Machine Learning Operations Project

Master's in Data Science and Advanced Analytics

NOVA Information Management School

Universidade Nova de Lisboa

Fraud Detection

US Credit Card Transactions

Report

Group Members

Bruna Simões, 20240491

Daniel Caridade, 20211588

Leonardo Caterina, 20240485

Marco Galão, 20201545

Spring Semester 2024-2025

TABLE OF CONTENT

1. Data Selection and Project Objectives

In today's increasingly connected world, digitalization has transformed how people interact, shop and manage their finances. The COVID-19 pandemic significantly accelerated this shift. With physical movement restricted, both individuals and businesses became more reliant on digital platforms, particularly for financial transactions. Recent studies have highlighted that consumers became more cautious about taking on credit during COVID *(Horvath et al., 2021)*, while others demonstrated a sharp rise in online credit card usage across 44 economies by analyzing MasterCard transactional data *(Alcedo et al., 2025)*.

In response to this challenge, our project focuses on building a machine learning pipeline to detect fraudulent credit card transactions. We use a U.S. based dataset from Kaggle that provides detailed credit card transaction records from 2019 to June 2020 *(Credit Card Transactions Dataset, n.d.)*. This allows us to simulate real fraud detection scenarios and build a robust, modular solution. Our group chose to explore this topic because improving fraud detection systems can significantly reduce financial losses for businesses and prevent damage caused by undetected fraud. By identifying suspicious activity early, companies can block fraudulent transactions before harm is done - protecting both their operations and their customers.

The success of this project will be measured primarily using the macro F1-score, which provides a balanced view of precision and recall across all classes, which is especially important in fraud detection due to the imbalance between legitimate and fraudulent transactions. A higher macro F1-score means better fraud detection with fewer mistakes - allowing businesses to act quickly, protect customers, and maintain trust. Additionally, macro precision, macro recall, and overall accuracy were also evaluated to get a well-rounded view of the model effectiveness.

2. Project Development

To reflect how the project was developed, we structured the report around five sprints. Each sprint corresponds to a stage in the project lifestyle, from exploratory data analysis to monitoring for data drift. This approach not only mirrors the actual sequence in which the work was carried out but also follows the principles from MLOps and Agile methodology, which emphasize modular design and iterative progress.

This architectural design is illustrated in the Kedro Viz representation of the project pipeline (Figure 6). The layout reveals a modular structure, starting with separate branches for data ingestion and converging through preprocessing and modeling stages into a unified output pipeline.

2.1 Sprint 1 – Exploration Data Analysis

The dataset contains over 90.000 credit card transactions, each identified by a unique ID, with no duplicated records. It offers a detailed representation of consumer behavior across 23 features, combining information about the cardholder, the merchant and the transaction itself. These include temporal data, geographic location, demographical attributes and a binary indicator denoting whether the transaction was identified as fraudulent – used as the target variable in our analysis. Notably, the dataset is highly imbalanced, with only 0.6% of transactions labeled as fraudulent. For a detailed breakdown of all features, refer to Table 1.

Aside from the merchant zip code, which contains approximately 15% missing values, the dataset appears to be complete and well-structured. The proportion of missing zip codes is consistent across both fraudulent and non-

fraudulent transactions, indicating the missingness is unlikely to be related to fraud. Moreover, locations with missing zip codes never appear with both missing and non-missing values across different transactions, suggesting that the issue is location-specific rather than random. Notably, a large share of these missing values is concentrated in California, particularly in Los Angeles, hinting at possible regional inconsistencies in data collection.

Most transactions are geographically consistent with locations in the U.S. As shown in Figure 1, states like Texas, California, Pennsylvania and New York account for the largest volume of transactions. Additionally, the proximity between user and merchant coordinates suggests that purchases are typically made locally.

The average transaction amount is around $70, but some purchases go well beyond that, with the highest reaching over $13,500 - suggesting considerable variation in spending and the likely presence of outliers [Figure 2 and Figure 3].

In terms of categories, there are 693 unique merchants, with fraud_Kilback LLC [Figure 4] standing out as the most frequent. The most common transaction type is gas_transport, accounting for more than 9.000 entries across 14 recorded categories [Figure 5]. Occupations are also diverse, with 486 distinct job titles represented in the dataset.

