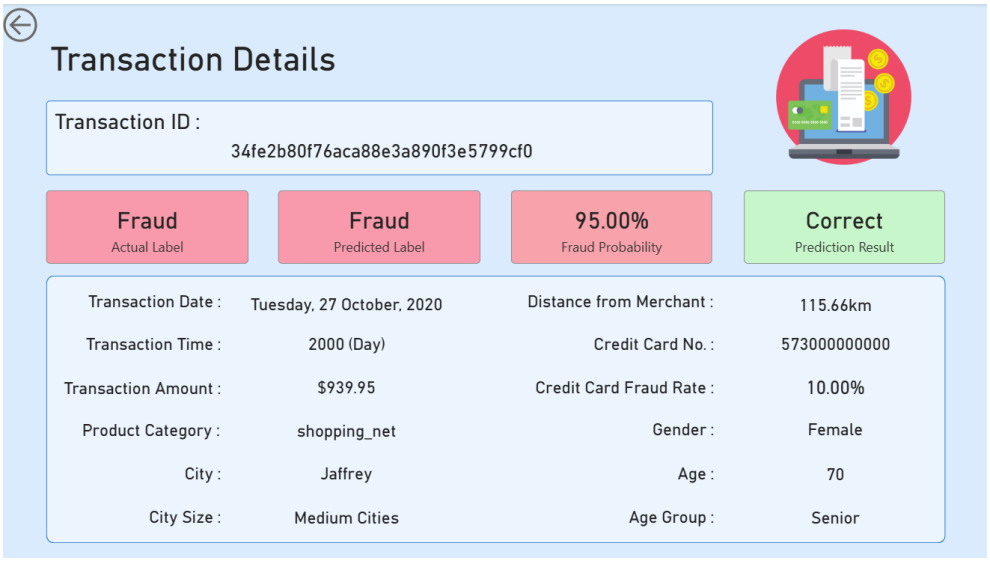


Figure 5.3.88: Transaction Details Page (drill through from Credit Card Transactions Page)

To enhance usability, **slicers and filters** were added to allow dynamic exploration of fraud patterns by date, category, demographic segment, and location. **Tooltips** were configured for charts to provide detailed values on demand, maintaining visual clarity while retaining precision. **Smart Narrative** were also applied to generate automated insights. Together, these features create an interactive, user-friendly system capable of supporting fraud analysts in monitoring patterns, identifying risks, and evaluating model reliability.

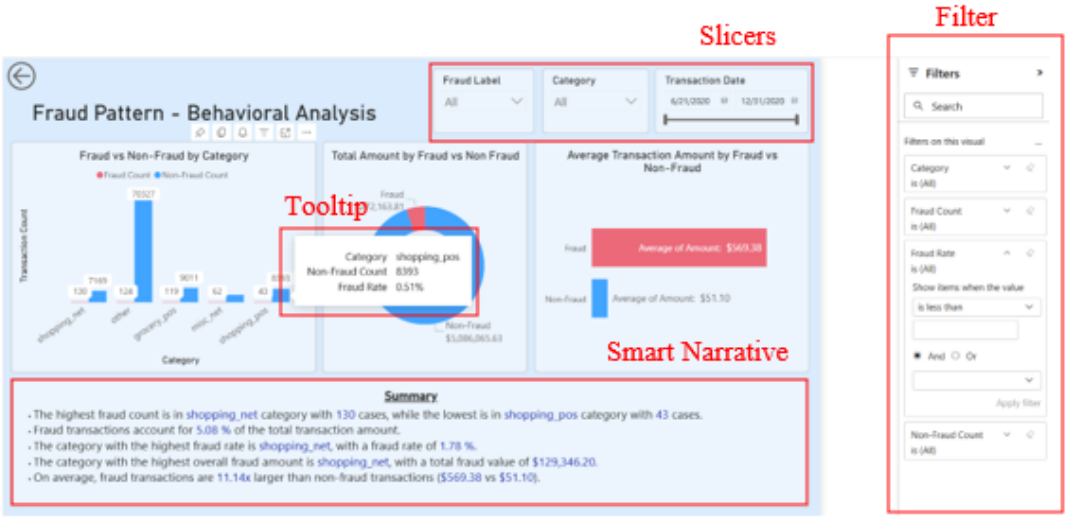


Figure 5.3.89: Interactive Dashboard Components Showing Slicers, Filters, Tooltips and Smart Narrative

The dashboard was also designed with a **mobile layout** in mind, ensuring key metrics and visualizations remain accessible and readable on smaller screens. Example pages of the dashboard on mobile view are shown in *Figure 5.3.90*.

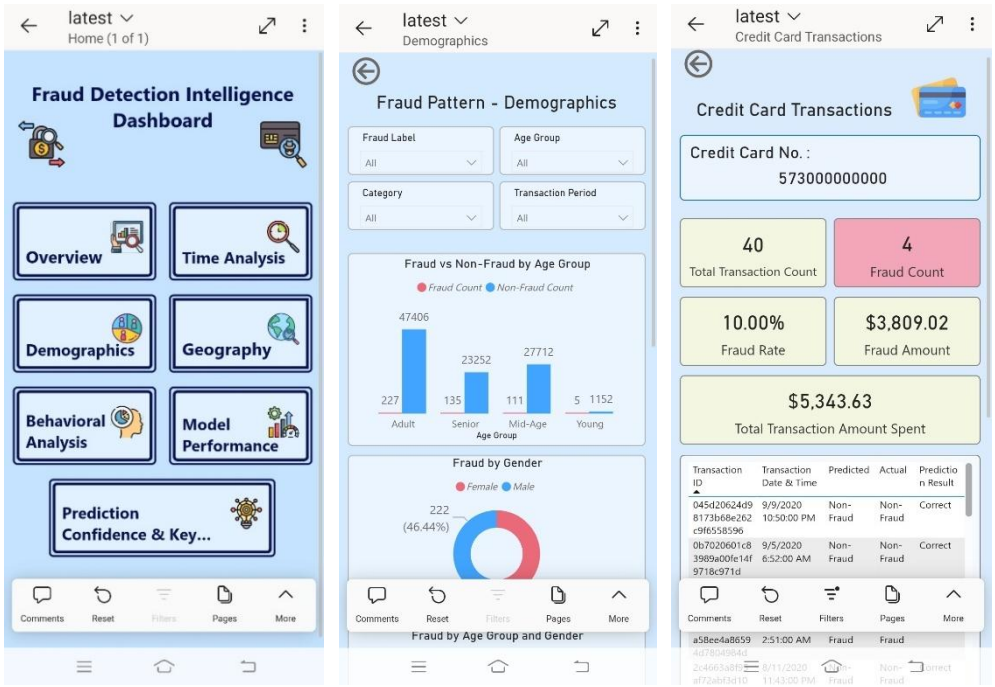


Figure 5.3.90: Mobile layout examples of the Fraud Detection Dashboard

5.3.14 Implementation Issues and Challenges

Publicly available fraud detection datasets often contain missing values, noise, and outliers, which can undermine model reliability, making **proper selection and preprocessing** essential. In this project, the first dataset tested was found unsuitable due to **weak feature correlations**, limiting its usefulness for fraud analysis. Many Kaggle datasets are also **anonymized**, replacing feature names with generic labels, which hinders interpretability for dashboard development. Additionally, the strong **class imbalance**, where fraud cases are rare, requires resampling techniques. After evaluation, a second dataset was chosen for its stronger feature relationships and practical relevance.

The implementation phase was marked by several significant challenges that impacted the project's workflow. A primary obstacle was the **severe computational and hardware constraints** faced during model training. Running complex algorithms such as Random Forest

and AdaBoost on a large dataset, combined with **extensive hyperparameter tuning** through Grid Search and Randomized Search, was extremely **time-consuming**, often taking hours for a single run. This issue was further showed by **synthetic data generation using CTGAN and TVAE**, which, even with Google Colab's GPU resources, required many hours of processing for a single experiment.

Another critical challenge was the **instability of Google Colab sessions**. Long-running experiments were frequently **interrupted due to idle timeouts, session duration limits**, or memory overuse. On several occasions, runs that had progressed halfway stopped unexpectedly when left unattended, forcing complete reruns and wasting computational resources. This instability proved particularly problematic during resource-intensive tasks such as synthetic data generation using CTGAN, where interruptions could mean the **loss of hours of work**.

From a dashboard development perspective, a key challenge was **designing an intuitive UI for a complex, multi-faceted analysis**. Creating a logical navigation structure from the homepage to various drill-through pages (e.g., from Demographics to Credit Card Transactions to Transaction Details) required careful planning to ensure a seamless user journey for fraud analysts. Implementing advanced features like conditional formatting (e.g., highlighting high-risk cards, colouring metrics based on performance) and interactive elements (slicers, filters, tooltips) without making the dashboard visually cluttered or overwhelming was a delicate balancing act between functionality and usability.

5.3.15 Concluding Remark

In conclusion, this chapter outlined the full implementation of the fraud detection system, covering software setup, data preprocessing, model development, synthetic data generation, and dashboard deployment. After testing multiple datasets and algorithms, a Random Forest model with oversampling was selected for its strong performance, achieving high recall and F1-score in detecting fraud. The model was successfully integrated into Power BI, creating an interactive dashboard that turns predictive analytics into actionable insights. Despite challenges with dataset quality, computational limits, Google Colab instability, and dashboard design complexity, the system delivers a robust and practical foundation for real-world fraud monitoring and analysis.

CHAPTER 6

System Evaluation and Discussion

6.1 Comparison of Test Set

Before selecting the final dataset for deployment in the Power BI dashboard, two candidate test sets were evaluated to determine which would provide the most reliable and representative assessment of model performance: a **real Kaggle test set** and a **synthetic test set** generated through the model pipeline. The performance metrics for both sets are summarized in *Table 6.1.1*.

	Kaggle Test Set	Synthetic Test Set
Accuracy	0.9991	0.9400
Recall	0.8410	0.9646
Precision	0.9640	0.7133
F1-Score	0.8983	0.8202
MCC	0.9000	0.7980
AUC	0.9552	0.9808

Table 6.1.1: Performance Comparison of Kaggle and Synthetic Test Sets

The **real test set**, sampled at 100,000 records, produced very strong performance with an accuracy of 0.9991, recall of 0.8410, precision of 0.9640, F1 score of 0.8983, MCC of 0.9000, and AUC of 0.9552.

In contrast, the **synthetic test set**, which was also sampled with 100,000 records, achieved an accuracy of 0.9400, a recall of 0.9646, a precision of 0.7133, an F1 score of 0.8202, an MCC of 0.7980, and an AUC of 0.9808. These results show that while the **synthetic dataset** produced **higher recall and AUC**, it came at the cost of **much lower precision and MCC**. In practical terms, this means that the synthetic data suggested the model could **catch more fraud cases**, but with a far **higher rate of false alarms**. Such false positives could overwhelm investigation teams, increase operational cost, and negatively affect customer experience.

On the other hand, the **Kaggle test set** demonstrated a much **better balance between precision and recall**. Its **higher MCC** reflects strong overall classification quality across both classes, while its precision of 0.964 indicates **very few false alarms**—an essential property in financial fraud detection. Although recall was somewhat lower than on synthetic data, the trade-off is acceptable because fewer genuine frauds are missed without significantly compromising efficiency. Furthermore, because the **Kaggle test set** represents the **actual data distribution**, it offers a **more trustworthy estimate** of model performance in **real-world deployment**. The **synthetic test set**, while useful, can **introduce optimistic bias** by smoothing distributions or reducing noisy, borderline cases, which inflates metrics such as AUC and recall but may not hold under production conditions.

Nevertheless, an important **limitation** must be acknowledged: the Kaggle test set does **not come from real company or e-commerce transactions**, but from a publicly available benchmark dataset. While it serves as a strong proxy for real-world conditions, its transaction patterns, fraud rate, and feature distributions may not fully match what would be seen in an actual business environment. Therefore, the reported results should be **viewed with caution**, and **further testing on real company data** will be necessary to confirm the model's performance in practice. For future studies, the model should be validated on actual e-commerce transaction data, tested against different fraud patterns, and monitored over time to capture data drift and changing customer behaviour. This will help ensure the model remains accurate, reliable, and ready for real deployment in a business environment.

