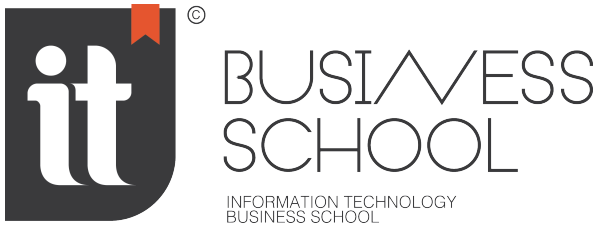
République Tunisienne
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Ecole Supérieure Privée des Technologies de l'Information et de Management de Nabeul



Thesis submitted in partial fulfillment of the requirements for the

National Diploma in Computer Engineering
Specialization: Business Intelligence

Prepared by

Adem Mami

Credit Card Fraud Detection: End-to-End MLOps Pipeline

Academic Supervisor: Mr. Name **Professional Supervisor:** Mr. Name

Thesis completed at

IT Business School (ITBS)

Academic Year 2024-2025

BUSINESS SCHOOL
INFORMATION TECHNOLOGY BUSINESS SCHOOL

Signature et cachet de l'entreprise

BUSINESS SCHOOL
INFORMATION TECHNOLOGY BUSINESS SCHOOL

Signature de l'encadrant académique

Dedication

To ... for their sacrifice and support,
as a token of my infinite gratitude and deep attachment

To all those who are dear to me...

Acknowledgment

Je n'aurais jamais pu réaliser ce projet sans la précieuse aide et sans le soutien d'un grand nombre de personnes dont la générosité, la bonne humeur et l'intérêt manifestés à l'égard de mon PFE m'ont permis de progresser.

Ma reconnaissance va à ceux qui ont plus particulièrement assuré le soutien affectif de ce travail : ma famille ainsi que mes amis. Mes parents...

Abstract

Abstract

Credit card fraud represents losses of several billion dollars annually for the financial industry. This project presents the design and implementation of a complete MLOps (Machine Learning Operations) pipeline for automated fraud detection.
Using the public Kaggle dataset containing 284,807 real transactions, of which only 0.172% are fraudulent, we developed a system capable of identifying suspicious activities with high accuracy. The Random Forest Classifier model, trained with a class weighting strategy to handle the extreme imbalance, achieves the following performance:

- ROC-AUC: 0.9766 (excellent discrimination capability)
- F1-Score: 0.8223 (good precision/recall balance)
- Recall: 82.65% (detection of the majority of frauds)

The implemented MLOps infrastructure ensures end-to-end reproducibility and traceability:

- **DVC** for data versioning and pipeline orchestration
- **MLflow** for experiment tracking and model registry
- **FastAPI** for exposing the model via a high-performance REST API
- **Docker** for containerization and portable deployment

A modern user interface developed with Next.js and Tailwind CSS allows analysts to interact with the system, whether for single or batch predictions.

Keywords: MLOps, Fraud Detection, Machine Learning, Random Forest, Imbalanced Classification, DVC, MLflow, FastAPI, Docker, Next.js

Contents

List of Figures

List of Tables

Introduction Générale

Context and Motivation

In an increasingly digital world, electronic financial transactions have become the norm. Every day, millions of credit card payments are made worldwide, representing trillions of dollars in transactions. This exponential growth is unfortunately accompanied by a proportional increase in fraud attempts. According to the 2023 Nilson Report, global payment card fraud losses exceeded $32 billion, with alarming projections for the coming years.
Faced with this growing threat, financial institutions are investing heavily in automated detection systems based on artificial intelligence. Traditional rule-based methods are proving insufficient against increasingly sophisticated fraudsters who continually adapt their techniques.

Problem Statement

Credit card fraud detection presents several major technical challenges:

- **Class Imbalance**: Fraudulent transactions represent less than 0.2% of all transactions, creating an extreme imbalance problem that can bias classification models.
- **Evolving Patterns**: Fraudsters constantly adapt their methods, requiring models capable of generalizing and detecting new types of fraud.
- **Real-time Constraints**: Decisions must be made in milliseconds to avoid impacting user experience.
- **Asymmetric Error Costs**: A false negative (undetected fraud) costs much more than a false positive (blocked legitimate transaction).

Beyond these Machine Learning challenges, deploying and maintaining such systems in production presents its own complexities. This is where **MLOps** (Machine Learning Operations) comes into play.

Project Objectives

This project aims to design and implement a complete MLOps pipeline for credit card fraud detection. The specific objectives are:

1. **Develop a high-performance model** for classification capable of detecting fraudulent transactions with high precision while minimizing false positives.

2. **Establish a reproducible pipeline** using DVC (Data Version Control) to version data and orchestrate processing steps.

3. **Implement experiment tracking** with MLflow to log hyperparameters, metrics, and artifacts for each training run.

4. **Deploy the model** as a REST API with FastAPI, production-ready.

5. **Create a modern user interface** with Next.js allowing analysts to interact with the system.

6. **Containerize the application** with Docker to facilitate deployment across different cloud infrastructures.

Report Organization

This report is structured as follows:

Chapter 1 presents the general context of the project, the importance of fraud detection in the banking sector, and precisely defines the problem to be solved.

Chapter 2 provides a state of the art of fraud detection techniques and MLOps practices, comparing different existing approaches.

Chapter 3 details the exploratory analysis of the dataset used and the preprocessing steps implemented.

Chapter 4 describes the modeling process, algorithm selection, training, and performance evaluation.

Chapter 5 presents the deployment architecture, REST API, web application, and containerization aspects.

Finally, we conclude with a synthesis of the work accomplished and propose perspectives for improvement for future work.

Chapter 1

Contexte Général et Problématique

1.1 Introduction

This chapter presents the general context of this project. We begin with an overview of the electronic payments sector and the growing importance of transaction security. Then, we detail the specific challenges of fraud detection and the associated issues.

1.2 The Electronic Payments Sector

1.2.1 Market Evolution

The electronic payments market has experienced exponential growth over recent decades. With the advent of e-commerce, contactless payments, and digital wallets, the volume of transactions processed daily has reached unprecedented levels.

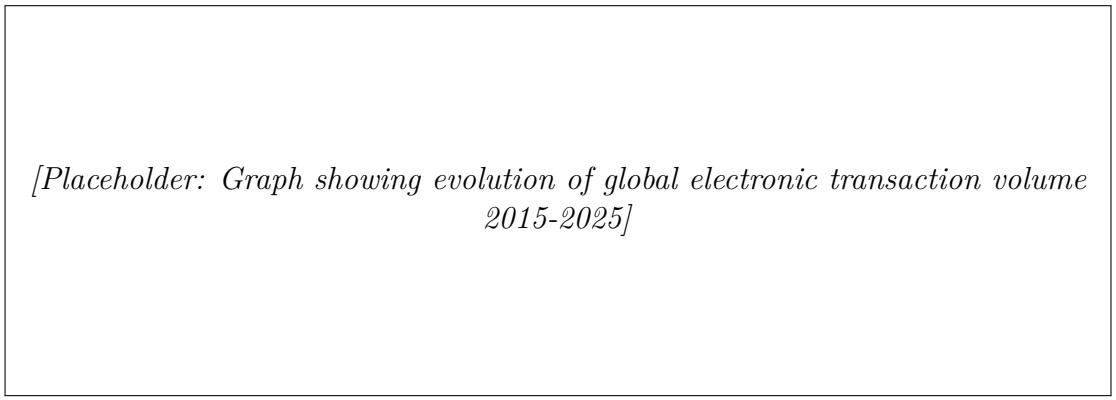


Figure 1.1: Evolution of global electronic transaction volume

The main market players include:

- Card networks (Visa, Mastercard, American Express)
- Issuing and acquiring banks
- Payment processors (Stripe, PayPal, Adyen)
- Fintechs and neobanks

1.2.2 Economic Impact of Fraud

Credit card fraud represents a considerable cost for the financial industry. These losses manifest in several forms:

Cost Type	**Estimated Impact**
Direct losses (refunds)	\$32+ billion/year
Detection operational costs	\$8+ billion/year
Loss of customer trust	Difficult to quantify
Regulatory fines (PCI-DSS)	Variable

Table 1.1: Economic impact of credit card fraud

1.3 Types of Credit Card Fraud

Credit card frauds can be classified into several main categories:

1.3.1 Lost or Stolen Card Fraud

This type of fraud occurs when a fraudster physically uses a card that does not belong to them. Although this is the most traditional form, it remains common at physical points of sale.

1.3.2 Card-Not-Present (CNP) Fraud

CNP fraud occurs during transactions where the card is not physically present, typically for online purchases. This is the fastest-growing form of fraud with the rise of e-commerce.

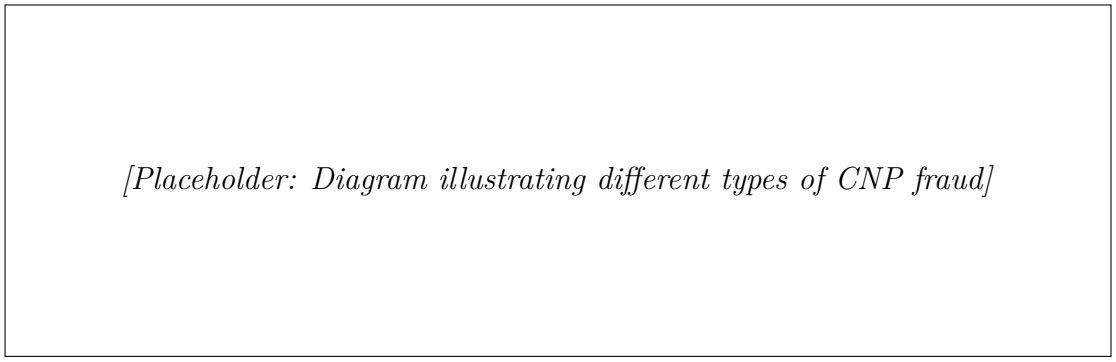


Figure 1.2: Taxonomy of Card-Not-Present frauds

1.3.3 Identity Theft Fraud

In this case, the fraudster obtains enough personal information to open new accounts or modify existing accounts in the victim's name.

1.3.4 Friendly Fraud (Chargeback Fraud)

Also called "chargeback fraud," this occurs when a legitimate customer makes a purchase then disputes the transaction with their bank, claiming they did not make it.

1.4 Regulatory Framework

1.4.1 PCI-DSS Standards

The Payment Card Industry Data Security Standard (PCI-DSS) defines security requirements for organizations that process payment card data. Compliance with these standards is mandatory and non-compliance can result in significant fines.

1.4.2 PSD2 Directive and Strong Authentication

In Europe, the PSD2 directive (Payment Services Directive 2) mandates Strong Customer Authentication (SCA) for electronic payments, adding an additional layer of security.

1.5 Problem Definition

1.5.1 Problem Statement

Problem Statement

How to design and deploy a credit card fraud detection system that is:

1. Accurate enough to detect the majority of frauds while minimizing false positives
2. Capable of processing transactions in real-time
3. Easily maintainable and scalable through MLOps practices
4. Reproducible and traceable to meet audit requirements

1.5.2 Technical Constraints

The constraints identified for this project are:

Performance The model must achieve a recall greater than 80% while maintaining acceptable precision.

Latency Predictions must be provided in less than 100 milliseconds.

Scalability The system must be able to handle load spikes during high-activity periods (holidays, sales).

Interpretability Model decisions must be explainable to fraud analysts.

1.6 Summary

This chapter has presented the economic and regulatory context of credit card fraud detection. The different types of fraud have been identified, along with their impact on the industry. The problem has been formally defined, laying the groundwork for the following chapters that will detail the state of the art and our solution approach.