2.2 Sprint 2 – Data Ingestion, Unit Tests and Preprocessing

2.2.1 Data Ingestion

The data ingestion served as the foundation for all subsequent stages of the pipeline, focusing on loading, standardizing and validating the raw credit card transaction data.

The ingestion pipeline began with initial data cleaning, including the removal of duplicates and the conversion of transaction timestamps to a standardized datetime format.

To protect user privacy, personally identifiable information, such as credit card number and cardholder names, were anonymized through cryptographic hashing. This ensured that sensitive attributes remained secure without compromising the ability to link related transactions. At this stage, we also created the cardholder's age, calculated from the date of birth and transaction date.

For effective downstream processing, the dataset was then organized into numerical features, categorical features and the target label. This logical separation supported independent validation, transformation and modeling of each feature group. Schema validation routines were implemented to ensure all expected columns were present with the correct data types and acceptable value ranges, thereby improving data quality and structural consistency.

Hopsworks credentials were securely managed using the *credentials.yml* configuration file. At runtime, the following environment variables were set in the terminal to authenticate access:

- *export FS_API_KEY="..."*
- *export FS_PROJECT_NAME="..."*

Once validation, the cleaned datasets were uploaded to the Hopsworks Feature Store, with descriptive metadata and feature versioning, enabling centralized feature reuse across pipelines, simplified collaboration development and reproducibility.

2.2.2 Data Unit Test

Following the ingestion step, the dataset was split into reference data (80%) and analysis data (20%) using stratified sampling on the target variable to preserve class distribution. The reference data was then used to define a suite of data quality unit tests to establish baselines and constraints that incoming data must conform to.

The unit tests were implemented through a manual test suite developed with Great Expectations, complemented by an automated profiling suite generated from YData Profiling. Together, these two approaches ensured both general data quality and more specific validation aligned with the project's context.

The manual suite consisted of several assertions across the following dimensions:

- **Schema and Structure**: Ensured that all expected columns were present, with data types enforced (e.g., int64 for age, datetime64[ns] for timestamps), unique transaction IDs and completeness of essential fields.
- **Domain Rules**: Numerical variables such as transaction amount were checked against defined bounds (0-1,000,000), along with validation of geographic coordinates and demographic features such as age, restricted to plausible values between 16 and 120. Notably, the test for age failed for a
- **Categorical Constraints**: Allowed values for fields such as gender (['M', 'F', 'U']) and U.S. state abbreviations were explicitly defined and enforced.
- **Target Label Integrity**: The fraud label was validated to contain only acceptable classes (0 or 1), with no nulls.

In addition to automated validation through Great Expectations' checkpoint mechanism, we implemented a few manual assertions – such as verifying column consistency and the uniqueness of transaction IDs – to catch edge cases immediately.

Ultimately, both approaches produced outputs that facilitate review, including:

- CSV and HTML reports, saved in the reporting directory.
- A concise summary of validation results, detailing failed expectations and the percentage of unexpected values, which was logged and reviewed.

It is important to note that we developed unit tests for all critical components, including data preprocessing, data splitting, feature selection, model training, selection and inference. These tests validate the expected behavior of each function and serve as a safeguard against regressions as the pipeline evolves.

2.2.3 Data Preprocessing

To promote reusability throughout the pipeline and maintain a clear separation between training and inference logic, the preprocessing was modularized into three main components: *preprocessing_batch*, *preprocessing_train*, and a centralized *utils* module.

The initial cleaning involved correcting data types, enforcing a minimum age threshold and handling known inconsistencies. Missing values in merchant zip codes were imputed using a hierarchical imputation strategy: the most frequent zip code within each city was used when available, with the most common zip code in the corresponding state used as a fallback. This logic was wrapped into a reusable function that also returns the necessary mappings for consistent application during inference. Additional flags were added to preserve information about missingness and invalid entries, such as improperly formatted zip codes or credit card numbers.

Categorical features were treated according to their cardinality. Low-cardinality variables were encoded using one-hot encoding. To ensure consistency during inference, the fitted encoder object was stored and returned for reuse. High-cardinality variables, on the other hand, were transformed using frequency encoding, which maps each category to its relative frequency in the training data. For deployment, the frequency mappings used during training were also returned for the batch inference pipeline. Temporal features such as hour, weekday, and month were derived from the transaction timestamp and encoded cyclically using sine and cosine functions.