6.2 Model Evaluation on Kaggle Test Set

The **final model** selected for deployment is a **Random Forest model with oversampling (default settings)**. Its performance on the **split test set** (held-out portion of the original dataset) and the **Kaggle test set** (sampled 100,000 records for deployment) is summarized in *Table 6.2.1* below:

Metric	Internal Split Test Set	Kaggle Test Set
Accuracy	0.9999	0.9991
Recall	1.0000	0.8410
Precision	0.9998	0.9640
F1-Score	0.9999	0.8983
MCC	0.9998	0.9000
AUC	1.0000	0.9552

Table 6.2.1: Performance Comparison of Final Random Forest Model on Split Test Set and Kaggle Test Set

The metrics on the **split test set** indicate **near-perfect performance**, which is expected because this set was drawn from the same dataset used for training. The model has effectively learned patterns present in the original distribution.

The **Kaggle test set**, although drawn from the same overall dataset, shows **noticeably lower recall, F1-score, MCC, and AUC**. This difference highlights that even within the same dataset, variations in sampling, feature distributions, and the presence of borderline or rare fraud cases can affect model performance.

The **lower recall (0.8410)** on the Kaggle test set indicates that the model misses more actual fraud cases compared to the split test set, suggesting that this subset contains transactions that are harder to classify, such as subtle or borderline fraud patterns. Correspondingly, the **drop in F1-score and MCC** reflects the challenge of balancing fraud detection with minimizing false positives; the Kaggle set is more difficult, providing a more realistic picture of operational performance. Additionally, the **slightly lower AUC (0.9552)** shows that the model's ability to rank transactions by fraud likelihood is somewhat weaker on truly unseen examples, emphasizing the importance of evaluating the model on an independent dataset before deployment.

The comparison shows that while the model performs exceptionally on the split test set, **the Kaggle test set provides more practical insights into real-world performance**. Its lower metrics reveal challenging transaction cases and emphasize that deployment must consider such variability. This analysis highlights the importance of evaluating the model on a representative, independent dataset before operational use.

6.3 Dashboard Evaluation

The evaluation of the developed Power BI dashboard was conducted to assess its functionality, usability, and analytical effectiveness. The process considered both technical aspects—such as data accuracy, visual correctness, interactivity, and performance—as well as user-focused aspects, including usability, clarity, and satisfaction. This dual perspective ensures that the dashboard not only functions correctly from a system perspective but also provides a positive experience for end-users. To present the findings clearly, the evaluation is divided into two parts: **technical evaluation** and **user acceptance evaluation**.

6.3.1 Technical Evaluation

Component	Objective	Procedures and Expected Outcomes	Actual Result	Pass/Fail
Data Accuracy	Verify correct data source link and metrics	Confirm dashboard is connected to OneDrive; compare dashboard totals vs raw dataset. Dashboard should match OneDrive data and reflect updates after refresh.	Dashboard successfully connected to OneDrive and show no error; total transactions = 100,000, fraud cases = 478, matching the raw dataset totals; metrics update correctly after refresh	Pass
Visual Accuracy	Ensure charts and tables represent data correctly	Inspect line, bar, and pie charts. Labels, axes, legends should be correct. Charts should reflect accurate trends, proportions, and values.	All charts accurately displayed trends and proportions. Axes, labels, and legends were correct.	Pass
Slicer Functionality	Confirm slicers interact correctly with visuals	On **Time Analysis** page, apply different slicer (category, fraud label) sequentially. All visuals should update dynamically without conflicts.	Slicer selections updated all visuals correctly. Multiple slicers worked together without issues (see *Figure 6.3.1–6.3.3*).	Pass
Conditional Formatting	Validate performance card colour coding	On **Model Performance Page**, check KPI cards. Metrics ≥ 90% should display **green**, 75–89% **yellow**, <75% **red**.	Colours applied correctly for all metrics (see *Figure 6.3.4*).	Pass
Conditional Formatting	Validate fraud/wrong prediction highlighting	On **Credit Card Transactions Page**, check transaction table. Fraud cases and wrong predictions should display in **red**.	Highlighting worked as expected — only fraud cases were highlighted, and since no wrong predictions existed, none were marked (see *Figure 6.3.5*).	Pass

Drill-through	Ensure navigation to detailed view works	On **Credit Card Transactions Page**, click a transaction in the Transaction History table. Dashboard should open **Transaction Details Page** with matching metrics, card info, and demographics.	Drill-through worked correctly. Details matched the selected record, and Transaction ID was consistent with the selected row (see *Figure 6.3.6–6.3.7*).	Pass
Performance	Test load and refresh speed	Measure time to open PBIX file and refresh dataset of 100k records. File should open <1 min, refresh <2 min.	Dashboard PBIX file opened in 30s; refresh completed in 1 min.	Pass

Table 6.3.1: Technical Evaluation Test Cases and Results

Figure 6.3.1: Time Analysis Page with Category Slicer Not Applied

Figure 6.3.2: Time Analysis Page with Category Slicer Applied

Figure 6.3.3: Time Analysis Page with Category and Fraud Label Slicers Applied

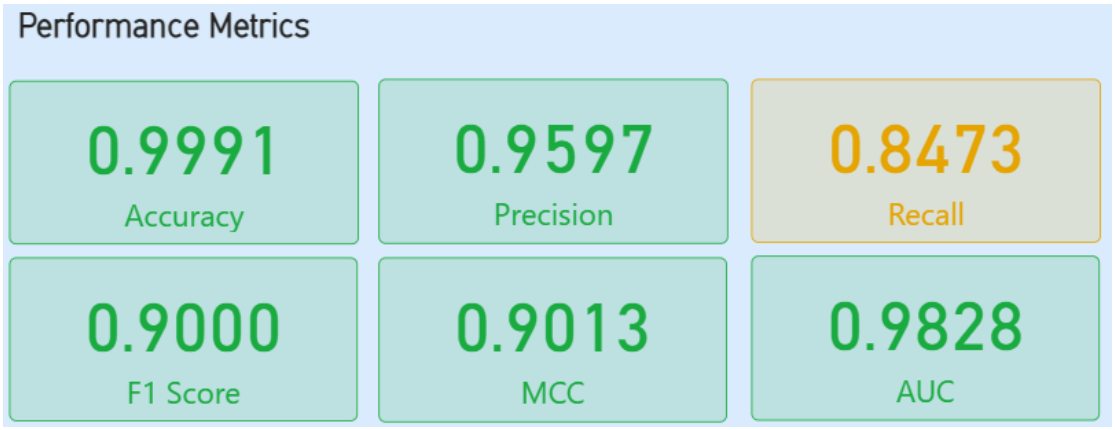


Figure 6.3.4: KPI Cards on Model Performance Page with Colour Coding Applied

Transaction ID	Transaction Date & Time	Predicted	Actual	Prediction Result
2c4663a8f95af72abf3d106876a3ac4f	11/8/2020 11:43:00 PM	Non-Fraud	Non-Fraud	Correct
34fe2b80f76aca88e3a890f3e5799cf0	27/10/2020 8:29:00 PM	Fraud	Fraud	Correct
38eed4db80f63261cb6b5144da18eb1f	2/7/2020 2:32:00 PM	Non-Fraud	Non-Fraud	Correct
39ffc5b7d01d5e6cb76333c4c1764a49	29/10/2020 4:52:00 AM	Non-Fraud	Non-Fraud	Correct
44fcd839d523c96336d89b319e5dcf4f	30/7/2020 11:38:00 AM	Non-Fraud	Non-Fraud	Correct
479c5378269750cedaf11135ae8406ab	23/8/2020 4:02:00 AM	Non-Fraud	Non-Fraud	Correct
4a8025cda65535e6b13aebb2671e9596	11/8/2020 2:35:00 AM	Non-Fraud	Non-Fraud	Correct
4e03ae076be457f47fb0ca471258c637	28/12/2020 8:30:00 AM	Non-Fraud	Non-Fraud	Correct
502efb313d5a3263ab0094e7fb5a3959	29/9/2020 8:09:00 PM	Non-Fraud	Non-Fraud	Correct
534bf5f53bba506801237c87e6f33bc5	5/12/2020 8:41:00 AM	Non-Fraud	Non-Fraud	Correct
5596bdd96ed1a23ea7fc1a2326491219	25/8/2020 4:44:00 AM	Non-Fraud	Non-Fraud	Correct
59fdaaf4af02ae2614009a3e0dbd8f88	7/9/2020 10:01:00 AM	Non-Fraud	Non-Fraud	Correct
685db86ce0a2e8fe864dbf40450c9181	27/10/2020 11:54:00 PM	Fraud	Fraud	Correct

Figure 6.3.5: Transaction Table on Credit Card Transactions Page Showing Fraud Case Highlighting

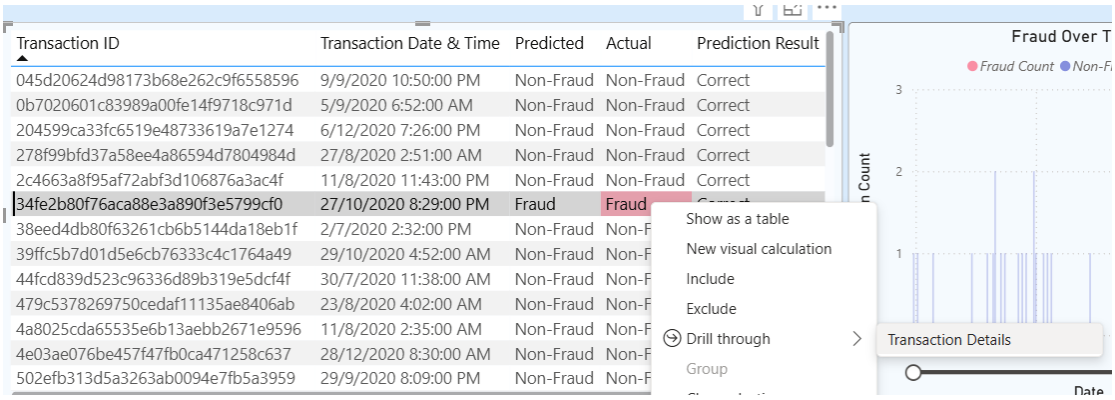


Figure 6.3.6: Selected Transaction in Credit Card Transactions Page Prior to Drill-through

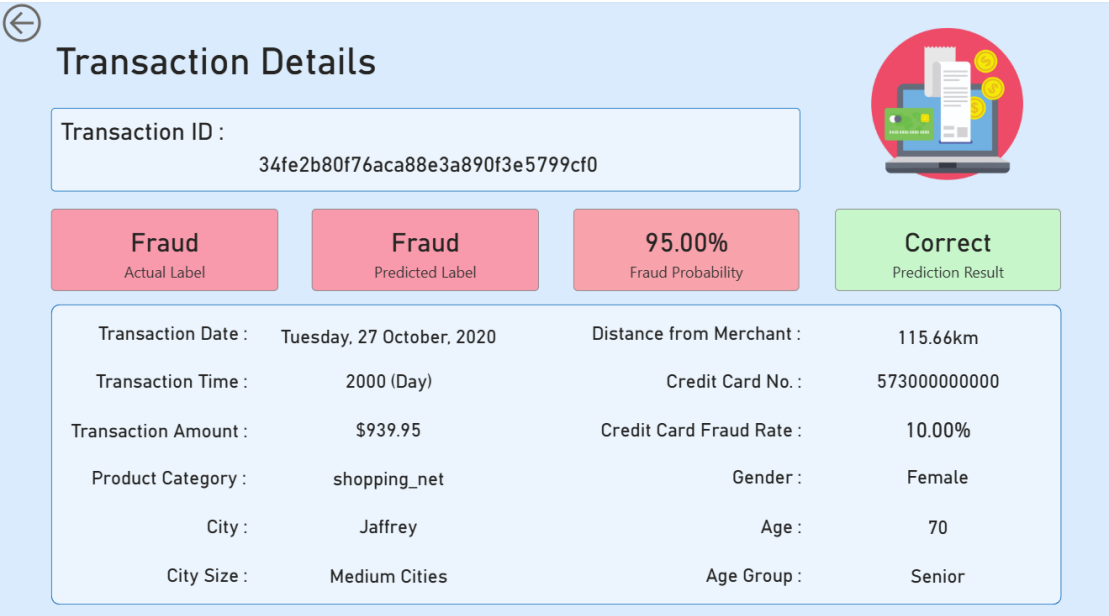


Figure 6.3.7: Transaction Details Page Showing Matching Transaction ID

The technical evaluation confirmed that the dashboard performed reliably across all tested components. Data accuracy was validated by matching dashboard metrics with the raw dataset, while visual accuracy checks ensured charts, labels, and legends were displayed correctly. Interactive features such as slicers, conditional formatting, and drill-through functionality all operated without error, providing smooth navigation and clear visual feedback. Finally, performance testing showed that the dashboard opened and refreshed efficiently within the expected time limits. Overall, all test cases passed successfully, demonstrating that the dashboard meets the required technical standards for accuracy, usability, and performance.