Chapter 2

État de l'Art

2.1 Introduction

This chapter presents a state of the art of fraud detection techniques and MLOps practices. We first examine traditional and modern fraud detection approaches, then introduce the fundamental concepts of MLOps that guide our solution architecture.

2.2 Fraud Detection Techniques

2.2.1 Rule-Based Approaches

Historically, fraud detection systems relied on manually defined expert rules. These rules typically include:

- Amount thresholds (transactions above a certain amount)
- Transaction frequency (number of transactions within a time window)
- Geographic location (transactions from high-risk countries)
- Unusual purchasing patterns

Advantages: Simplicity, interpretability, ease of updating.
Disadvantages: Rigidity, inability to detect new patterns, high false positive rate.

2.2.2 Statistical Approaches

Traditional statistical methods include:

Logistic Regression Simple but effective linear model for binary classification.

Discriminant Analysis Class separation based on linear combinations of features.

Anomaly Detection Identification of transactions that significantly deviate from normal behavior.

2.2.3 Machine Learning Approaches

Decision Trees and Ensembles

Ensemble methods are particularly popular for fraud detection:

Figure 2.1: Random Forest Classifier Architecture

- **Random Forest**: Ensemble of decision trees trained on bootstrap subsamples. Robust to overfitting and capable of handling imbalanced data with the `class_weight='balanced'` option.
- **Gradient Boosting (XGBoost, LightGBM)**: Sequential construction of trees, each tree correcting the errors of the previous one. Very performant but more sensitive to hyperparameters.
- **Isolation Forest**: Anomaly detection algorithm based on the principle that anomalies are easier to isolate.

Neural Networks

Deep learning approaches offer superior performance for certain use cases:

- **Multilayer Perceptrons (MLP)**: Fully-connected networks for classification.
- **Autoencoders**: Unsupervised learning for anomaly detection.
- **LSTM/GRU**: Recurrent networks to capture temporal dependencies in transaction sequences.

Method	Accuracy	Interpretability	Training Time	Latency
Manual Rules	Low	Very High	N/A	Very Low
Logistic Regression	Medium	High	Low	Very Low
Random Forest	High	Medium	Medium	Low
XGBoost	Very High	Medium	Medium	Low
Deep Learning	Very High	Low	High	Medium

Table 2.1: Comparison of different fraud detection approaches

2.2.4 Comparison of Approaches

2.3 Introduction to MLOps

2.3.1 Definition and Principles

MLOps (Machine Learning Operations) is a discipline that combines Machine Learning, DevOps, and data engineering to deploy and maintain ML systems in production reliably and efficiently.

[Placeholder: MLOps lifecycle diagram showing phases: Data → Model → Deploy → Monitor → Retrain]

Figure 2.2: MLOps Lifecycle

The fundamental principles of MLOps include:

1. **Automation**: Reduce manual interventions at each pipeline step.
2. **Reproducibility**: Ensure that each experiment can be exactly reproduced.
3. **Versioning**: Version not only code, but also data and models.
4. **Monitoring**: Continuously monitor model performance in production.
5. **Testing**: Validate data quality, code, and models.

2.3.2 MLOps Maturity Levels

Google defines three MLOps maturity levels:

Level 0 - Manual Entirely manual process, no automated pipeline. Suitable for prototypes.

Level 1 - ML Pipeline Automated pipeline for training, but manual deployment.

Level 2 - CI/CD for ML Complete automation including continuous training and automatic deployment.

Our project targets **Level 1** with elements of Level 2.

2.4 MLOps Tools Used

2.4.1 DVC (Data Version Control)

DVC is an open-source tool for data and ML pipeline versioning. It allows:

- Versioning large files (data, models) without storing them in Git
- Defining reproducible pipelines via a `dvc.yaml` file
- Sharing data between collaborators via remote storage (S3, GCS, Azure)

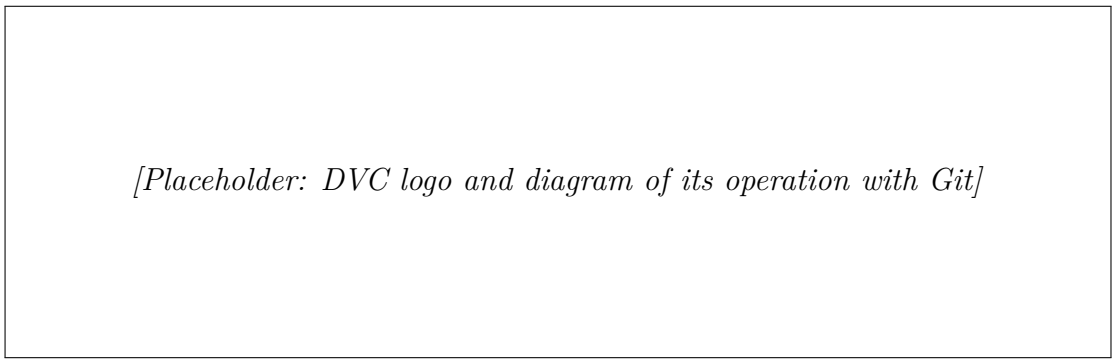


Figure 2.3: DVC Integration with Git

2.4.2 MLflow

MLflow is an open-source platform for managing the ML model lifecycle. Its main components are:

Tracking Recording parameters, metrics, and artifacts for each run.

Projects Packaging ML code for reproducibility.

Models Standard format for saving and deploying models.

Registry Centralized management of model versions.

2.4.3 FastAPI

FastAPI is a modern Python framework for building REST APIs. Its advantages include:

- Performance close to asynchronous frameworks (Starlette)
- Native data validation (Pydantic)
- Automatic interactive documentation (Swagger/OpenAPI)
- Python type hints support for better maintainability

2.4.4 Docker

Docker enables application containerization to ensure a consistent execution environment:

- Dependency isolation
- Portability between environments (dev, staging, prod)
- Easy deployment on cloud platforms (AWS ECS, GCP Cloud Run, Azure Container Instances)

2.5 Related Work

Several academic and industrial works have addressed fraud detection:

- **Dal Pozzolo et al. (2015)** proposed an adaptive approach using undersampling techniques to handle class imbalance.
- **Google Pay** uses recurrent neural networks to analyze transaction sequences in real-time.
- **PayPal** has developed a hybrid system combining expert rules and ML, processing over 5 billion transactions per year.

2.6 Summary

This chapter has presented the state of the art of fraud detection techniques, from the simplest (rules) to the most sophisticated (deep learning). We have also introduced MLOps concepts and the tools we will use in our implementation. The choice of Random Forest, combined with a robust MLOps infrastructure, represents a good compromise between performance, interpretability, and ease of deployment.

Chapter 3

Analyse et Préparation des Données

3.1 Introduction

This chapter presents the dataset used for this project and details the exploratory analysis and preprocessing steps. A thorough understanding of the data is essential for building a performant model and avoiding pitfalls related to imbalanced data.

3.2 Dataset Presentation

3.2.1 Data Source

The dataset used comes from the Kaggle platform and was provided by the Machine Learning Group of the Free University of Brussels (ULB). It contains real transactions made by European credit card holders in September 2013.

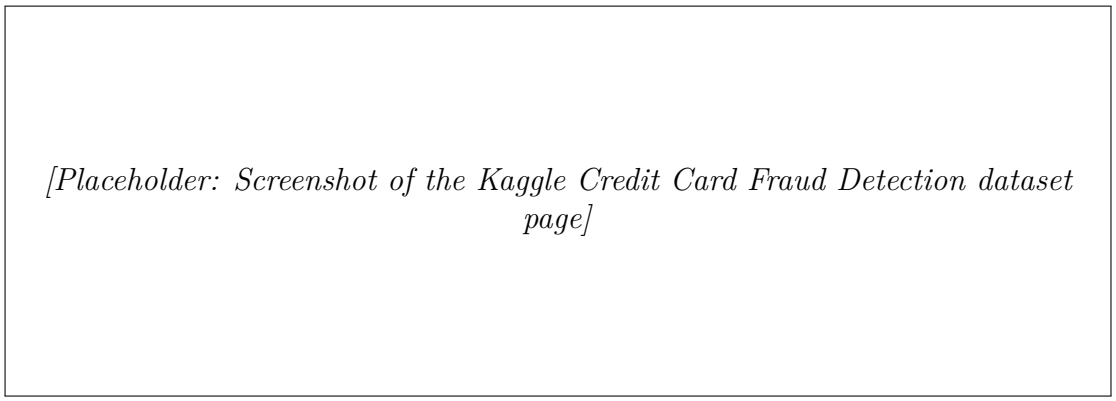


Figure 3.1: Kaggle Dataset - Credit Card Fraud Detection

3.2.2 Dataset Characteristics

3.2.3 Variable Description

The dataset contains the following variables:

Characteristic	Value
Total number of transactions	284,807
Number of fraudulent transactions	492
Number of normal transactions	284,315
Fraud rate	0.172%
Number of features	30
File size	150 MB

Table 3.1: General characteristics of the dataset

Time Number of seconds elapsed between this transaction and the first transaction in the dataset. This variable allows analysis of temporal patterns.

V1 to V28 These 28 features are the result of a PCA (Principal Component Analysis) transformation applied to the original data. For confidentiality reasons, the original features are not disclosed.

Amount Transaction amount in euros. This is the only non-transformed variable besides Time and Class.

Class Binary target variable: 0 for a normal transaction, 1 for fraud.

3.3 Exploratory Data Analysis (EDA)

3.3.1 Target Variable Distribution

Analysis of the target variable distribution reveals an extreme imbalance between the two classes:

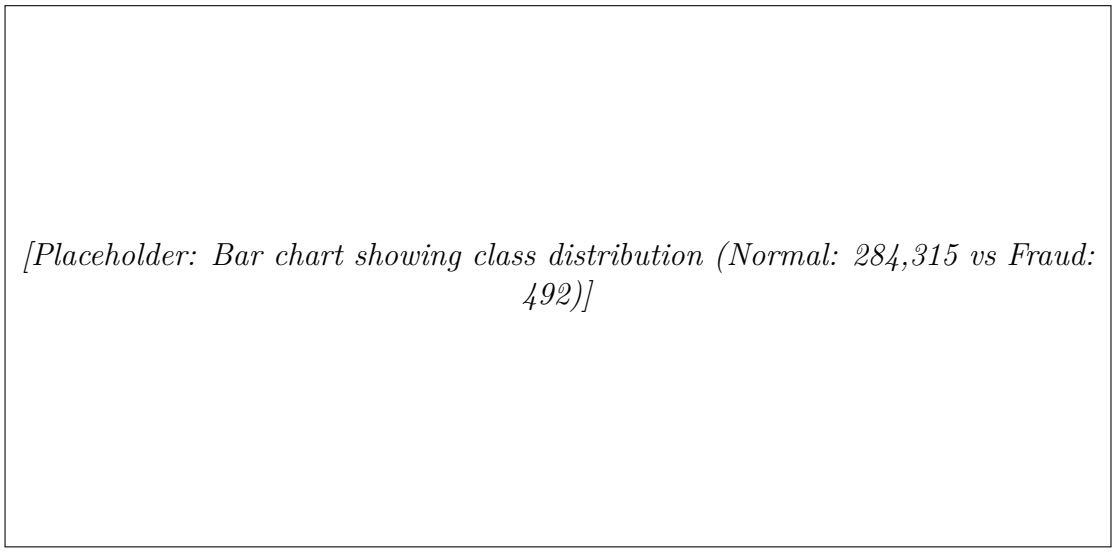


Figure 3.2: Class Distribution - Extreme Imbalance

This imbalance (578:1 ratio) poses a major challenge for training classification models, as a naive classifier could achieve 99.83% accuracy by systematically predicting the majority class.