A wide range of domain-informed features were also engineered, including the geographical distance between the cardholder and the merchant (via the Haversine formula[1]), logarithmic transformations of skewed variables, flags for weekend activity and interaction features such as transaction amount per kilometer. Outliers in transaction amounts were handled through capping at the 99th percentile, followed by log transformation. Finally, all numerical features were scaled using a *StandardScaler*, trained on the training set and reapplied during batch inference to ensure consistency.

2.3 Sprint 3 – Feature Selection and Model Development

The third sprint focused on building and integrating the components necessary for the model training, mainly feature selection, model selection and final model training.

2.3.1 Feature Selection

The first stage involved feature selection, where we implemented a flexible architecture capable of supporting multiple strategies. The core goal was to identify a subset of informative features that improved model performance while reducing redundancy and overfitting.

The pipeline includes support for univariate statistical tests using ANOVA, which ranks features based on their individual relationship with the target variable. In parallel, we incorporated embedded methods by computing feature importances from Random Forests [Figure 7]. These importances, learned directly during training, offer insights into the marginal contribution of each variable in a tree-based ensemble context. This test allowed us to understand that, among all variables, the transaction amount emerged as the most influential predictor of fraud.

Additionally, Recursive Feature Elimination was implemented using Random Forest as the base estimator, to recursively eliminate the least important features until a specified number of predictors is retained, allowing for a more exhaustive ranking based on model performance.

The framework also includes a configurable combination mechanism, enabling users to define how selected features from different strategies should be aggregated - either through union, intersection or a weighted heuristic prioritizing importance score. This flexibility allows for experimentation with different feature subsets, depending on downstream modeling constraints or interpretability goals.

In addition, we developed a utility to compute and visualize the correlation matrix, to facilitate the detection of multicollinearity among numerical variables and supporting more informed decisions during feature selection.

[1] The Haversine formula is a mathematical equation used to compute the shortest distance between two points on the surface of a sphere, based on their latitude and longitude coordinates.

2.3.2 Model Development

The model development phase began with a comparison of several baseline classifiers, followed by hyperparameter tuning using Optuna. The selection process was designed to identify a strong performing model in terms of macro F1-score.

An initial comparison was performed across six candidate models: Logistic Regression, Decision Tree, Random Forest, Extra Trees, Gradient Boosting and XGBoost. Each model was trained using default hyperparameters and evaluated on a holdout set. All training and evaluation steps were logged to MLflow, including performance metrics such as accuracy, macro precision, macro recall, and macro F1-score. The best-performing model from this baseline round- based on macro F1 - was selected for further tuning.

Hyperparameter optimization was then carried out using Optuna, with model-specific search spaces defined in the configuration file. The objective function maximized the macro F1-score over a predefined number of trials, using the Tree-structured Parzen Estimator[2] sampler.

Once the best model type and hyperparameters were identified, the final training phase was executed. Rather than manually passing the model and the optimal parameters, the selected model is retrieved directly from the MLflow tracking server using its run tag.

Once loaded, the model is retrained on the full training data and evaluated on both training and test sets, and the computed set of metrics – including accuracy, precision, recall and macro f1-score, are logged to MLflow for tracking and later comparison.

To support explainability, SHAP values are computed for tree-based models that support probability estimates. A SHAP summary plot is generated to highlight the most influential features contributing to the model's predictions, providing an interpretable view of the decision process [Figure 8]. One conclusion we drew from the analysis is that lower transaction amounts tend to correspond to lower SHAP values, indicating that the model predicts these cases as less likely to be fraudulent.

In keeping with a champion-challenger framework, the newly trained model is evaluated against the existing champion model, based on macro F1-score. If the new model outperforms the current champion, it is saved and promoted as the new baseline. Otherwise, the existing model is retained. This safeguard ensures that model updates in production are not only systematic and reproducible but also justified by measurable improvements in predictive performance.