6.3.2 User Acceptance Evaluation (SUS Questionnaire)

To complement the technical evaluation, user acceptance testing was conducted using the **System Usability Scale (SUS) questionnaire**. 10 standardised questions were adapted to the context of the fraud detection dashboard, covering usability, clarity, responsiveness, and interactivity. The survey was created using Google Forms and distributed to respondents, who rated each statement on a five-point Likert scale (1 = Strongly Disagree to 5 = Strongly Agree). This approach provides a structured and quantifiable evaluation of the dashboard's usability and user satisfaction.

A total of **15 respondents** participated in the evaluation. The feedback gathered from the respondents was used to calculate the SUS score and provide insights into the usability, clarity, responsiveness, and interactivity of the dashboard. To provide a clearer interpretation of the results, the SUS statements were grouped into five themes: ease of use, perceived complexity, visual clarity, responsiveness and interactivity, and overall satisfaction.

The **ease-of-use** theme (**Q1, Q3, Q7**), which contained positively worded items, received high ratings, with the majority of respondents selecting 4 or 5, as shown in *Figure 6.3.8-6.3.10*. Only one respondent gave a neutral score of 3 for navigation (Q3), as shown in *Figure 6.3.9*. This indicates that the dashboard is intuitive, requires minimal learning effort, and provides clear navigation paths. Overall, the results suggest that users can quickly adapt to the system without prior training.

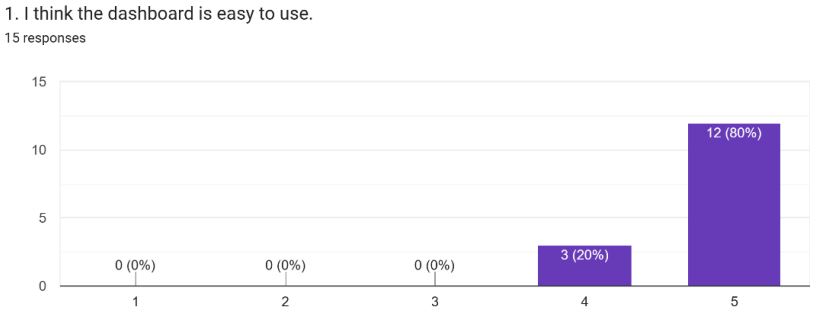


Figure 6.3.8: Ease of Use Evaluation

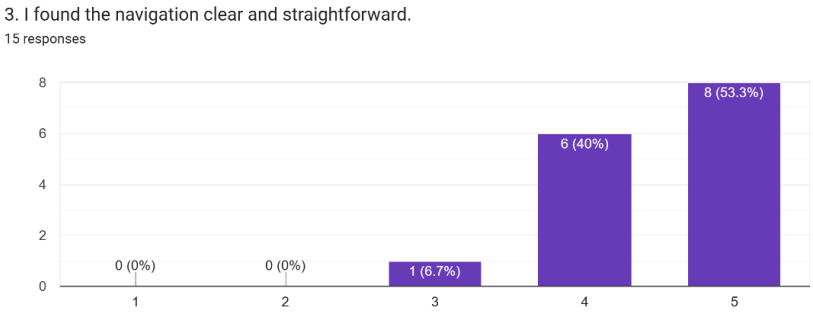


Figure 6.3.9: Navigation Clarity Evaluation

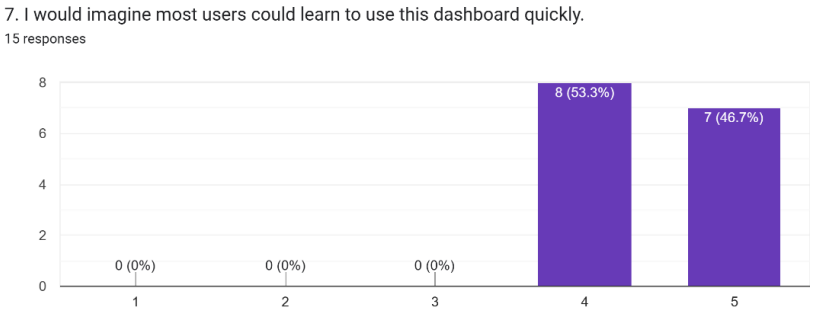


Figure 6.3.10: Ease of Learning Evaluation

In contrast, the **perceived complexity** theme (**Q2, Q4, Q6**), which contained negatively worded items, scored very low, with most respondents choosing 1 or 2, as shown in *Figure 6.3.11-6.3.13*. Notably, all respondents rated Q6 with a 1, indicating strong agreement that the dashboard was neither inconsistent nor confusing. These results suggest that users did not perceive the dashboard as unnecessarily complex or requiring technical support.

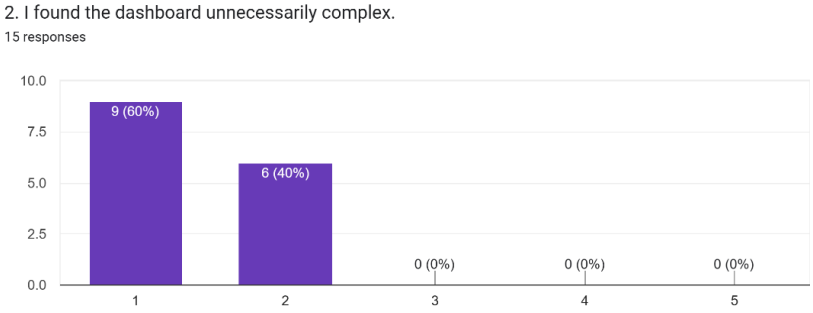


Figure 6.3.11: Perceived Complexity Evaluation

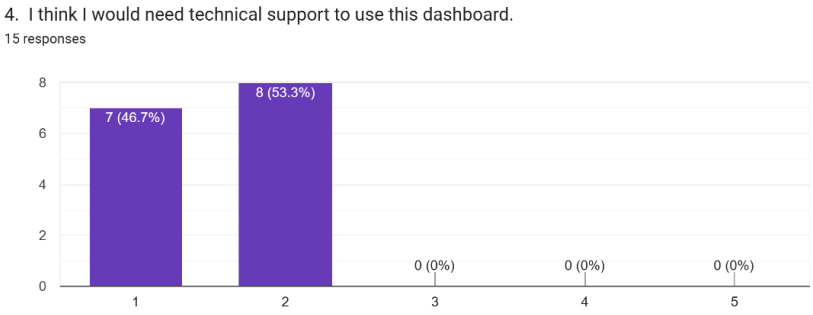


Figure 6.3.12: Need for Support Evaluation

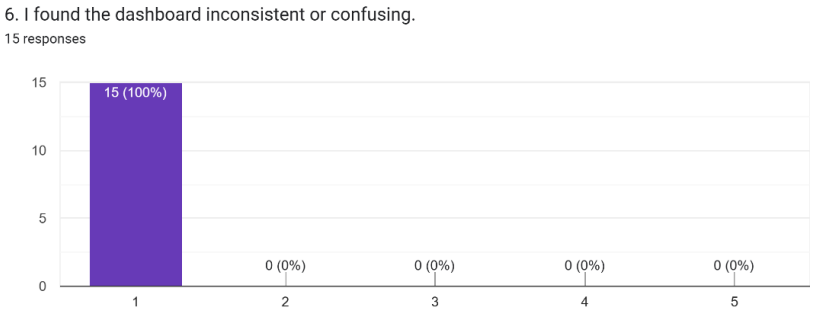


Figure 6.3.13: Consistency Evaluation

The **clarity of visuals** (Q5) was rated particularly highly, with almost all respondents strongly agreeing that charts, tables, and numbers were easy to understand, as shown in *Figure 6.3.14*.

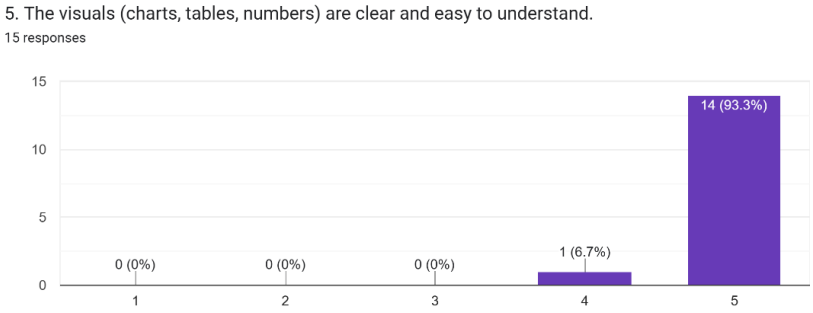


Figure 6.3.14: Visual Clarity Evaluation

For the **responsiveness and interactivity** theme (Q8, Q9), *Figures 6.3.15-–6.3.17* illustrate that nearly all respondents selected 4 or 5, indicating that the dashboard responds quickly to filters and slicers and that interactive features work as expected. This demonstrates that the system provides a smooth, responsive, and reliable interaction experience.

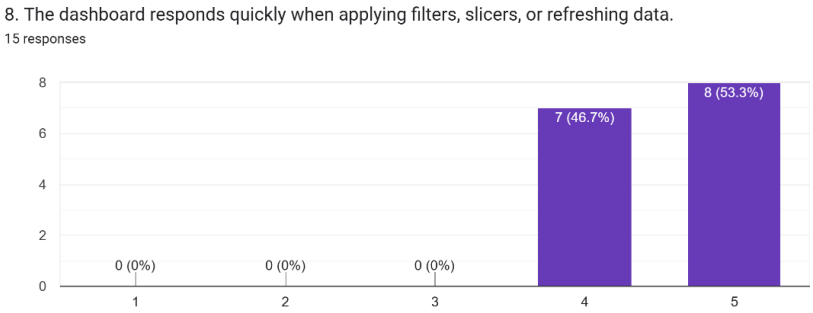


Figure 6.3.15: Responsiveness Evaluation

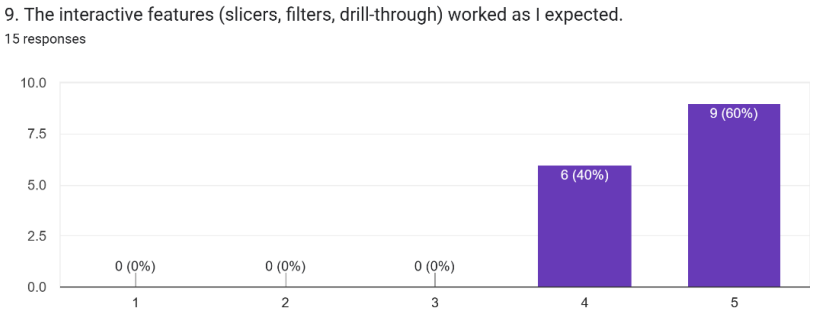


Figure 6.3.16: Interactivity Evaluation

Finally, as seen in *Figure 6.3.17*, all respondents rated satisfaction highly (4 or 5), showing strong acceptance and confidence in the dashboard. This confirms that the system meets user expectations and achieves a high level of usability.