3.3.2 Time Variable Analysis

Analysis of the Time variable reveals interesting patterns:

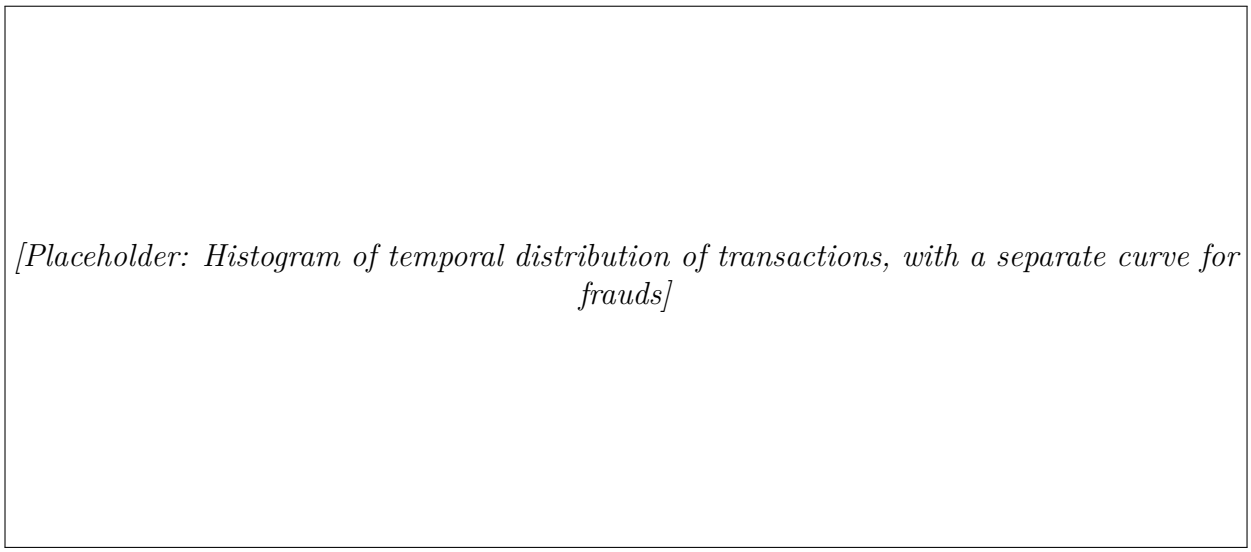


Figure 3.3: Temporal Distribution of Transactions

Key observations are:

- Transactions are spread over approximately 48 hours (172,800 seconds).
- A cyclical day/night pattern is visible for normal transactions.
- Frauds appear less correlated with normal temporal patterns.

3.3.3 Amount Variable Analysis

Transaction amount analysis shows significant differences between classes:

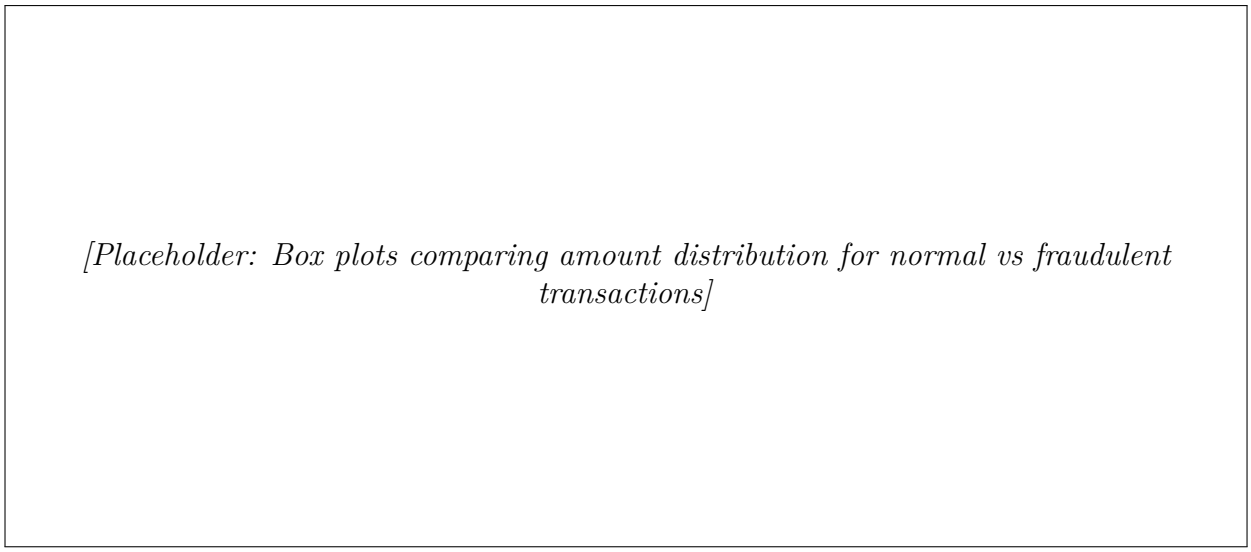


Figure 3.4: Amount Distribution by Class

Statistic	Normal	Fraud
Mean	$88.29	$122.21
Median	$22.00	$9.25
Standard Deviation	$250.11	$256.68
Maximum	$25,691.16	$2,125.87

Table 3.2: Descriptive statistics of amounts by class

3.3.4 Correlation Analysis

Analysis of the correlation matrix between PCA features and the target variable helps identify the most predictive variables:

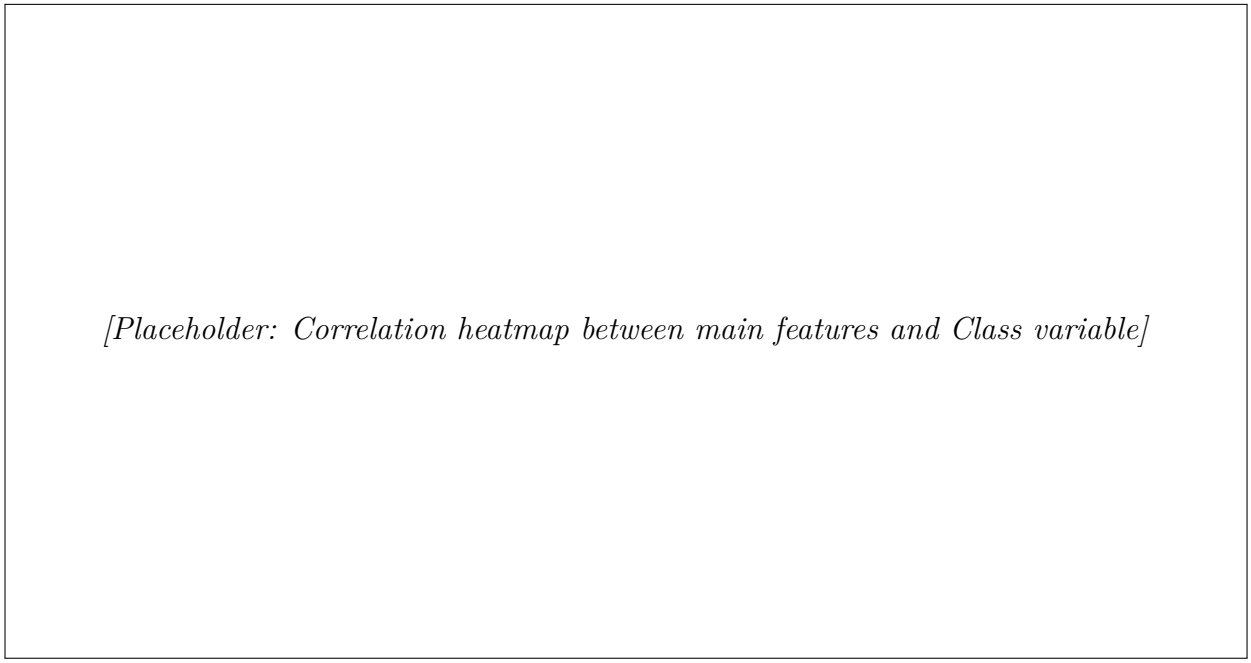


Figure 3.5: Correlation Matrix of Main Features

The features most correlated (positively or negatively) with fraud are:

- V17 (strong negative correlation)
- V14 (strong negative correlation)
- V12 (strong negative correlation)
- V10 (negative correlation)
- V11 (positive correlation)

3.4 Data Preprocessing

3.4.1 Preprocessing Pipeline

Preprocessing is implemented in the `src/data/make_dataset.py` script and orchestrated by DVC. The pipeline includes the following steps:

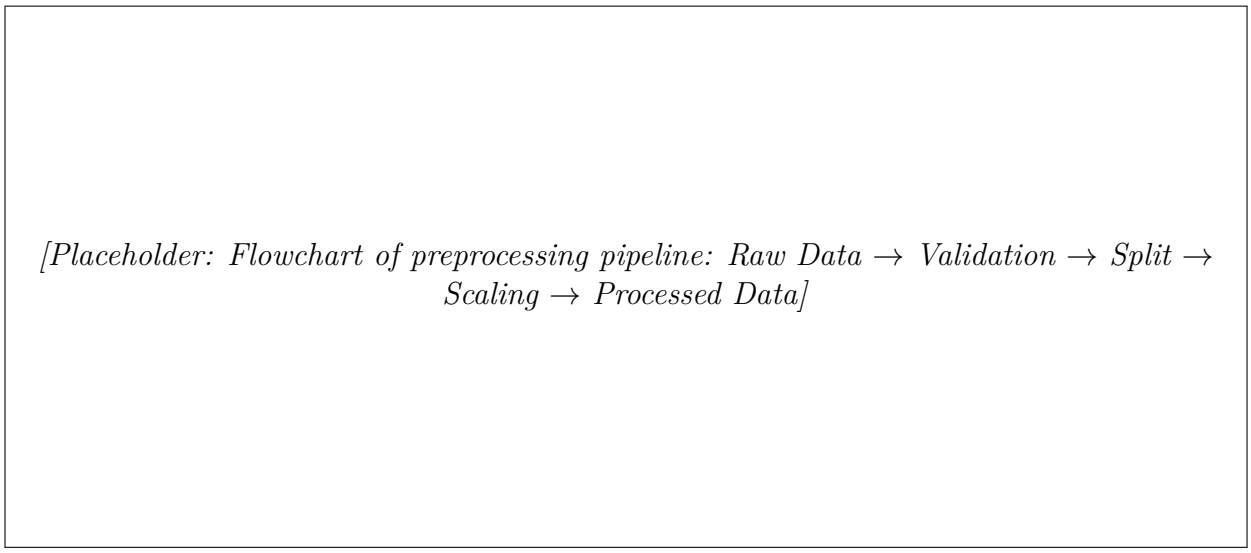


Figure 3.6: Data Preprocessing Pipeline

3.4.2 Train/Test Split

Data is divided into training and test sets with the following parameters:

- **Ratio**: 80% training / 20% test
- **Stratification**: Maintaining the fraud ratio in each set
- **Random State**: 42 (for reproducibility)

Set	Total	Frauds	Rate
Training	227,845	394	0.173%
Test	56,962	98	0.172%

Table 3.3: Data distribution after split

3.4.3 Feature Normalization

The `Time` and `Amount` variables are normalized using a StandardScaler to put them on the same scale as the PCA components (already normalized):

$$z = \frac{x - \mu}{\sigma} \tag{3.1}$$

where μ is the mean and σ the standard deviation calculated on the training set only (to avoid data leakage).

3.4.4 Handling Class Imbalance

Several strategies exist for handling class imbalance:

Undersampling Reducing the number of majority class examples. Risk of losing important information.

Oversampling (SMOTE) Generating synthetic examples of the minority class.

Class Weighting Assigning a higher weight to minority class examples during training.