The final model achieved strong performance, with perfect scores on the training set and generalization that remained high on the test set. Specifically, it reached a training accuracy, precision, recall and macro F1-score of 1.00, while on the test set it achieved a macro F1-score of 0.92, with accuracy of 99.8%, precision of 0.96 and macro recall of 0.88.

2.4 Sprint 4 – Model Serving

In Sprint 4, our focus shifted to making the trained model accessible in a production-like environment. To support this, we built a RESTful API using FastAPI, which allowed for asynchronous request handling and automatically generated

[2] The Tree-structured Parzen Estimator is a sequential model-based optimization algorithm used in hyperparameter tuning. Unlike grid or random search, TPE models the probability of achieving high performance based on prior evaluations, allowing it to focus the search in more promising regions of the parameter space.

documentation through Swagger UI. The service was containerized using Docker, based on a lightweight Python 3.11-slim image, and designed to keep model inference code separate from preprocessing logic and training routines.

The Dockerfile follows a multi-stage build process, installing only the necessary dependencies – such as FastAPI, Uvicorn, scikit-learn, XGBoost - listed in a separate *requirement-serving.txt* file. This approach helped reduce the image size and avoid packaging unnecessary development tools. Input and output validation within the API was handled using Pydantic models, which ensured type safety and helped catch malformed requests early.

To align with deployment best practices, the container runs as a non-root user, exposes port 8000 and includes a health check endpoint for service monitoring. A *.dockerignore* file was configured to exclude unnecessary artifacts (raw data, intermediate files, and development resources), keeping only the trained model and required files for inference. This helped minimize the attack surface and streamline deployment.

The deployment architecture supports horizontal scaling and can be easily extended to orchestration environments like Kubernetes. Each API request is validated and processed, returning predictions along with appropriate HTTP status codes and error messages when necessary. Logging is structured, and hooks were added to support future integration with monitoring and observability tools, allowing for performance tracking and potential drift detection over time.

2.5 Sprint 5 – Data Drift

The goal of this sprint was to enhance the reliability and robustness of our fraud detection model by carefully monitoring data drift—meaning significant changes in the distribution of input features over time that could impact model performance. To achieve this, we adopted a layered approach, combining both univariate and multivariate statistical techniques.

Firstly, Population Stability Index (PSI) was applied to key numerical features, especially those involved in creating intermediate features such as transaction amounts, that were relevant for the model evaluated through feature importance. Based on literature, we used PSI thresholds, where values above 0.1 suggest potential drift and values above 0.25 indicate significant drift. To capture changes that might occur in the relationships between features rather than in individual ones, we applied Principal Component Analysis (PCA) on the numerical features and then calculated PSI on the top two principal components, to detect multivariate drift that could otherwise go unnoticed.

Next, we leveraged the Evidently library to perform the Kolmogorov-Smirnov (KS) statistical test on numerical features. Evidently also generated a visual HTML report automatically, which we saved to support data reporting and auditing efforts. For categorical features, we used the NanmyML library, which applies the Jensen-Shannon divergence metric setting a threshold of 0.2 for this metric to flag drift.

Beyond simply identifying data drift, we leveraged the insights from drift analysis to trigger concrete actions within our MLOps pipeline. When drift is detected in key features, an automated hook is triggered to signal the need for model retraining. As part of this retraining process, features exhibiting drift are excluded from the feature selection step, as their instability makes them unreliable until further analysis is conducted. To evaluate the robustness of this approach, we introduced a controlled drift scenario using a test dataset containing only credit card transactions from New York state. This allowed us to simulate a realistic shift in data distribution. Despite the injected drift, the model's performance remained stable, with evaluation metrics indicating less degradation than initially expected.

3. Production Implementation Plan

One significant limitation of the current proof of concept is its reliance on Pandas for data processing. While Pandas is suitable for small to moderately sized datasets, it isn't designed for distributed computing. As a result, memory bottlenecks and degraded performance may arise when scaling to large volumes of transactional data, which is a common scenario in fraud detection, where data availability increases exponentially each day.

To address this scalability concern, we propose migrating the core data processing components to a distributed computing framework such as Apache Spark or Dask. Both are well-suited for handling large datasets efficiently by distributing computations across multiple nodes in a cluster. This shift would enhance both throughput and fault tolerance, providing the performance needed for continuous data processing at scale. We estimate that adapting and optimizing the current pipeline using Spark would require approximately five weeks of focused development.