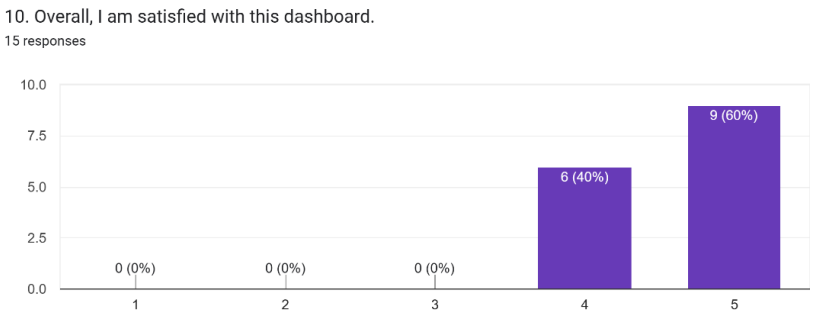


Figure 6.3.17: Overall Satisfaction

After analysing the individual questions, the overall **System Usability Scale (SUS) scores** were calculated to provide a single benchmark of usability. The SUS scoring followed the standard procedure: for **positively worded items** (Q1, Q3, Q5, Q7, Q8, Q9, Q10), the adjusted score was calculated as ***Response − 1***, while for **negatively worded items** (Q2, Q4, Q6), the adjusted score was calculated as ***5 − Response***. This converted all responses into a range from 0 to 4. The adjusted scores were then summed across all 10 statements to obtain a total score between 0 and 40, which was subsequently multiplied by 2.5 to produce the final SUS score, ranging from 0 to 100.

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Total (0-40)	SUS (0-100)
5	2	5	2	5	1	5	4	4	5	36	90
4	1	4	1	5	1	5	5	5	5	38	95
5	1	4	2	4	1	4	4	5	4	34	85
5	1	5	1	5	1	5	5	5	5	40	100
5	1	5	2	5	1	4	5	4	4	36	90
5	2	5	2	5	1	5	5	4	5	37	92.5
4	2	4	2	5	1	4	4	4	4	32	80
5	1	4	1	5	1	5	4	5	5	38	95
4	2	5	2	5	1	4	5	4	4	34	85
5	2	4	1	5	1	4	4	5	5	36	90
5	2	3	2	5	1	4	5	4	4	33	82.5
5	1	5	1	5	1	5	5	5	5	40	100
5	1	4	1	5	1	5	5	5	5	39	97.5
5	1	5	2	5	1	4	4	5	5	37	92.5
5	1	5	1	5	1	4	4	5	4	37	92.5
Average										**36.47**	**91.17**

Table 6.3.2: System Usability Scale (SUS) Evaluation Results

Table 6.3.2 presents the SUS scoring results for all 15 respondents. The scores ranged from **80 to 100**, with an overall average of **91.17**, which is above the commonly accepted usability benchmark of 70. This indicates that the dashboard achieved a high level of usability and user satisfaction, meeting the success criteria set for the evaluation.

When considered alongside the technical evaluation, these results confirm that the dashboard is not only functionally accurate and reliable but also user-friendly and well-accepted by end users.

6.4 Insights from Dashboard Results

The dashboard not only serves as a visualization tool but also provides valuable insights into fraud behaviour across temporal, spatial, demographic, and behavioural dimensions. Using the **Kaggle credit card fraud test set** as the underlying data source, the analysis of outputs across different dashboard pages revealed several key patterns and risk factors.

Overview

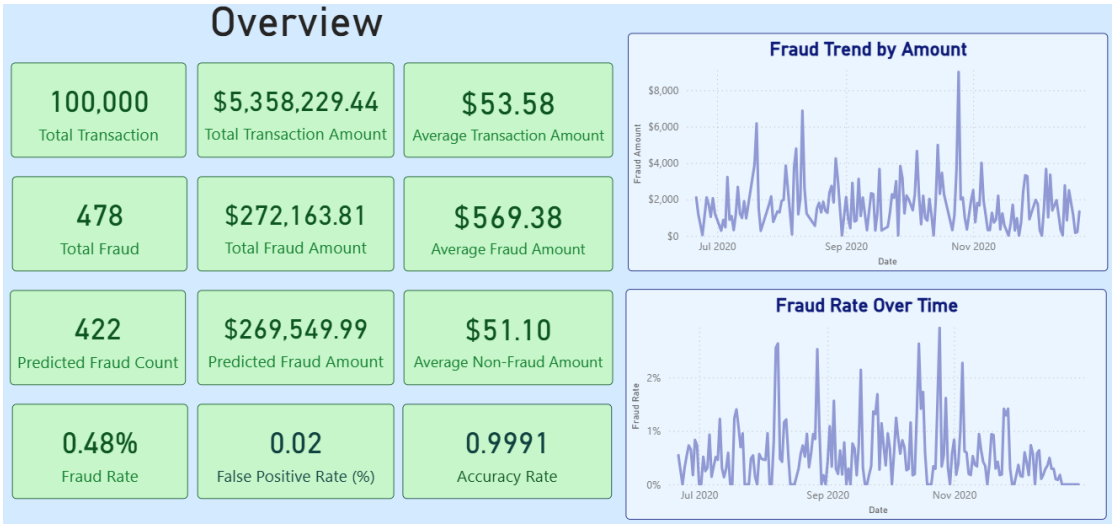


Figure 6.4.1: Overview Page

The Overview Page provides a clear summary of the dataset, fraud patterns, and model performance. It covered 100,000 transactions with a total value of $5.36 million, averaging $53.58 per transaction. Although fraud cases made up only 0.48% of all transactions, their financial impact was significant, with a total fraud amount of $272,163.81 and an average of $569.38 per fraudulent transaction—more than ten times higher than the average non-fraud amount of $51.10. The model successfully identified 422 fraud cases with very high accuracy (99.91%) while maintaining a very low false positive rate (0.02%), indicating its strong ability to distinguish between legitimate and fraudulent activity.

The trend charts provide further insight, as both fraud amount and fraud rate show irregular spikes at certain periods, suggesting that fraud occurs in sudden increases rather than consistently over time. This opportunistic behaviour highlights the importance of continuous monitoring and timely detection, since fraud cases are not evenly distributed but concentrated in specific high-risk periods.

Time Analysis

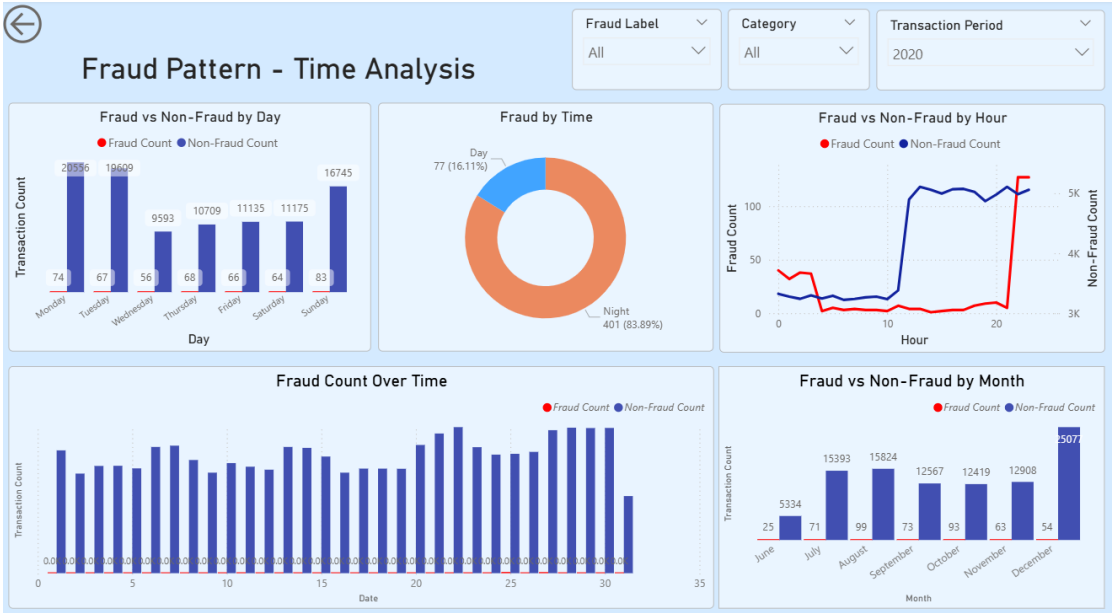


Figure 6.4.2: Time Analysis Page

The first chart, **Fraud vs Non-Fraud by Day**, compares daily transaction amounts between fraud and normal cases. From the chart, it can be seen that fraud does not always rise in line with total transactions. It is evident that fraud is disproportionately higher during weekends, particularly on Sundays, which recorded the highest fraud count despite not having the peak transaction volume. Mondays and Saturdays also show elevated levels of fraudulent activity, while mid-weekdays such as Tuesday and Wednesday display comparatively lower fraud counts. This pattern suggests that fraudsters may take advantage of weekends when both customer vigilance and institutional monitoring are potentially weaker.

The **Fraud by Time** chart shows that over 80% of fraud takes place at night, suggesting that fraudsters prefer to operate when monitoring is weaker, such as late at night or early in the morning. This observation is further supported by the **Fraud vs Non-Fraud by Hour** chart, which compares fraudulent and legitimate transactions across the day. It highlights that fraud risk starts to rise around 10 PM and remains elevated until about 3 AM. Although the overall number of transactions is lower during these hours, the proportion of fraud is significantly higher than during the day, making late-night hours the most vulnerable period for fraudulent activity.

The **Fraud vs Non-Fraud Count Over Time** chart shows that while non-fraud transactions remain stable across the days, fraud cases appear in small, scattered amounts. The variation does not indicate a strong pattern but rather suggests random or opportunistic fraud attempts. This reinforces the need for continuous monitoring, as fraud can occur unpredictably at any time, even when transaction volumes appear normal.

The last chart, **Fraud vs Non-Fraud by Month**, shows longer-term patterns. The monthly view shows that while December records the highest number of transactions overall, the fraud count during this period is relatively low. This suggests that higher transaction volume does not necessarily lead to higher fraud cases. In contrast, months with fewer total transactions, such as August and October, show higher fraud counts, indicating that fraud patterns are not strictly tied to transaction activity levels. Instead, fraud appears to occur more steadily across months, without a direct seasonal spike.