In this project, we use **class weighting** via the `class_weight='balanced'` parameter of Random Forest, which automatically adjusts weights based on the class ratio.

3.5 Versioning with DVC

3.5.1 Pipeline Configuration

The `dvc.yaml` file defines the preprocessing pipeline:

```
stages:
  make_dataset:
    cmd: python src/data/make_dataset.py
    deps:
      - src/data/make_dataset.py
      - params.yaml
      - data/raw/creditcard.csv
    outs:
      - data/processed/train.csv
      - data/processed/test.csv
```

3.5.2 Benefits of Versioning

Using DVC provides several advantages:

- **Reproducibility**: The `dvc repro` command reproduces exactly the same pipeline.
- **Traceability**: Each data version is linked to a Git commit.
- **Collaboration**: Large files are shared via remote storage.

3.6 Summary

This chapter presented an in-depth analysis of the Credit Card Fraud Detection dataset. The extreme class imbalance (0.172% frauds) was identified as the main challenge. The preprocessing pipeline, including stratified split and normalization, has been implemented and versioned with DVC. The following chapter will detail model training and performance evaluation.

Chapter 4

Modélisation et Entraînement

4.1 Introduction

This chapter presents the complete modeling process, from algorithm selection to performance evaluation. We detail the use of MLflow for experiment tracking and analyze the results obtained.

4.2 Model Selection

4.2.1 Selection Criteria

The algorithm choice was guided by several criteria:

1. **Performance**: Ability to achieve high recall while maintaining acceptable precision.
2. **Robustness**: Resistance to overfitting, particularly important with imbalanced data.
3. **Interpretability**: Ability to explain decisions to fraud analysts.
4. **Inference Time**: Low latency for real-time predictions.
5. **Deployment Ease**: Compatibility with chosen MLOps tools.

4.2.2 Random Forest Classifier

After evaluating several algorithms, we selected the **Random Forest Classifier** for the following reasons:

- **Bagging**: Training on bootstrap subsamples reduces variance and overfitting.
- **Feature Subsampling**: Each split considers a random subset of features, improving tree diversity.
- **Class Weighting**: Native support for class weighting via `class_weight='balanced'`.
- **Feature Importance**: Automatic calculation of variable importance.
- **Parallelization**: Training easily parallelizable across multiple CPU cores.

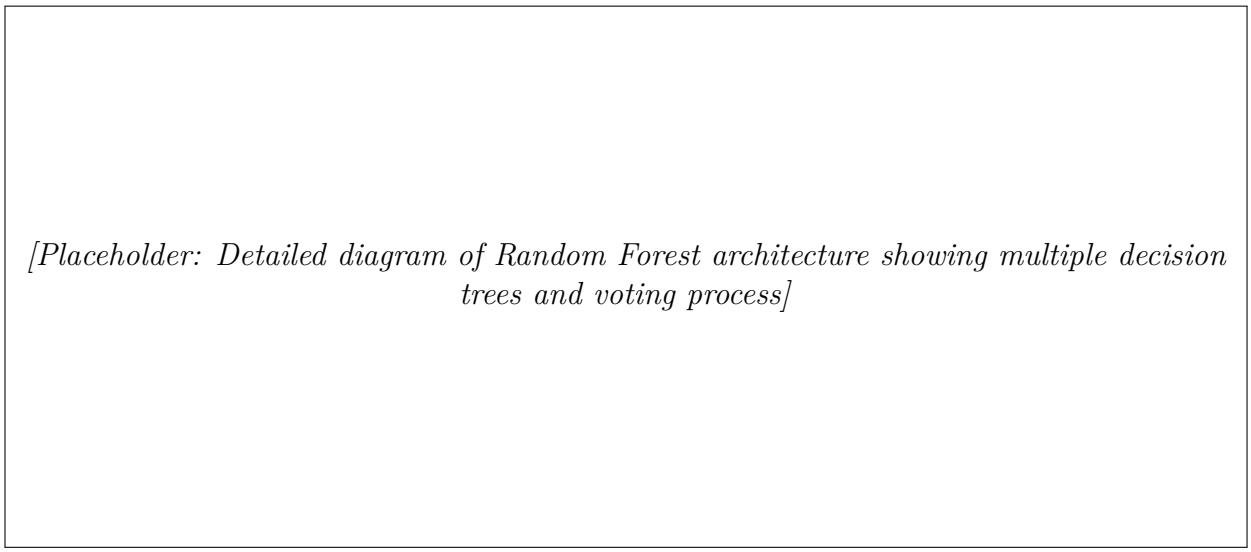


Figure 4.1: Detailed Random Forest Architecture

4.3 Hyperparameter Configuration

4.3.1 Main Parameters

Model hyperparameters are centralized in the `params.yaml` file:

Parameter	Value	Justification
n_estimators	100	Good compromise between performance and training time
max_depth	10	Limits depth to prevent overfitting
class_weight	balanced	Automatically compensates for class imbalance
random_state	42	Ensures reproducibility
n_jobs	-1	Uses all available CPU cores

Table 4.1: Random Forest hyperparameter configuration

4.3.2 Impact of Class Weighting

Class weighting is calculated automatically according to the formula:

$$w_i = \frac{n_{samples}}{n_{classes} \times n_{samples_i}} \tag{4.1}$$

Which gives approximately:

- Weight for class 0 (Normal): ≈ 0.5
- Weight for class 1 (Fraud): ≈ 289 (578 times higher)

4.4 Training with MLflow

4.4.1 MLflow Configuration

MLflow is used for experiment tracking. The configuration includes:

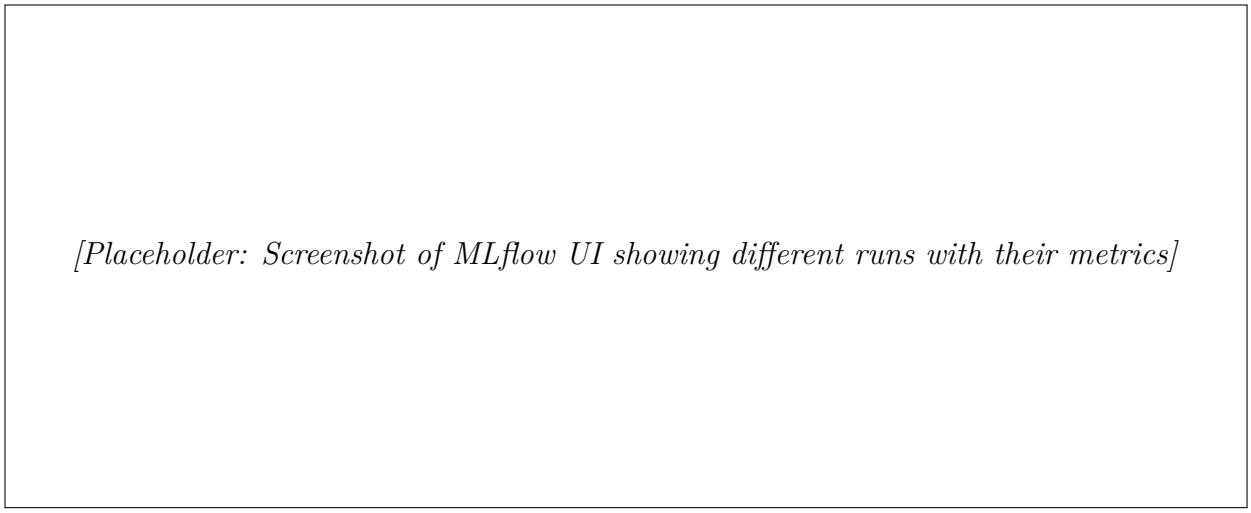


Figure 4.2: MLflow Tracking Interface

```
# MLflow configuration in train_model.py
mlflow.set_tracking_uri("sqlite:///mlflow.db")
mlflow.set_experiment("fraud-detection-mlflow")

with mlflow.start_run():
    # Log parameters
    mlflow.log_params(params)

    # Train model
    model.fit(X_train, y_train)

    # Log metrics
    mlflow.log_metrics(metrics)

    # Save model
    mlflow.sklearn.log_model(model, "model")
```

4.4.2 Recorded Metrics

Each MLflow run records the following information:

Parameters n_estimators, max_depth, class_weight, random_state

Metrics Precision, Recall, F1-Score, ROC-AUC, PR-AUC

Artifacts Serialized model (.pkl), confusion matrices, ROC curves

4.5 Performance Evaluation

4.5.1 Metrics Used

For an imbalanced classification problem, the following metrics are particularly relevant:

Definition 4.1 (Precision)**.**

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

Proportion of positive predictions that are correct.

Definition 4.2 (Recall (Sensitivity))**.**

$$Recall = \frac{TP}{TP + FN} \tag{4.3}$$

Proportion of actual frauds that are detected.

Definition 4.3 (F1-Score)**.**

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.4}$$

Harmonic mean of precision and recall.

4.5.2 Results Obtained

Performance obtained on the test set is as follows:

Metric	Value	Interpretation
ROC-AUC	0.9766	Excellent discrimination capability
F1-Score	0.8223	Good precision/recall balance
Precision	81.82%	82% of alerts are true frauds
Recall	82.65%	83% of frauds are detected

Table 4.2: Model performance on test set

4.5.3 Confusion Matrix

Matrix analysis:

- **True Negatives (TN)**: 56,854 correctly classified normal transactions
- **False Positives (FP)**: 10 normal transactions falsely flagged
- **False Negatives (FN)**: 17 undetected frauds (high cost!)
- **True Positives (TP)**: 81 correctly detected frauds

4.5.4 ROC Curve

4.5.5 Precision-Recall Curve

The Precision-Recall curve is particularly important for imbalanced datasets:

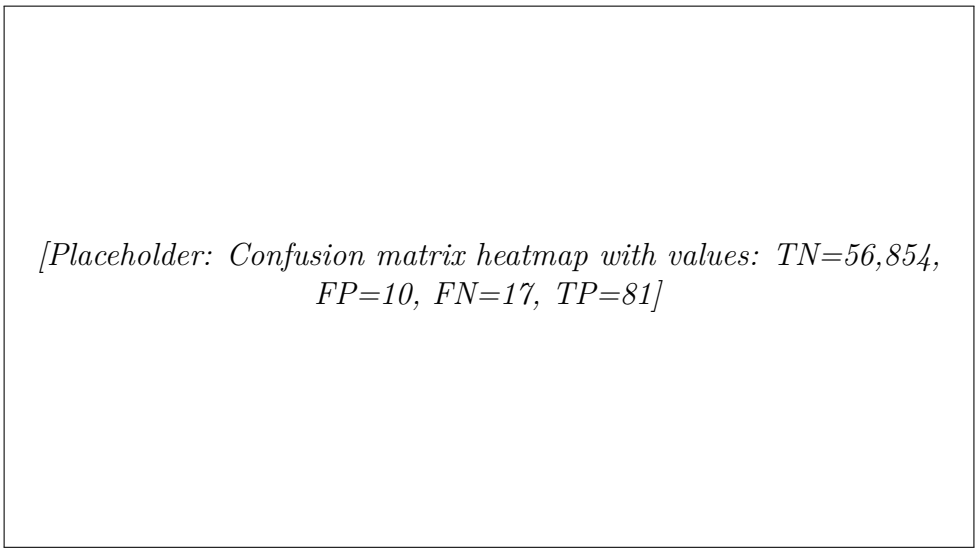


Figure 4.3: Confusion Matrix on Test Set

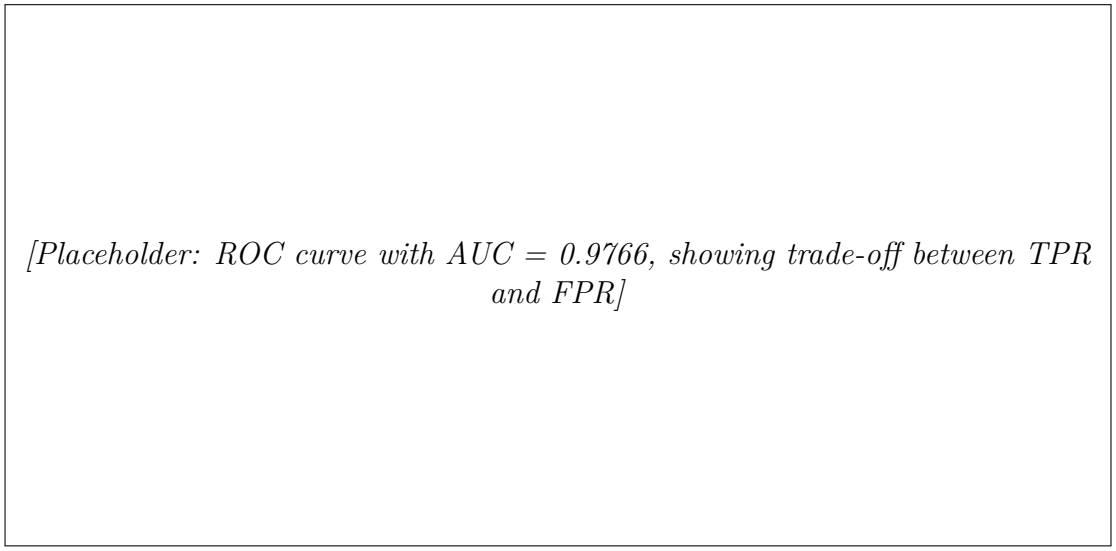


Figure 4.4: ROC Curve (AUC = 0.9766)

4.6 Feature Importance

Feature importance analysis reveals the most discriminative variables:
The most important features are:

1. V17 (relative importance: 0.15)
2. V14 (relative importance: 0.12)
3. V12 (relative importance: 0.10)
4. V10 (relative importance: 0.08)
5. Amount (relative importance: 0.07)

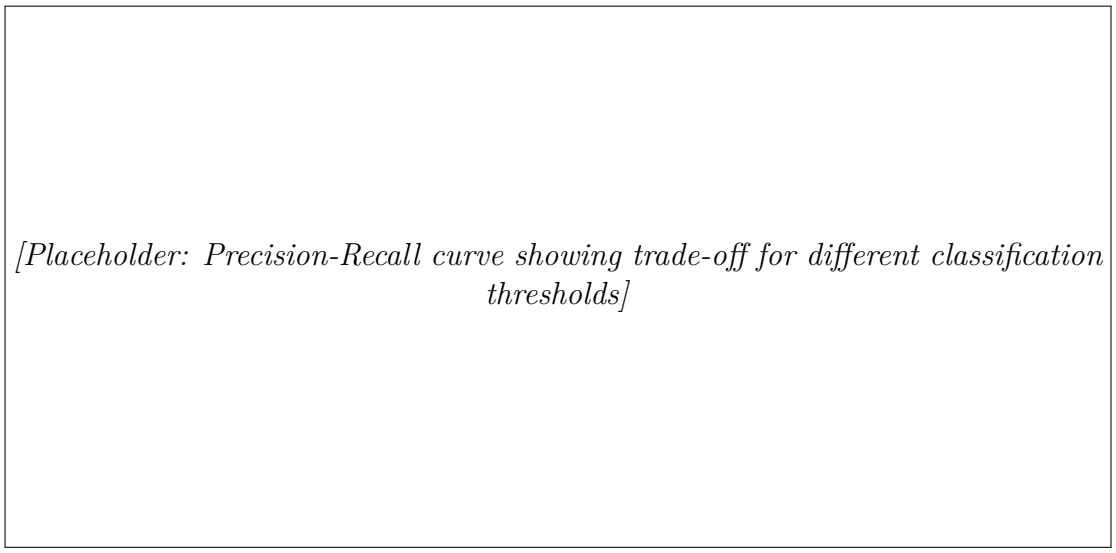


Figure 4.5: Precision-Recall Curve

4.7 Classification Threshold Selection

4.7.1 Threshold Impact

The default classification threshold is 0.5, but it can be adjusted based on the use case:

Threshold	Precision	Recall	F1-Score
0.3	65%	92%	0.76
0.5	82%	83%	0.82
0.7	90%	70%	0.79

Table 4.3: Threshold impact on metrics

4.7.2 Cost-Benefit Analysis

In a real context, threshold choice depends on the relative cost of errors:

- **FN Cost**: Amount of undetected fraud + management fees
- **FP Cost**: Investigation cost + customer dissatisfaction

If the average cost of a fraud is much higher than the investigation cost, a lower threshold (favoring recall) is recommended.

4.8 Model Saving and Registry

4.8.1 Model Serialization

The trained model is saved in pickle format:

```
import joblib
joblib.dump(model, "models/rf_model.pkl")
```

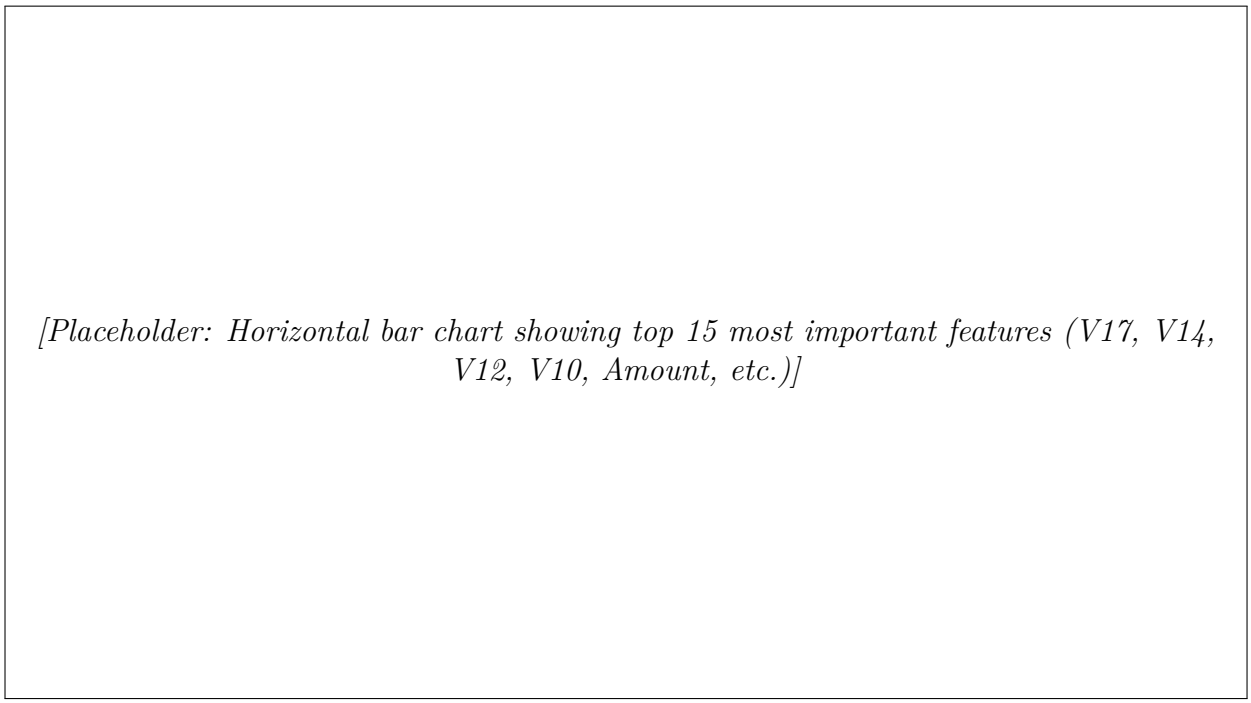


Figure 4.6: Feature Importance (Top 15)

4.8.2 MLflow Registry

The model is also registered in the MLflow Model Registry:

```
mlflow.register_model(
    model_uri=f"runs:/{run_id}/model",
    name="fraud_detection_rf"
)
```

This allows:

- Model versioning (v1, v2, ...)
- Phase tagging (Staging, Production)
- Deployment facilitation

4.9 Summary

This chapter presented the complete modeling process. The Random Forest Classifier, with its configuration optimized for imbalanced data, achieves excellent performance (F1 = 0.82, AUC = 0.98). MLflow integration ensures complete traceability of experiments. The following chapter will detail the deployment of this model to production.

Chapter 5

Déploiement et Application Web

5.1 Introduction

This chapter presents the deployment architecture of the fraud detection system. We detail the REST API developed with FastAPI, the user interface built with Next.js, and containerization with Docker.

5.2 Overall System Architecture

5.2.1 Overview

The fraud detection system consists of several interconnected components:

Frontend Next.js web application for user interaction

Backend API FastAPI service exposing prediction endpoints

ML Model Trained Random Forest, loaded in memory at startup

Monitoring Request logging and usage statistics

MLOps DVC for pipeline, MLflow for tracking

5.2.2 Data Flow

1. User submits a transaction via the web interface
2. Frontend sends a POST request to the API
3. API validates data and calls the model
4. Model returns fraud probability
5. API logs the request and sends response
6. Frontend displays result to user

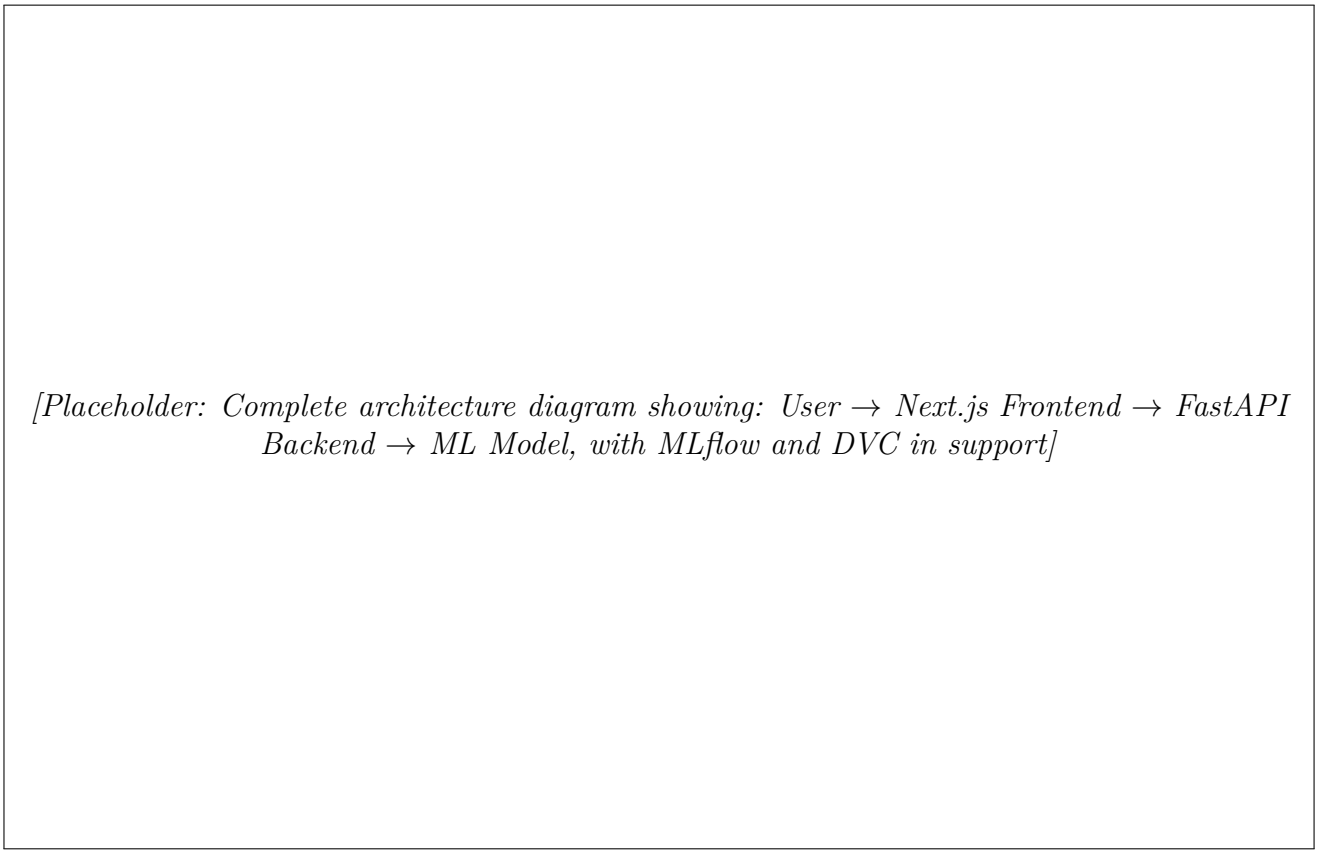


Figure 5.1: Overall Architecture of the Fraud Detection System

5.3 REST API with FastAPI

5.3.1 Why FastAPI

FastAPI was chosen for its many advantages:

- **Performance**: Based on Starlette and Uvicorn (ASGI), very fast
- **Validation**: Native Pydantic integration for data validation
- **Documentation**: Automatic Swagger/OpenAPI documentation generation
- **Type Hints**: Uses Python type annotations
- **Async Support**: Native support for asynchronous operations

5.3.2 API Structure

5.3.3 Available Endpoints

5.3.4 Data Model

Pydantic schemas ensure input/output validation:

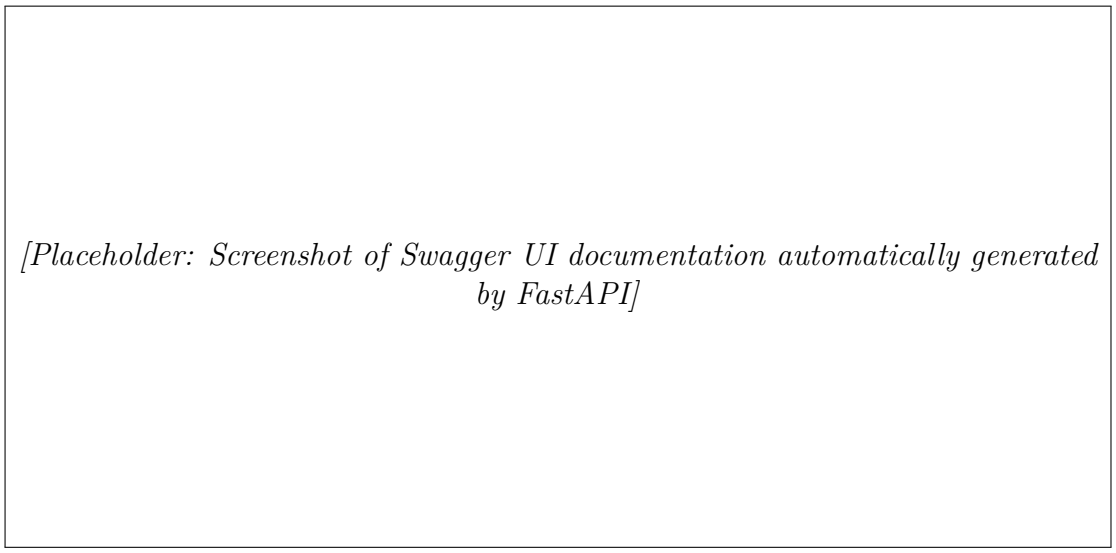


Figure 5.2: API Swagger Documentation

Method	Endpoint	Description
GET	/health	Check API and model status
POST	/predict	Prediction for a single transaction
POST	/predict/batch	Batch predictions for multiple transactions
GET	/stats	API usage statistics
GET	/	General service information

Table 5.1: Fraud Detection API Endpoints

```
class Transaction(BaseModel):
    Time: float
    Amount: float = Field(..., ge=0)
    V1: float
    V2: float
    # ... V3 to V28
    V28: float

class PredictionResponse(BaseModel):
    fraud_probability: float
    is_fraud: bool
    threshold: float
```

5.3.5 Error Handling

The API implements robust error handling:

- **400 Bad Request**: Invalid input data
- **500 Internal Server Error**: Error during prediction
- **503 Service Unavailable**: Model not loaded

5.3.6 Logging and Monitoring

Each prediction is logged in a CSV file for monitoring:

```
timestamp,time,amount,fraud_probability,is_fraud
2025-01-04T10:30:15,12345.0,149.99,0.1234,false
```

5.4 User Interface with Next.js

5.4.1 Technologies Used

The user interface is built with a modern stack:

- **Next.js 15**: React framework with server-side rendering
- **TypeScript**: Static typing for better maintainability
- **Tailwind CSS**: Utility-first CSS framework
- **Lucide React**: Modern icons

5.4.2 Interface Features

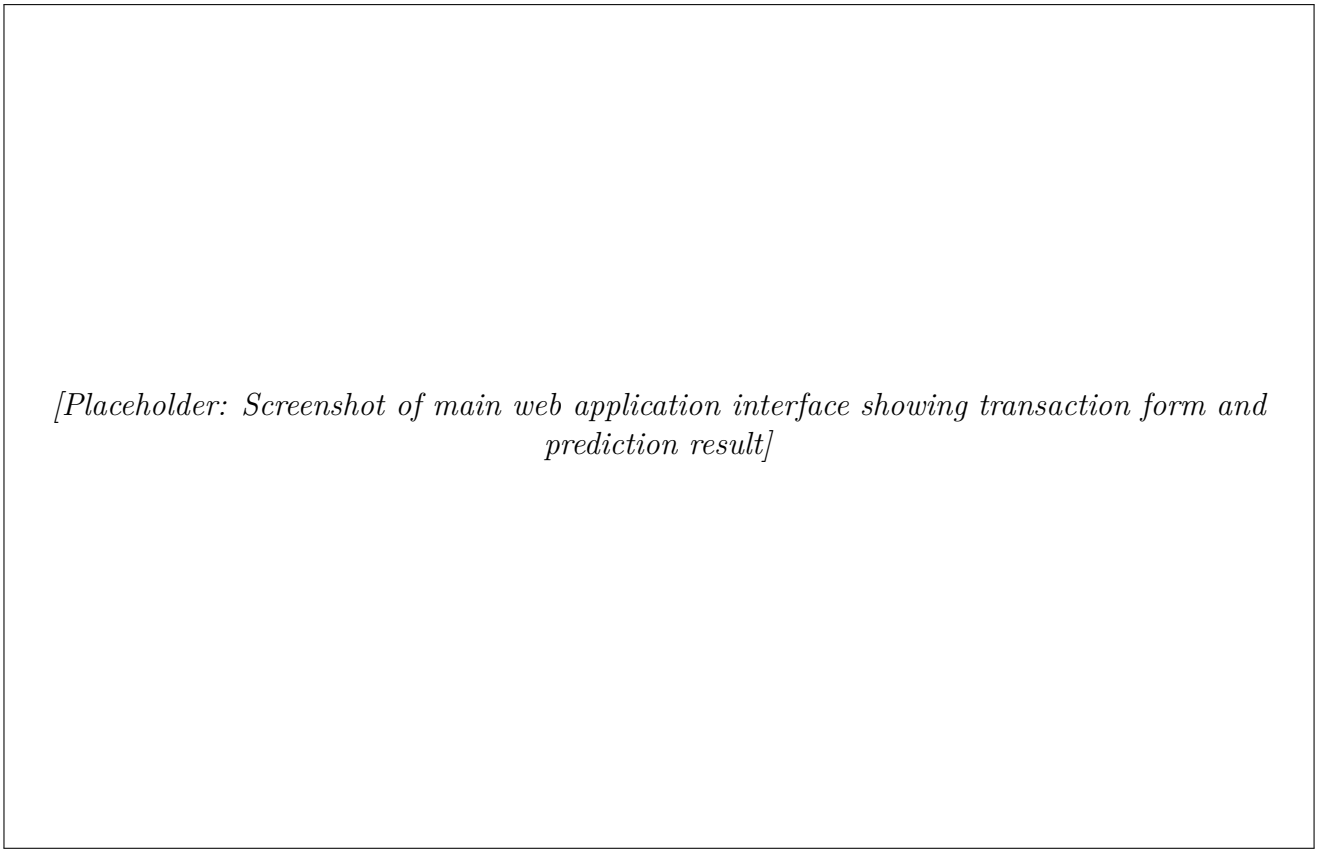


Figure 5.3: Main Interface of Fraud Detection Application

Single Prediction

The main tab allows users to:

- Manually enter transaction characteristics
- Use sample data (normal or fraudulent transaction)
- View result with probability and classification

[Placeholder: Screenshot of transaction input form with Time, Amount, V1-V28 fields]

Figure 5.4: Transaction Input Form

Batch Prediction

The Batch tab allows users to:

- Upload a CSV file containing multiple transactions
- Run predictions on all transactions
- View results in a table
- Download results as CSV

Statistics Dashboard

The Statistics tab displays:

- Total number of requests processed
- Average fraud probability
- Number and rate of detected frauds