Geography

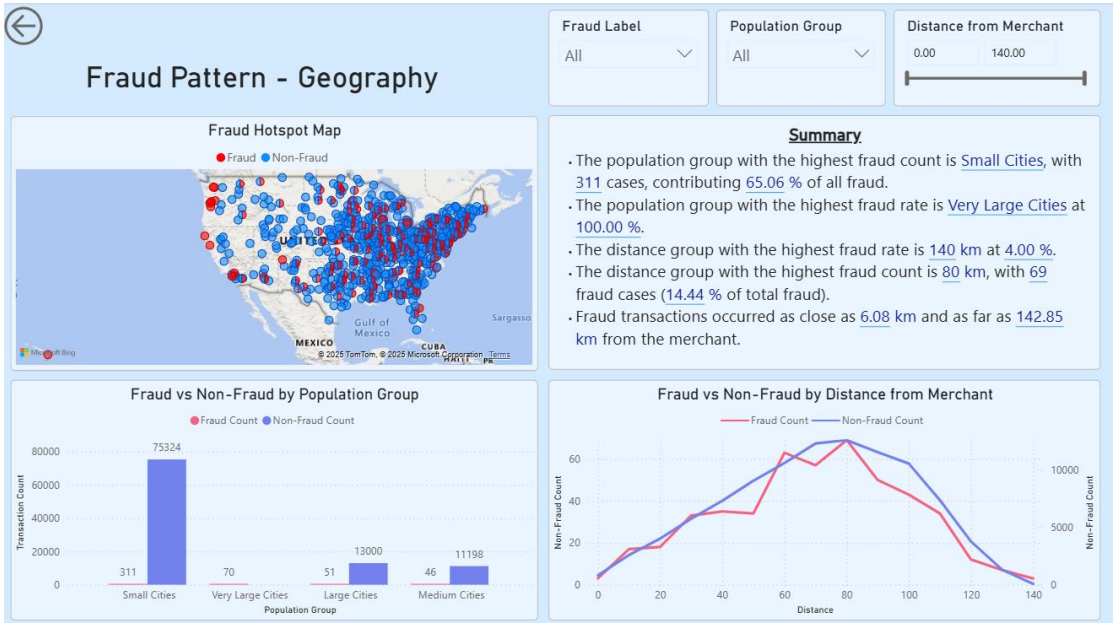


Figure 6.4.3: Geography Page

The **Fraud Hotspot Map** visualizes the distribution of fraudulent and non-fraudulent transactions across the United States. Each dot represents a transaction location, with red indicating fraud and blue indicating non-fraud, while a half-red, half-blue marker signifies the presence of both types at the same location. This visualization makes it easier to identify

geographical clusters and potential hotspots. For instance, *Figure 6.4.4* illustrates that when the cursor hovers over an isolated point, the tooltip displays detailed information such as the city name Honokaa, the fraud label indicating it as a fraudulent case, the latitude and longitude coordinates, a transaction count of four, the corresponding merchant coordinates, and a recorded distance of 84.16 km from the merchant. The results indicate that fraud is not evenly distributed: while most transactions are concentrated in the eastern and central United States, fraud hotspots appear scattered within these clusters, suggesting that fraudulent activity tends to emerge in particular areas rather than being uniformly spread.

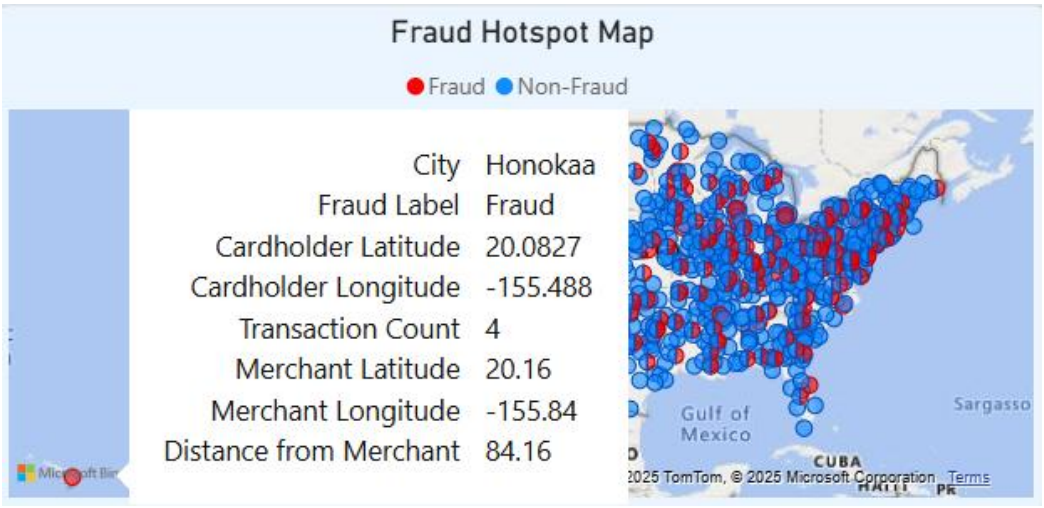


Figure 6.4.4: Tooltip Information from Honokaa Transaction Point

The **Fraud vs Non-Fraud by Population Group** bar chart compares transaction counts across city sizes. Small Cities record the highest transaction volumes overall, making them the biggest source of fraud in absolute terms. However, fraud cases are also observed in Large Cities, Medium Cities, and Very Large Cities, though in smaller numbers. This indicates that fraud is not exclusive to one type of city—while small cities dominate due to volume, larger urban areas are not immune to fraud risk.

The **Fraud vs Non-Fraud by Distance from Merchant** line chart plots fraud and non-fraud counts against distance. Both fraud and non-fraud transactions increase together as distance grows, peaking around **80 km**, before declining again. The parallel trend suggests that fraudsters often mimic normal transaction distance patterns to avoid detection, but the elevated fraud share at 80 km shows a hotspot where risk is disproportionately high. Beyond 100 km, both fraud and non-fraud counts drop sharply, showing that long-distance transactions are less common overall.

Demographics

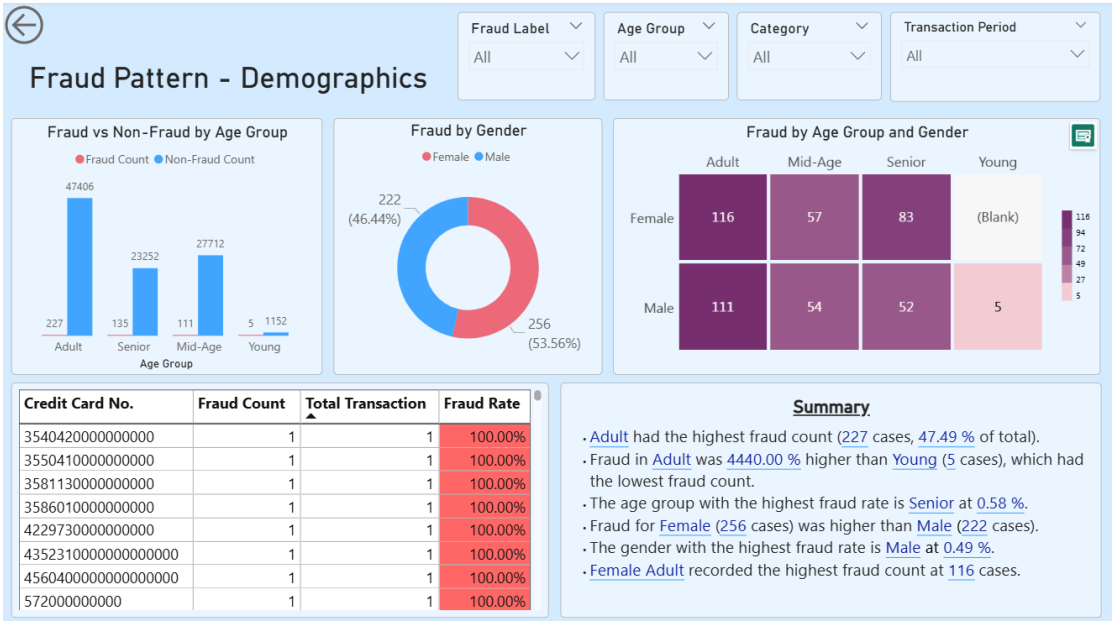


Figure 6.4.5: Demographics Page

The **Fraud vs Non-Fraud by Age Group** chart compares the volume of fraudulent and legitimate transactions across different age categories. Fraud is represented in red, while non-fraudulent transactions are displayed in blue. From this chart, it is evident that the **adult age group records the highest number of fraud cases**, while the **young age group shows the least**. This suggests that adults, due to higher transaction activity, face greater exposure to fraud compared to younger or mid-age groups.

Fraud by Gender chart provides a clear breakdown of fraud cases between male and female cardholders. While the difference is not extreme, the chart shows that **female cardholders record slightly more fraud cases than male cardholders**. This finding suggests minor variations in fraud exposure or transaction behaviour between genders.

The **Fraud by Age Group and Gender** heatmap combines the two demographics to reveal intersections. Darker shades indicate higher fraud counts. The heatmap shows that **adult females represent the group with the highest fraud count**, followed by adult males. Meanwhile, younger groups records minimal fraud activity regardless of gender. This combined view provides deeper insight, showing not only which groups are most affected individually, but also how fraud concentrates at the intersection of gender and age.

The **Summary Box** highlights the age and gender groups with the highest and lowest fraud counts and rates, as well as the age–gender combination most affected, giving a quick view of key demographic risks.

The **Credit Card Fraud table** lists all cards associated with fraudulent transactions. It displays the card number, fraud count, total transactions, and fraud rate. The fraud rate column is highlighted in red, where darker shades indicate higher rates, with some cases reaching 100%. This table provides investigators with detailed visibility into which specific credit cards are consistently linked to fraudulent activity. In addition, it supports drill-through functionality, allowing users to right-click on a card number and navigate to the Credit Card Transaction page for further analysis. For instance, **drilling through on card number *'573000000000'*** provides a detailed view of its associated transactions as shown in *Figure 6.4.6*.

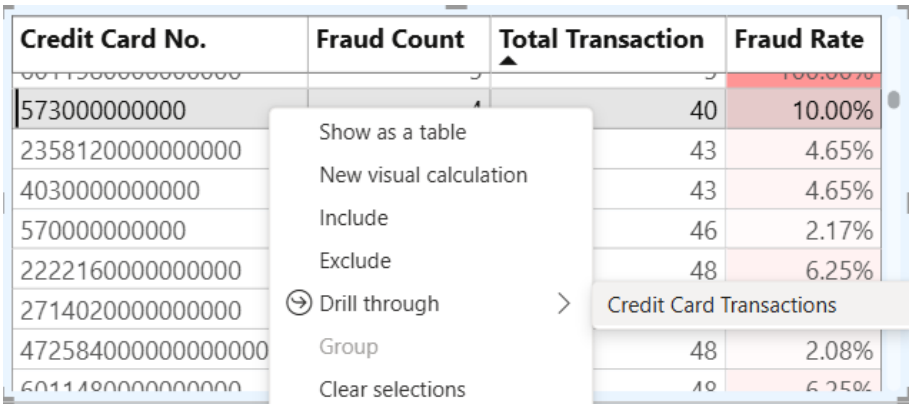


Figure 6.4.6: Example of Drill-Through Navigation from Credit Card Fraud Table

Credit Card Transactions

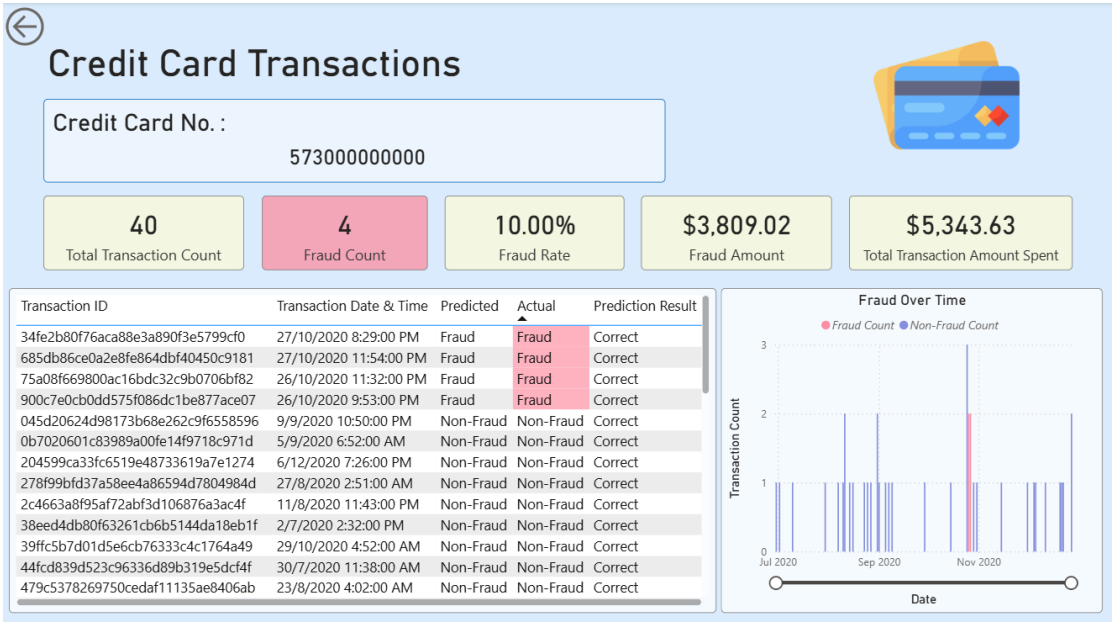


Figure 6.4.7: Credit Card Transactions Page

Drilling through to the transaction-level view of credit card number ***'573000000000'*** provides deeper insights into the fraud patterns associated with this account. Overall, the card has **40 recorded transactions**, out of which **4 were confirmed as fraudulent**, resulting in a **fraud rate of 10%**. This is a significant proportion, given the financial impact: fraudulent transactions alone amounted to **$3,809.02**, which represents more than **70% of the total amount spent ($5,343.63)**. This highlights that while fraudulent activity was limited in count, it carried a disproportionately large monetary impact compared to legitimate transactions.

The **Transaction-level details table** further confirm the accuracy of the fraud detection model. Each of the four fraudulent transactions was correctly identified and flagged, reflecting strong predictive performance. These fraud cases occurred primarily during **late-night hours** (e.g., 11:32 PM and 11:54 PM), which aligns with broader time-analysis findings that fraudsters exploit reduced monitoring during night periods. In contrast, non-fraudulent transactions were distributed more evenly across different times of the day, suggesting a clear behavioural distinction between fraudulent and legitimate usage.

The **Fraud Over Time** chart shows transaction activity between July and November 2020. Most of the card's transactions were legitimate and spread evenly across the months, but the fraudulent ones appeared in small clusters, especially in **late October**. These spikes are

important because they suggest planned attempts at fraud rather than random events. Investigators can use this trend to check if fraud matches seasonal patterns, busy shopping periods, or possible leaks of cardholder information.

Overall, the drill-through analysis shows that even though only 10% of this card's transactions were fraudulent, they caused most of the financial loss. The fraud cases mainly happened late at night and were grouped into certain months, pointing to deliberate and opportunistic misuse. For investigators, this page provides useful evidence to strengthen fraud controls on the account, such as adding extra checks for late-night transactions, reviewing high-value purchases more closely, and monitoring for repeated bursts of suspicious activity.

By right-clicking on a transaction, users can navigate to the Transaction Details page for deeper analysis. For example, drilling through on the first transaction reveals its full details as shown in *Figure 6.4.8*.

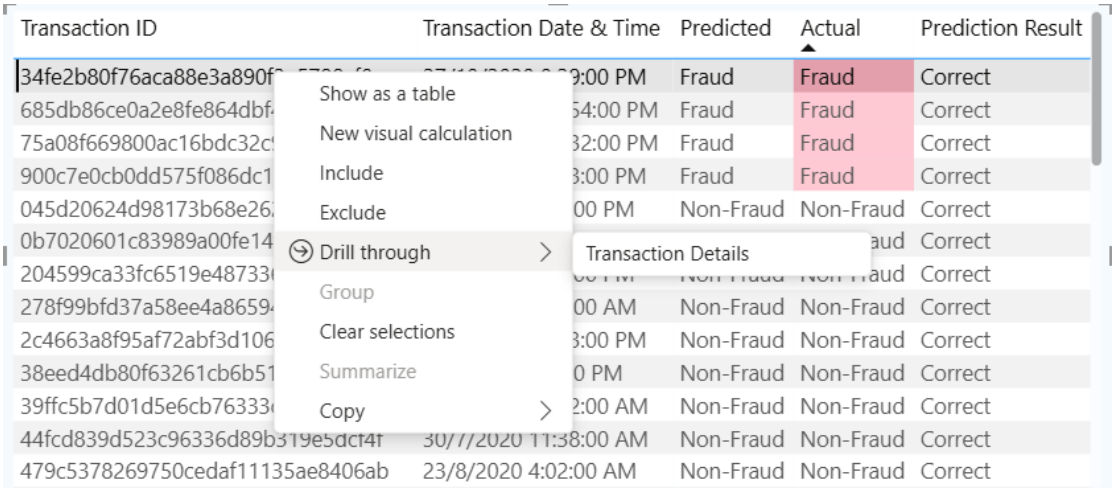


Figure 6.4.8: Example of Drill-Through Navigation from Transaction-level Details Table

Transaction Details

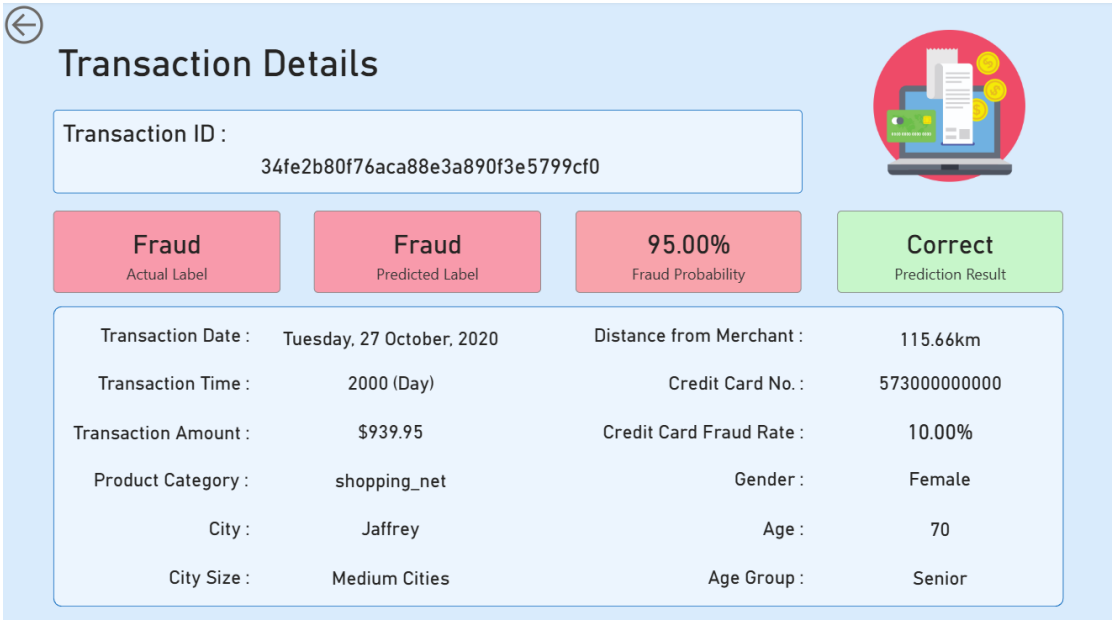


Figure 6.4.9: Transaction Details Page

The transaction details page provides a deeper look into one of the fraudulent transactions linked to this credit card account. In this case, transaction ID *'34fe2b80f76aca88e3a890f3e5799cf0'* was **correctly identified** as fraud with **a high confidence score of 95%**, confirming the accuracy of the detection model. The transaction amount of $939.95 is relatively large, consistent with the broader pattern where fraudulent activity on this card tends to involve disproportionately high-value purchases. The payment was made under the **shopping_net** category, indicating an online channel that is generally more susceptible to misuse compared to in-person transactions.

Additional context highlights that the cardholder is a 70-year-old female from Jaffrey, categorized as a **senior** in a **medium-sized city**—a demographic that can be more vulnerable to fraud attempts. Furthermore, the transaction occurred **115.66 km away from the merchant** location, suggesting a possible geographic inconsistency that may raise suspicion. Although many fraud cases on this card occurred late at night, this one happened earlier in the evening (8 PM), showing that fraud is **not limited to specific hours**. Overall, the case reinforces how high value and online transactions are strong indicators of fraudulent behaviour.

Behavioral Analysis

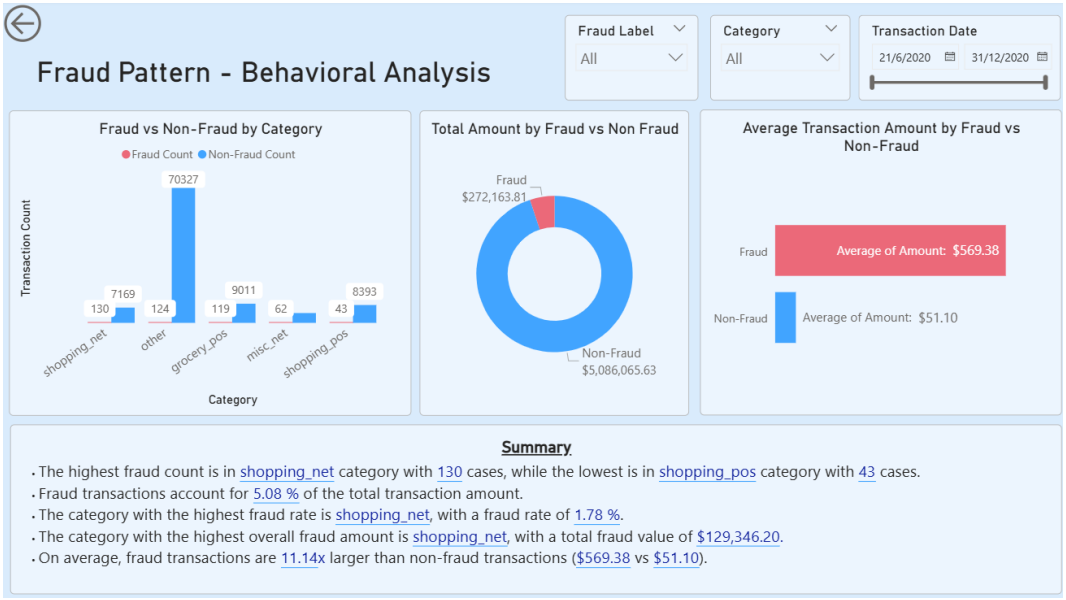


Figure 6.4.10: Behavioral Analysis Page

The first visualization, **Fraud vs Non-Fraud by Category chart**, compares the number of fraudulent and non-fraudulent transactions across various categories such as shopping_net, grocery_pos, misc_net, shopping_pos, and others. Fraudulent transactions are represented in red, while non-fraudulent transactions appear in blue. From the chart, while categories like shopping_net and grocery_pos record the highest transaction counts, fraud is disproportionately concentrated in shopping_net, which registers the highest fraud cases. This indicates that fraudsters **prefer online shopping platforms** over point-of-sale transactions, likely due to weaker verification measures and the ease of executing remote purchases.

The **Total Amount by Fraud vs Non-Fraud** chart shows that although fraudulent transactions make up only a small fraction of the overall count, they account for a disproportionately large share of the total value, reaching $272,163.81 compared to $5,086,065.63 for non-fraudulent transactions. This highlights that fraud is often concentrated in higher-value cases rather than in volume. The **Average Transaction Amount by Fraud vs Non-Fraud** chart further supports this pattern, showing that the average fraudulent transaction is $569.38, which is more than eleven times higher than the $51.10 average for legitimate transactions. Together, these findings suggest that fraudsters deliberately **target high-value purchases** to maximize returns, underscoring the importance of applying stricter monitoring and risk controls to larger transactions.

The **Summary box** provides a concise overview of fraud trends. It shows that the highest fraud activity occurs in the online shopping category, while point-of-sale transactions record the lowest. Online channels also have the highest fraud rate and overall fraud amount, indicating that they are the most vulnerable environment and represent the greatest financial exposure.

Model Performance

Figure 6.4.11: Model performance Page

Confusion Matrix heatmap shows that the majority of predictions fall correctly into their categories, with **405 true positives** and **99,505 true negatives**, while only **17 false positives** and **73 false negatives** appear. This demonstrates excellent predictive strength, but the presence of missed fraud cases (false negatives) highlights a key risk area, since undetected fraud can cause significant financial losses.