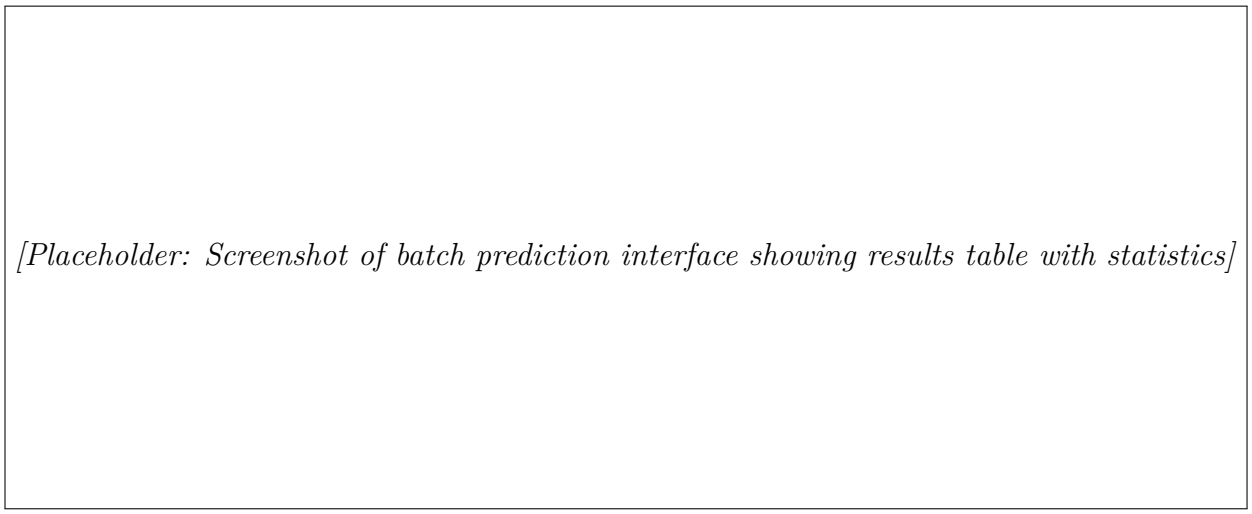


Figure 5.5: Batch Prediction Interface

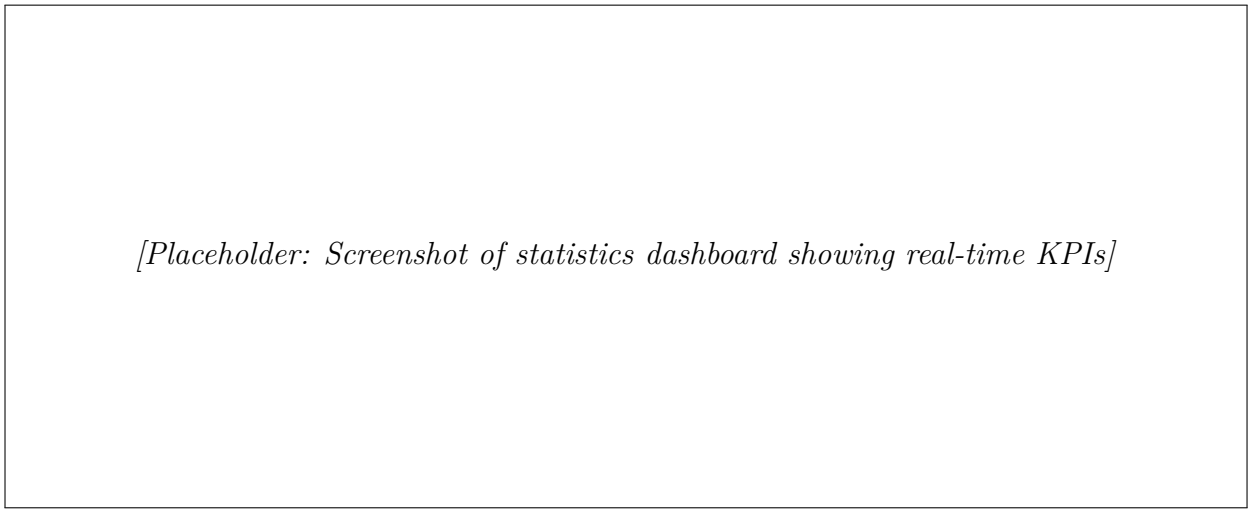


Figure 5.6: Statistics Dashboard

5.4.3 Responsive Design

The interface is fully responsive, adapting to different screen sizes:

5.5 Containerization with Docker

5.5.1 Docker Architecture

The system uses Docker to ensure consistent deployment:

5.5.2 Dockerfile

The Dockerfile uses a multi-stage build to optimize image size:

```
# Stage 1: Builder
FROM python:3.11-slim as builder
```

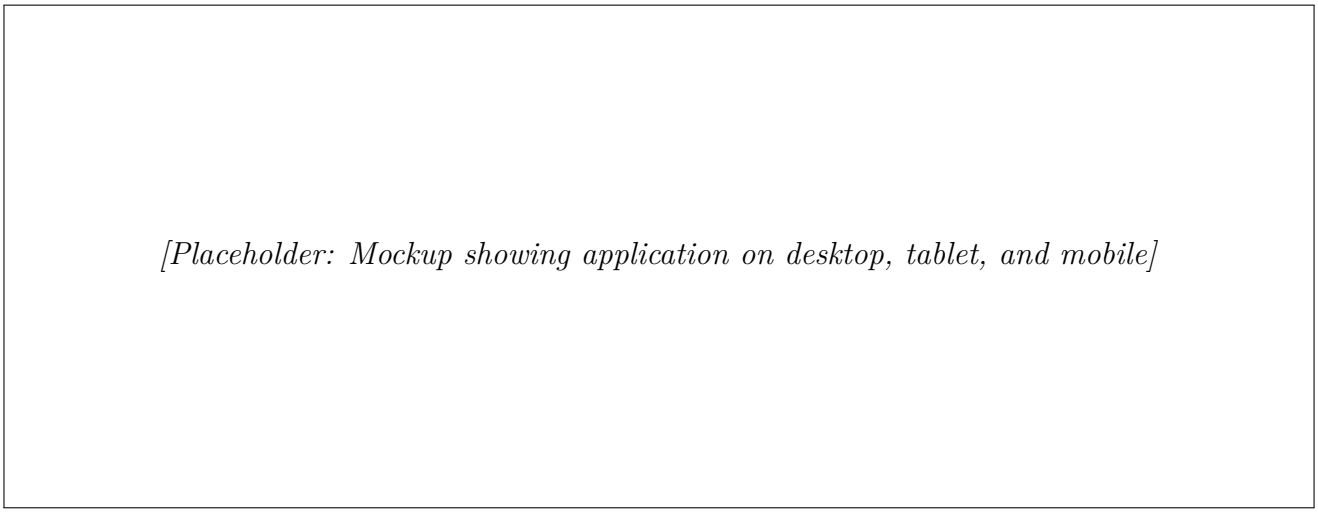


Figure 5.7: Responsive Application Design

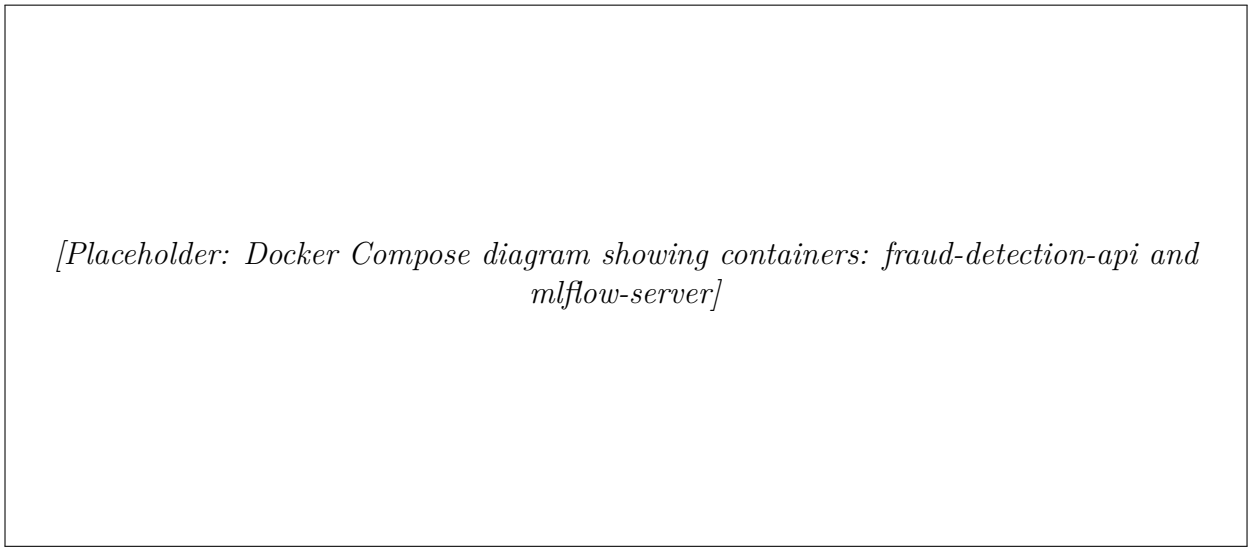


Figure 5.8: Docker Compose Architecture

```
WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt

# Stage 2: Production
FROM python:3.11-slim as production
COPY --from=builder /root/.local /root/.local
COPY api/ ./api/
COPY src/ ./src/
COPY models/ ./models/
COPY params.yaml .
CMD ["uvicorn", "api.main:app",
     "--host", "0.0.0.0", "--port", "8000"]
```

5.5.3 Docker Compose

The `docker-compose.yml` file orchestrates services:

```
services:
  api:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./models:/app/models:ro
      - ./reports:/app/reports
    healthcheck:
      test: ["CMD", "curl", "-f",
             "http://localhost:8000/health"]
      interval: 30s
```

5.5.4 Starting Services

```
# Start the API
docker-compose up -d

# Check logs
docker-compose logs -f

# Stop services
docker-compose down
```

5.6 Cloud Deployment

5.6.1 Deployment Options

The containerized system can be deployed on various cloud platforms:

Platform	**Service**	**Advantages**
Google Cloud	Cloud Run	Serverless, auto-scaling
AWS	ECS / Fargate	AWS ecosystem integration
Azure	Container Instances	Deployment simplicity
Kubernetes	GKE / EKS / AKS	Advanced orchestration

Table 5.2: Cloud deployment options

5.6.2 Example: Google Cloud Run Deployment

```
# Build and push image
gcloud builds submit --tag gcr.io/PROJECT_ID/fraud-api
```

```
# Deploy to Cloud Run
gcloud run deploy fraud-detection-api \
  --image gcr.io/PROJECT_ID/fraud-api \
  --platform managed \
  --region europe-west1 \
  --allow-unauthenticated
```

5.7 Security

5.7.1 Implemented Measures

- **CORS**: Restrictive configuration of allowed origins
- **Validation**: Strict input validation via Pydantic
- **Rate Limiting**: Can be added via middleware
- **HTTPS**: Required in production

5.7.2 Production Recommendations

For production deployment, the following measures are recommended:

1. API authentication (API Keys, OAuth2)
2. Data encryption in transit (TLS)
3. Monitoring and alerting (Prometheus, Grafana)
4. Regular backup of models and data

5.8 Summary

This chapter presented the complete deployment architecture of the fraud detection system. The FastAPI REST API offers performant and well-documented endpoints. The Next.js interface enables intuitive interaction with the system. Docker containerization ensures portability across different cloud environments. This architecture respects MLOps best practices and allows for future system evolution.

Conclusion et Perspectives

Objectives Recap

This project aimed to design and implement a complete MLOps pipeline for credit card fraud detection. The specific objectives were to:

1. Develop a high-performance classification model
2. Establish a reproducible pipeline with DVC
3. Implement experiment tracking with MLflow
4. Deploy the model via a REST API
5. Create a modern user interface
6. Containerize the application with Docker

Summary of Work Accomplished

Data Analysis

We conducted an in-depth analysis of the Kaggle dataset containing 284,807 transactions, of which only 0.172% are fraudulent. This analysis revealed:

- Extreme class imbalance requiring specific techniques
- Discriminative patterns in PCA features
- The importance of variables V17, V14, and V12 for detection

Modeling

The Random Forest Classifier model was trained with the following performance:
These results demonstrate the model's ability to effectively detect frauds while maintaining an acceptable false positive rate.

Metric	Value
ROC-AUC	0.9766
F1-Score	0.8223
Precision	81.82%
Recall	82.65%

Table 5.3: Model Performance Summary

MLOps Infrastructure

The infrastructure put in place includes:

- **DVC**: Data versioning and pipeline orchestration
- **MLflow**: Experiment tracking and model registry
- **FastAPI**: High-performance REST API with automatic documentation
- **Docker**: Containerization for portable deployment

Web Application

The interface developed with Next.js offers:

- Real-time prediction for individual transactions
- Batch processing with CSV import/export
- Usage statistics dashboard
- Modern and responsive design

Main Contributions

The main contributions of this project are:

1. **Complete MLOps Pipeline**: An end-to-end implementation, from data preprocessing to production deployment, following industry best practices.
2. **Imbalance Management**: An effective class weighting strategy integrated into the Random Forest model.
3. **Reproducibility**: Every pipeline step is versioned and can be exactly reproduced.
4. **Documentation**: The code is documented and the API automatically generates its documentation.

Challenges Encountered

During this project, several challenges were encountered:

- **Data Imbalance**: The 578:1 ratio between classes required particular attention when evaluating metrics.
- **Interpretability**: With anonymized features (V1-V28), business interpretation of model decisions is limited.
- **Tool Configuration**: Integrating DVC, MLflow, and Docker required careful configuration.