The **Performance Metrics cards** give a concise yet powerful snapshot of model quality across six dimensions. Accuracy (0.9991), precision (0.9597), F1-score (0.9000), MCC (0.9013), and AUC (0.9828) are all in green, reflecting excellent results. Recall (0.8473), however, is highlighted in yellow, signalling a relative weakness. This indicates that while the model is highly precise in detecting fraud, it sometimes misses fraudulent cases.

The **Prediction Result slicer** is set by default to display **false negatives** and **false positives**, since these errors are most critical for fraud detection. The **Predicted Results by category chart** breaks down misclassifications into false negatives and false positives across transaction categories. The *"other"* category stands out, with **73 missed frauds** and **17 false alarms**. This suggests that certain types of transactions, possibly due to their diverse or irregular patterns, present more challenges for the model.

Similarly, the **Predicted results by age group chart** shows that most errors occur in the adult group, with **42 missed frauds** and **10 false positives**, while seniors and mid-age groups show moderate levels of misclassification, and young users experience very few errors. These insights reveal where the model struggles most, allowing targeted refinements.

Prediction confidence & Key Influencers

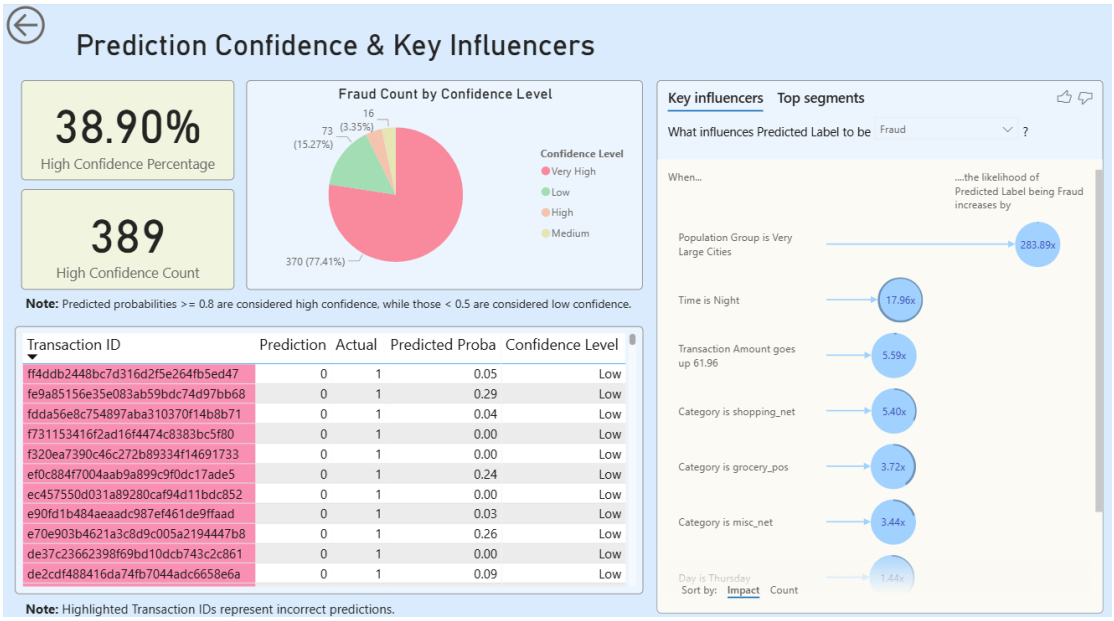


Figure 6.4.12: Prediction confidence & Key Influencers Page

The **prediction confidence section**, located on the left side of the page, begins with two KPI cards that summarize overall certainty. The first card shows that **38.90%** of predictions fall into the **high-confidence** range, while the second indicates this corresponds to **389 transactions**. High confidence is defined as cases where the model's **predicted probability is at least 0.8**, meaning the model is strongly confident in its decision for fraud cases. This gives

stakeholders a clear benchmark for trust in the model's outputs, while also showing how often the model produces highly reliable predictions.

Beside these cards, a **Fraud Count by Confidence Level** pie chart breaks down predictions into Very High, High, Medium, and Low categories. The chart reveals that most cases fall into the **Low-confidence range (77.41%)**, with far fewer in the Medium, High, and Very High ranges. In this model, a probability of **0.5 or higher** is classified as **fraud**, while anything **below 0.5** is classified as **non-fraud**. As a result, the large share of Low-confidence predictions (< 0.5) mostly represents legitimate transactions, which aligns with real-world conditions where genuine transactions far outnumber fraudulent ones. However, the key limitation is that some fraud cases also fall into this Low-confidence group, where the model assigns them a probability below 0.5 and misclassifies them as legitimate, showing the difficulty of detecting fraud that mimics normal behaviour.

The **transaction-level details table** provides a tabular breakdown of all individual predictions. Each row displays the Transaction ID, Predicted Label, Actual Label, Predicted Probability, and Confidence Level. Incorrect predictions are highlighted in red, making errors easy to spot. For example, some fraud cases were misclassified as non-fraud because they had very low probability scores, showing situations where the model lacked sufficient confidence to correctly flag them.

The **key influencers section**, on the right side of the page, explains **what drives the model to classify a transaction as fraud**. Using AI-driven analysis, the visual ranks the most important factors. The most significant driver is whether the **Population Group is in Very Large Cities**, which increases the likelihood of fraud by nearly **284 times**. Other strong influencers include whether the **transaction occurs at night (17.96x higher likelihood)**, whether the **amount exceeds 61.96 (5.59x higher)**, and whether the **category is shopping_net, grocery_pos, or misc_net**, all of which increase the likelihood by **3–5 times**. These findings not only validate the model's reasoning but also provide actionable insights for investigators, such as focusing additional scrutiny on urban, late-night, high-value online transactions.

6.5 Project Challenges

This project faced several important challenges that shaped the approach, methods, and results. One key issue was the **strong class imbalance**, as fraud made up less than 1% of all transactions. This created a risk of building a model that looked accurate but failed to detect fraud, since predicting only non-fraud would still give high accuracy. To address this, different resampling methods such as SMOTE, oversampling, and under-sampling were tested. These helped improve recall but also **brought trade-offs**, such as lower precision, higher computation time, and the risk of generating unrealistic patterns.

Another challenge was **ensuring the model could work well on unseen data** and avoiding **data leakage**. The very high performance seen on the internal test set did not carry over to the independent Kaggle test set, showing that the model was too optimistic when tested on familiar data. In addition, an early mistake in the pipeline—applying target encoding before splitting the data—caused leakage, which made the model look better than it really was. Fixing this required carefully rebuilding the preprocessing pipeline to give a fairer measure of real-world performance.

Finally, evaluation was limited by the **lack of strong benchmarks**. Unlike software areas with well-known standards, fraud detection dashboards have very few public examples for comparison. This made it hard to judge if the dashboard created was competitive or just functional. As a result, the evaluation focused on meeting project goals and user feedback, rather than direct comparison with industry-leading tools.

6.6 Objectives Evaluation

This project aimed to address key problems in fraud detection, including data imbalance, misclassification and evolving fraud patterns through the development of machine learning models and a Power BI dashboard. The extent to which each objective was achieved is discussed below.

Addressing data imbalance through resampling techniques

The extreme class imbalance, with fraud making up less than 1% of all transactions, was addressed using SMOTE, oversampling and under-sampling. Oversampling combined with Random Forest gave the most reliable results, producing a balanced trade-off between recall, precision, and F1-score. SMOTE also improved minority detection, while under-sampling performed poorly due to information loss. No resampling gave high accuracy but weak recall, F1, and MCC, showing that accuracy alone is misleading under imbalance. Prior research highlighted SMOTE's performance for fraud detection [11,12,18]. This project extends their findings by systematically comparing multiple resampling methods, proving oversampling to be the most effective. Although reevaluated results were not as strong as the near-perfect scores initially seen on the split test set, the model still achieved a solid F1-score of 0.9, which is considered satisfactory under highly imbalanced conditions. This successfully met the first sub-objective of enhancing fraud detection performance.

Reducing misclassification with ensemble models

To reduce costly misclassification errors, particularly false negatives, this project compared Random Forest, AdaBoost, and XGBoost. While all three models performed strongly on the internal split test, re-evaluation on the independent Kaggle test set confirmed Random Forest with oversampling is still reliable, achieving accuracy of 0.9991, precision of 0.96, recall of 0.85, and an F1-score of 0.90. This balance demonstrated its strength in minimizing false negatives while maintaining high precision. This directly addressed the second sub-objective of developing models that reduce misclassification.

Monitoring evolving fraud patterns through visualization

The evolving nature of fraud was addressed by deploying the chosen model within an interactive Power BI dashboard. The dashboard provides real-time monitoring of fraudulent activity and model performance through drill-through pages, conditional formatting, and interactive filters. This enables continuous tracking of fraud patterns, supporting timely updates when concept drift occurs. Thus, the third sub-objective of visualizing fraud detection performance and patterns was effectively achieved.

Integration of machine learning and Power BI for real-time monitoring

The main objective, integrating machine learning with a Power BI dashboard, was successfully realized. The exported Random Forest model and its preprocessing pipeline were embedded into the dashboard, resulting in a user-friendly system that combines predictive analytics with actionable visualization. This integration ensures the solution is not only technically sound but also practically valuable for fraud analysts.

6.7 Concluding Remark

This chapter provided a full review of the fraud detection system, covering both the machine learning model and the Power BI dashboard. The comparison of test sets showed that the real Kaggle dataset, while not perfect, is a better and more realistic way to measure performance than a synthetic one. This revealed the trade-off between recall and precision. Testing the final Random Forest model on this independent dataset proved it was strong, with a solid F1-score of 0.90, but also showed a drop in recall compared to the internal test set, reminding us why outside validation is important.

The dashboard was also carefully tested in two ways. The technical check confirmed that it worked correctly from data input to visualization. The user test, with a very high SUS score of 91.17, showed that people found it easy to use and helpful. The dashboard gave clear insights into fraud patterns over time, location, demographics, and behaviour, making the system not just predictive but also an investigation tool.

In short, this chapter shows that the project goals were achieved. The system handles class imbalance, reduces misclassification errors, and provides a strong platform for tracking fraud trends. By combining a reliable model with an easy-to-use dashboard, the project delivers a complete solution that is useful for both research and real-world fraud detection.

CHAPTER 7

Conclusion and Recommendation

7.1 Conclusion

The preliminary phase of this project established a strong foundation for building an effective fraud detection system in e-commerce. Key steps included thorough EDA, data preprocessing and handling class imbalance using resampling techniques. These efforts were essential in preparing the dataset for robust model training and evaluation.

Based on the analysis of model performance, Random Forest and XGBoost consistently outperformed AdaBoost in fraud detection. Among the resampling methods tested, both SMOTE and Oversampling significantly improved key metrics such as Recall, F1-score, and MCC, effectively addressing the challenges of class imbalance. Pipeline 2, which applied resampling before data splitting and then target encoding, was implemented during hyperparameter tuning. This pipeline was better to prevent target leakage and provided more reliable performance metrics that better reflect real-world deployment conditions.

Hyperparameter tuning further refined the models, but results indicated that the base models were already highly effective. Random Forest and XGBoost showed only marginal improvements, while AdaBoost benefited more obviously, though it still lagged behind the other two. Random Forest combined with Oversampling achieved near-perfect performance across multiple evaluation metrics, confirming its suitability as the final deployed model.

Robustness was validated through testing on both synthetic datasets and the Kaggle dataset, where the model demonstrated strong generalization and adaptability. Although the performance was not as high as on the internal split test set (which was near perfect), it still remained strong, ensuring reliability beyond internal testing and addressing concerns of overfitting and concept drift.

The integration of the final model into Power BI transformed predictive outcomes into a decision-support tool. The dashboard enabled monitoring of model performance and fraud patterns across dimensions such as time, geography, demographics, and behavioural attributes. Interactive features like slicers, drill-through navigation, and smart narratives enhanced

usability, while evaluation using the System Usability Scale (SUS) yielded a score of 91.17, reflecting excellent acceptance and satisfaction among users.