Future Work and Improvements

Model Improvements

- **Model Ensemble**: Combine multiple algorithms (XGBoost, LightGBM, Neural Networks) to improve performance.
- **SMOTE**: Test synthetic oversampling to handle imbalance.
- **Hyperparameter Tuning**: Use Optuna or Ray Tune for automatic hyperparameter optimization.
- **Sequential Models**: Explore LSTMs to capture temporal dependencies in transaction sequences.

MLOps Improvements

- **CI/CD**: Set up a continuous integration pipeline with GitHub Actions to automate testing and deployment.
- **Production Monitoring**: Implement drift detection (data drift, concept drift) with tools like Evidently AI.
- **A/B Testing**: Infrastructure to compare performance of different model versions in production.
- **Feature Store**: Centralize feature management to ensure consistency between training and inference.

Application Improvements

- **Authentication**: Add an authentication system to secure access.
- **Advanced Visualizations**: Temporal graphs, probability heatmaps, etc.
- **Explainability**: Integrate SHAP or LIME to explain individual predictions.
- **Alerting**: Real-time notifications when high-probability frauds are detected.

Final Conclusion

This project demonstrates the feasibility and effectiveness of an MLOps pipeline for credit card fraud detection. By combining proven machine learning techniques with modern model lifecycle management tools, we have built a system capable of detecting over 82% of frauds with precision exceeding 81%.

The reproducibility, traceability, and deployment ease offered by this architecture constitute a solid foundation for real production deployment. The identified improvement perspectives pave the way for future work that could further enhance system performance and robustness.

This work illustrates how MLOps practices, by industrializing data science processes, enable the transition from prototype to production in a structured and maintainable manner.

Appendix A

Source Code and Configuration

A.1 Project Structure

The complete project structure is as follows:

```
fraud_detection_mlops/
|-- api/
|   +-- main.py                 # FastAPI application
|-- data/
|   |-- raw/
|   |   +-- creditcard.csv      # Raw Kaggle dataset
|   +-- processed/
|       |-- train.csv           # Training split
|       +-- test.csv            # Test split
|-- models/
|   +-- rf_model.pkl            # Trained model
|-- src/
|   |-- data/
|   |   +-- make_dataset.py     # Data preprocessing
|   |-- features/
|   |   +-- build_features.py   # Feature engineering
|   +-- models/
|       |-- train_model.py      # Model training
|       +-- predict_model.py    # Inference utilities
|-- webapp/
|   |-- app/
|   |   |-- layout.tsx          # Root layout
|   |   +-- page.tsx            # Main page
|   |-- components/
|   |   |-- ApiStatus.tsx
|   |   |-- TransactionForm.tsx
|   |   |-- ResultDisplay.tsx
|   |   |-- BatchPrediction.tsx
|   |   +-- StatsPanel.tsx
|   +-- lib/
```

```
|       +-- api.ts              # API client
|-- docker-compose.yml
|-- Dockerfile
|-- dvc.yaml
|-- params.yaml
+-- requirements.txt
```

A.2 Configuration File params.yaml

```
# Configuration file for fraud detection MLOps pipeline

data:
  raw_path: "data/raw/creditcard.csv"
  processed_dir: "data/processed"
  test_size: 0.2
  random_state: 42

model:
  type: "RandomForestClassifier"
  n_estimators: 100
  max_depth: 10
  class_weight: "balanced"
  random_state: 42

train:
  experiment_name: "fraud-detection-mlflow"
  threshold: 0.5
  random_state: 42

paths:
  model_dir: "models"
  figures_dir: "reports/figures"
```

A.3 DVC Pipeline (dvc.yaml)

```
stages:
  make_dataset:
    cmd: python src/data/make_dataset.py
    deps:
      - src/data/make_dataset.py
      - params.yaml
      - data/raw/creditcard.csv
    outs:
      - data/processed/train.csv
      - data/processed/test.csv
```

```
  build_features:
    cmd: python src/features/build_features.py
    deps:
      - src/features/build_features.py
      - params.yaml
      - data/processed/train.csv
      - data/processed/test.csv
    outs:
      - data/processed/train_features.csv
      - data/processed/test_features.csv

  train_model:
    cmd: python src/models/train_model.py
    deps:
      - src/models/train_model.py
      - params.yaml
      - data/processed/train_features.csv
      - data/processed/test_features.csv
    outs:
      - models/rf_model.pkl
```

A.4 Preprocessing Script (make_dataset.py)

```
import pandas as pd
import yaml
from sklearn.model_selection import train_test_split

def main():
    # Load configuration
    with open("params.yaml", 'r') as f:
        params = yaml.safe_load(f)

    # Load raw data
    df = pd.read_csv(params['data']['raw_path'])
    print(f"Data loaded! Shape: {df.shape}")

    # Split features and target
    X = df.drop('Class', axis=1)
    y = df['Class']

    # Train/test split with stratification
    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=params['data']['test_size'],
        random_state=params['data']['random_state'],
        stratify=y
```

```
    )

    # Reconstruct DataFrames
    train_df = pd.concat([X_train, y_train], axis=1)
    test_df = pd.concat([X_test, y_test], axis=1)

    # Save processed data
    train_df.to_csv(
        f"{params['data']['processed_dir']}/train.csv",
        index=False
    )
    test_df.to_csv(
        f"{params['data']['processed_dir']}/test.csv",
        index=False
    )

    print("Data preprocessing completed!")

if __name__ == "__main__":
    main()
```

A.5 Training Script (train_model.py) - Excerpts

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    precision_score, recall_score,
    f1_score, roc_auc_score
)

def train_model():
    # Configure MLflow
    mlflow.set_tracking_uri("sqlite:///mlflow.db")
    mlflow.set_experiment(params['train']['experiment_name'])

    with mlflow.start_run():
        # Initialize model
        model = RandomForestClassifier(
            n_estimators=params['model']['n_estimators'],
            max_depth=params['model']['max_depth'],
            class_weight=params['model']['class_weight'],
            random_state=params['model']['random_state'],
            n_jobs=-1
        )
```

```
        # Log parameters
        mlflow.log_params(params['model'])

        # Train
        model.fit(X_train, y_train)

        # Evaluate
        y_pred = model.predict(X_test)
        y_proba = model.predict_proba(X_test)[:, 1]

        metrics = {
            'precision': precision_score(y_test, y_pred),
            'recall': recall_score(y_test, y_pred),
            'f1': f1_score(y_test, y_pred),
            'roc_auc': roc_auc_score(y_test, y_proba)
        }

        # Log metrics
        mlflow.log_metrics(metrics)

        # Log model
        mlflow.sklearn.log_model(model, "model")

        # Register model
        mlflow.register_model(
            f"runs:/{mlflow.active_run().info.run_id}/model",
            "fraud_detection_rf"
        )

        # Save locally
        joblib.dump(model, "models/rf_model.pkl")

    print("Training completed!")
```

A.6 FastAPI Application (main.py) - Excerpts

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field

app = FastAPI(
    title="Fraud Detection API",
    description="Real-time credit card fraud detection",
    version="1.0.0"
)

class Transaction(BaseModel):
```

```
    Time: float
    Amount: float = Field(..., ge=0)
    V1: float
    # ... V2 to V28
    V28: float

class PredictionResponse(BaseModel):
    fraud_probability: float
    is_fraud: bool
    threshold: float

@app.post("/predict", response_model=PredictionResponse)
async def predict(transaction: Transaction):
    if model is None:
        raise HTTPException(
            status_code=503,
            detail="Model not loaded"
        )

    # Convert to dict and predict
    data = transaction.dict()
    fraud_prob = predict_proba(model, data)
    is_fraud = fraud_prob >= threshold

    return PredictionResponse(
        fraud_probability=fraud_prob,
        is_fraud=is_fraud,
        threshold=threshold
    )
```

A.7 Python Libraries Used

Library	**Version**	**Usage**
scikit-learn	1.8.0	Random Forest, metrics
pandas	2.3.3	Data manipulation
numpy	2.4.0	Numerical operations
mlflow	3.8.0	Experiment tracking
dvc	latest	Data versioning
fastapi	0.127.0	REST API
uvicorn	latest	ASGI server
pydantic	latest	Data validation
joblib	latest	Model serialization

Table A.1: Main Python Libraries

A.8 Useful Commands

A.8.1 Installation

```
# Create virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

A.8.2 Running the Pipeline

```
# Reproduce complete pipeline
dvc repro

# Run specific stage
dvc repro train_model
```

A.8.3 Starting Services

```
# Start API (development)
uvicorn api.main:app --reload

# Start frontend (development)
cd webapp && npm run dev

# Start with Docker
docker-compose up -d
```

A.8.4 MLflow

```
# Launch MLflow interface
mlflow ui

# Access interface at
# http://localhost:5000
```

References

[Amershi et al., 2019] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. (2019). Software engineering for machine learning: A case study. *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300.

[Bhattacharyya et al., 2011] Bhattacharyya, S., Jha, S., Tharakunnel, K., and Westland, J. C. (2011). Data mining for credit card fraud: A comparative study. *Decision Support Systems*, 50(3):602–613.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[Chawla et al., 2002] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357.

[Dal Pozzolo et al., 2015] Dal Pozzolo, A., Caelen, O., Johnson, R. A., and Bontempi, G. (2015). Calibrating probability with undersampling for unbalanced classification. *IEEE Symposium Series on Computational Intelligence*, pages 159–166.

[Databricks, 2024] Databricks (2024). *MLflow: An Open Source Platform for the Machine Learning Lifecycle*.

[Docker Inc., 2024] Docker Inc. (2024). *Docker Documentation*.

[Géron, 2022] Géron, A. (2022). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, 3rd edition.

[He and Garcia, 2009] He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284.

[Iterative, 2024] Iterative (2024). *Data Version Control: Git for Data Scientists & ML Engineers*.

[Machine Learning Group - ULB, 2013] Machine Learning Group - ULB (2013). Credit card fraud detection dataset. `https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud`. Accessed: January 2025.

[McKinney, 2022] McKinney, W. (2022). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 3rd edition.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Ramírez, 2024] Ramírez, S. (2024). *FastAPI: Modern, Fast (high-performance), Web Framework for Building APIs with Python*.

[Sculley et al., 2015] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, 28.

[Tailwind Labs, 2024] Tailwind Labs (2024). *Tailwind CSS: A Utility-First CSS Framework*.

[The Nilson Report, 2023] The Nilson Report (2023). Global card fraud losses. Technical report, Nilson Report.

[Vercel, 2024] Vercel (2024). *Next.js: The React Framework for the Web*.

Abstract

Abstract

Credit card fraud represents losses of several billion dollars annually for the financial industry. This project presents the design and implementation of a complete MLOps (Machine Learning Operations) pipeline for automated fraud detection.
Using the public Kaggle dataset containing 284,807 real transactions, of which only 0.172% are fraudulent, we developed a system capable of identifying suspicious activities with high accuracy. The Random Forest Classifier model, trained with a class weighting strategy to handle the extreme imbalance, achieves the following performance:

- ROC-AUC: 0.9766 (excellent discrimination capability)
- F1-Score: 0.8223 (good precision/recall balance)
- Recall: 82.65% (detection of the majority of frauds)

The implemented MLOps infrastructure ensures end-to-end reproducibility and traceability:

- **DVC** for data versioning and pipeline orchestration
- **MLflow** for experiment tracking and model registry
- **FastAPI** for exposing the model via a high-performance REST API
- **Docker** for containerization and portable deployment

A modern user interface developed with Next.js and Tailwind CSS allows analysts to interact with the system, whether for single or batch predictions.

Keywords: MLOps, Fraud Detection, Machine Learning, Random Forest, Imbalanced Classification, DVC, MLflow, FastAPI, Docker, Next.js