In short, the project successfully addressed key challenges in fraud detection such as data imbalance, costly misclassification, and evolving fraud patterns through machine learning and visualization. By selecting Random Forest with Oversampling as the final model and embedding it within an interactive dashboard, the system achieved the objectives of improving detection accuracy, reducing false negatives, and providing actionable insights for fraud management in e-commerce.

7.2 Recommendation

Future studies can extend this work in several directions. First, deep learning techniques, as highlighted in previous studies [12,19] should be explored for fraud detection. Unlike ensemble models, deep learning can capture sequential, non-linear, and relational patterns, making it more effective for detecting complex and evolving fraud strategies.

Second, although this project has already employed CTGAN and TVAE for synthetic data generation, future work could involve training these models on the full dataset or developing hybrid approaches that combine synthetic and real-world data. This would improve the diversity and realism of the generated samples, further enhancing model robustness and privacy preservation.

Third, real-world data integration is important to validate the system's usefulness in industry. Collaborating with e-commerce platforms or financial institutions to test the model on real transactions would help identify challenges like handling large volumes of data, speed of processing and meeting security or regulatory requirements. This would make the system more prepared for actual deployment.

Finally, the system should be moved from Power BI Desktop to Power BI Service for real-time monitoring. Because Python scripts cannot run directly in the Service, the model should be hosted outside (for example in Azure ML or a cloud function). Power BI can then connect to the processed results using dataflows with automatic refresh, live data streams, and alerts. This would remove the need for manual refresh and give fraud analysts faster updates and timely warnings.

REFERENCES

[1] R. Khurana, “Fraud Detection in eCommerce Payment Systems: The Role of Predictive AI in Real-Time Transaction Security and Risk Management,” *International Journal of Applied Machine Learning and Computational Intelligence*, 2020, vol. 10, no. 6, pp. 1–32, Available: https://neuralslate.com/index.php/Machine-Learning-Computational-I/article/view/155.

[2] V. F. Rodrigues *et al.*, “Fraud detection and prevention in e-commerce: A systematic literature review,” *Electronic Commerce Research and Applications*, Oct. 2022, vol. 56, p. 101207, doi: https://doi.org/10.1016/j.elerap.2022.101207.

[3] M. GolyerI, S. Celik, F. Bozyigit and D. Kılınç, “Fraud Detection on E-Commerce Transactions Using Machine Learning Techniques,” *Artificial Intelligence Theory and Applications*, 2023, vol. 3, no. 1, pp. 45–50, Available: https://dergipark.org.tr/en/pub/aita/issue/77113/1273652.

[4] P. K. Sadineni, "Detection of Fraudulent Transactions in Credit Card using Machine Learning Algorithms," *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, Palladam, India, 2020, pp. 659-660, doi: 10.1109/I-SMAC49090.2020.9243545.

[5] S. N. Pundkar and M. Zubei, “Credit Card Fraud Detection Methods: A Review,” *E3S web of conferences*, Jan. 2023, vol. 453, pp. 01015–01015, doi: https://doi.org/10.1051/e3sconf/202345301015.

[6] S. R. Gayam, “AI-Driven Fraud Detection in E-Commerce: Advanced Techniques for Anomaly Detection, Transaction Monitoring, and Risk Mitigation,” *Distributed Learning and Broad Applications in Scientific Research*, 2020, vol. 6, pp. 124–151, Available: https://dlabi.org/index.php/journal/article/view/108

[7] Md. Nur-E-Arefin, “A Comparative Study of Machine Learning Classifiers for Credit Card Fraud Detection,” *International Journal of Innovative Technology and Interdisciplinary Sciences*, Jan. 2020, vol. 3, no. 1, pp. 395–406, doi: https://doi.org/10.15157/ijitis.2020.3.1.395-406.

[8] S. Ray, “Fraud Detection in E-Commerce Using Machine Learning,” *BOHR International Journal of Advances in Management Research*, 2022, vol. 1, no. 1, pp. 7–14, doi: https://doi.org/10.54646/bijamr.002.

[9] M. Puh and L. Brkić, "Detecting Credit Card Fraud Using Selected Machine Learning Algorithms," *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, 2019, pp. 1250-1255, doi: 10.23919/MIPRO.2019.8757212.

[10] U. Porwal and S. Mukund, "Credit Card Fraud Detection in e-Commerce: An Outlier Detection Approach," *arXiv:1811.02196 [cs, stat]*, May 2019, Available: https://arxiv.org/abs/1811.02196.

[11] V. N. Dornadula and S. Geetha, "Credit Card Fraud Detection using Machine Learning Algorithms," *Procedia Computer Science*, 2019, vol. 165, pp. 631–641, doi: https://doi.org/10.1016/j.procs.2020.01.057.

[12] A. Saputra and Suharjito, "Fraud Detection using Machine Learning in e-Commerce," *International Journal of Advanced Computer Science and Applications*, 2019, vol. 10, no. 9, doi: https://doi.org/10.14569/ijacsa.2019.0100943.

[13] O. Adepoju, J. Wosowei, S. lawte and H. Jaiman, "Comparative Evaluation of Credit Card Fraud Detection Using Machine Learning Techniques," *2019 Global Conference for Advancement in Technology (GCAT)*, Bangalore, India, 2019, pp. 1–6, doi: 10.1109/GCAT47503.2019.8978372.

[14] E. Ileberi, Y. Sun and Z. Wang, "Performance Evaluation of Machine Learning Methods for Credit Card Fraud Detection Using SMOTE and AdaBoost," in *IEEE Access*, vol. 9, pp. 165286-165294, 2021, doi: 10.1109/ACCESS.2021.3134330.

[15] S. Najem and S. Kadhem, "An efficient feature engineering method for fraud detection in e-commerce," *Iraqi Journal of Computer Communication Control and System Engineering*, pp. 40–52, Sep. 2021, doi: 10.33103/uot.ijccce.21.3.4.

[16] K. K. Mohbey, M. Z. Khan, and A. Indian, "Credit card fraud prediction using XGBoost," *International Journal of Information Retrieval Research*, vol. 12, no. 2, pp. 1–17, May 2022, doi: 10.4018/ijirr.299940.

[17] Y. Kırelli, S. Arslankaya, and M. T. Zeren. "Detection of Credit Card Fraud in E-Commerce Using Data Mining," *European Journal of Science and Technology,* Nov. 2020, doi: https://doi.org/10.31590/ejosat.747399.

[18] A. Q. Abdulghani, O. N. UCAN and K. M. A. Alheeti, "Credit Card Fraud Detection Using XGBoost Algorithm," *2021 14th International Conference on Developments in eSystems Engineering (DeSE)*, Sharjah, United Arab Emirates, 2021, pp. 487-492, doi: 10.1109/DeSE54285.2021.9719580.

[19] R. Sailusha, V. Gnaneswar, R. Ramesh and G. R. Rao, "Credit Card Fraud Detection Using Machine Learning," *2020 4th International Conference on Intelligent Computing and Control Systems (ICICCS)*, Madurai, India, 2020, pp. 1264-1270, doi: 10.1109/ICICCS48265.2020.9121114.

[20] C. Tejasri, C. S. U. Aryan, D. Deekshith, A. Chintu and T. S. Reddy "FRAUD DETECTION IN E-COMMERCE USING MACHINE LEARNING," *International Research Journal of Modernization in Engineering Technology and Science*, 2022, vol. 4, no. 6, pp. 2924–2926. [Online]. Available: https://www.irjmets.com/

[21] S. Sen and A. Ghosh, "Analysis and Prediction of Parkinson's Disease using Machine Learning Algorithms," *TechRxiv*, 2022. doi: https://doi.org/10.36227/techrxiv.20005703.v1.

[22] W. Wang, G. Chakraborty, and B. Chakraborty, "Predicting the Risk of Chronic Kidney Disease (CKD) Using Machine Learning Algorithm," *Applied Sciences*, vol. 11, no. 1, p. 202, Dec. 2020, doi: https://doi.org/10.3390/app11010202.

[23] E. Martiri, "Synthetic Data Generation," IGI Global, 2024, pp. 118–138. doi: 10.4018/979-8-3693-0255-2.ch005

[24] U. B. Bhadange, S. Jadhav, B. Jadhav, S. Ghatol, and P. Kahale, "Comprehensive Review of Synthetic Data Generation Techniques and Their Applications in Healthcare, Finance, and Marketing," International Journal of Advanced Research in Science, Communication and Technology, Nov. 2024, doi: 10.48175/ijarsct-22066

[25] M. Goyal and Q. H. Mahmoud, "A Systematic Review of Synthetic Data Generation Techniques Using Generative AI," Electronics, vol. 13, no. 17, p. 3509, Sep. 2024, doi: 10.3390/electronics13173509

[26] F. S. Karst, S.-Y. Chong, A. A. Antenor, E.-Y. Lin, M. M. Li, and J. M. Leimeister, "Generative AI for Banks: Benchmarks and Algorithms for Synthetic Financial Transaction Data," Dec. 2024, doi: 10.48550/arxiv.2412.14730

[27] S. Almasi, K. Bahaadinbeigy, H. Ahmadi, S. Sohrabei, and R. Rabiei, "Usability Evaluation of Dashboards: A Systematic Literature Review of Tools," BioMed Research International, vol. 2023, no. 1, pp. 1–11, Feb. 2023, doi: https://doi.org/10.1155/2023/9990933.

APPENDIX

POSTER

FACULTY OF INFORMATION COMMUNICATION AND TECHNOLOGY

UTAR
UNIVERSITI TUNKU ABDUL RAHMAN

Fraud Detection using Machine Learning in e-Commerce

Introduction

E-commerce has transformed traditional business through digital technologies, offering convenience and global reach. However, the rise in online transactions has led to increased fraud, making machine learning methods essential for detecting and preventing fraudulent activities in real time.

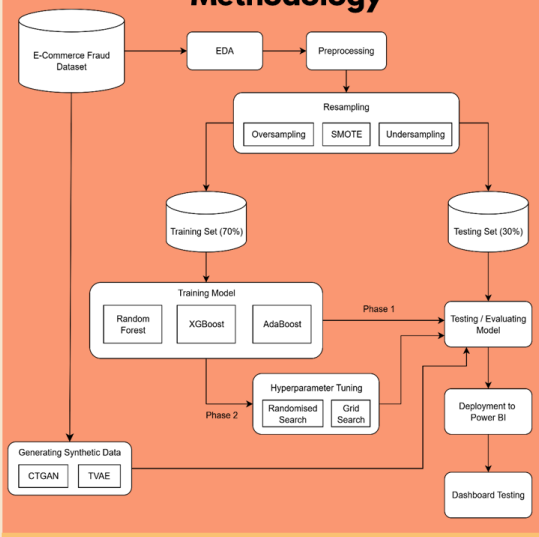
Problem Statements

- Data imbalance in fraud detection datasets.
- Misclassification in machine learning, where the models treat all errors equally.
- Evolving nature of fraud, where fraudulent patterns change over time.

Objectives

- To enhance fraud detection performance by addressing data imbalance through resampling techniques.
- To develop ensemble models that reduce misclassification errors.
- To visualize fraud detection model performance and fraud patterns using Power BI.

Methodology



Discussion

- Ensemble models (Random Forest, XGBoost & AdaBoost) show strong performance in fraud detection.
- SMOTE & Oversampling address class imbalance well, significantly improving recall and reducing false negatives.
- XGBoost and Random Forest outperformed AdaBoost.
- Hyperparameter tuning brought little improvement; final model used default Random Forest + Oversampling.
- On Kaggle test set, final model achieved F1-score = 0.90.
- Dashboard integration in Power BI provided real-time monitoring and actionable insights.

Conclusion

The system successfully improved detection accuracy, reduced costly false negatives, and enhanced fraud visualization through integration with a Power BI dashboard. For future work, deep learning approaches, real-time data integration, and deployment on Power BI Service with cloud hosting (e.g., Azure ML or cloud functions) are recommended to enable real-time fraud alerts.

Project Developer: Ang Su Huan
Project Supervisor: Ms Nurul Syafidah Binti Jamil