Once this migration is complete, the entire machine learning operations pipeline - ranging from feature transformation to drift detection - can be orchestrated using tools such as Airflow or Prefect. These orchestration tools will allow us to schedule, monitor, and manage workflows reliably, which combined with real-time alerting mechanisms enables the pipeline to be capable of supporting streaming or near-real-time data processing, this way enabling timely intervention that can prevent financial losses and protect users.

Overall, moving towards distributed processing will significantly enhance the scalability and responsiveness of our system by enabling proactive model monitoring, faster adaptation to data changes and more robust fraud detection in dynamic environments.

4. Environment Setup and Reproducibility

This project was developed using a modular and reproducible machine learning pipeline, leveraging a set of production-ready tools to ensure transparency, maintainability, and ease of experimentation. The implementation was carried out using Python 3.12.4. Below is the list of key packages and their corresponding versions used throughout the implementation. A complete list of all additional packages and their respective versions is provided in the requirements.txt file in the GitHub repository.

- **uv 0.7.16** – to handle reproducible and fast Python dependency resolution when creating virtual environments.
- **Kedro 0.19.14** – for pipeline orchestration, configuration management, and modular project structure.
- **great_expectations 0.17.23** – for generating data unit test to the data.
- **hopsworks 4.2.6** – used for feature store management.
- **mlflow 2.22.1** – for tracking experiments, parameters, and performing model registry and versioning.
- **nannyml 0.13.0 and evidently 0.6.6** – for drift detection with categorical and numerical features and report generation of those drift results.

The entire pipeline is reproducible end-to-end and follows a modular design pattern where each component—from data ingestion and preprocessing to drift analysis and reporting—can be executed independently or in a sequential workflow using Kedro nodes and pipelines. This design supports both development-time debugging and production-time orchestration.

For reproducibility and ease of sharing, the full project is available via a Git repository[3], which includes a README.md file with detailed setup instructions and usage guidelines ensuring that any team member or stakeholder can reproduce the results locally or deploy them into a CI/CD-enabled environment with minimal configuration effort.

5. Conclusion

This project aimed to build a fully automated machine learning pipeline for detecting fraudulent credit card transactions. Using Agile methodology, we structured our work into focused sprints that delivered each stage of the pipeline - from data exploration and processing to model training, evaluation, and data drift evaluation.

What sets our solution apart is its production-readiness and modular architecture. Unlike many fraud detection projects that rely heavily on Jupyter notebooks, ours is built using Kedro for clean orchestration, Docker for containerization, and MLflow for model versioning and experiment tracking. This makes our pipeline scalable, reproducible, and maintainable. In addition to standard MLOps practices, our system includes advanced capabilities such as automatic retraining of the model when data drift is detected, asynchronous API deployment using FastAPI with interactive Swagger documentation, hyperparameter tuning using Optuna, model explainability through SHAP to support transparency with stakeholders, and multimodel experimentation to continuously identify and evaluate the best performing models.

Future improvements would focus on scaling the system for real-time fraud detection using Big Data tools like Spark, enabling it to process high volumes of transactions in real time. Additionally, enhancing model comparison logic between the champion and challenges models would help ensure that deployed models are not only accurate but also fast, fair, and reliable in production environments. By addressing these areas, we move closer to building an end-to-end fraud detection system that is not only accurate and explainable but also scalable, adaptive, and production-ready in real-world, high-volume environments.

6. References

Horvath, A., Kay, B., & Wix, C. (2021). The COVID-19 Shock and Consumer Credit: Evidence from Credit Card Data. Finance and Economics Discussion Series, 2021.0(8), 1–55. https://doi.org/10.17016/feds.2021.008

Alcedo, J., Cavallo, A., Mishra, P., & Spilimbergo, A. (2025). Back to trend: COVID effects on E-commerce in 44 countries. Journal of Macroeconomics, 85, 103682. https://doi.org/10.1016/j.jmacro.2025.103682

Credit Card Transactions Dataset. (n.d.). Retrieved 27 June 2025, from https://www.kaggle.com/datasets/priyamchoksi/credit-card-transactions-dataset

[3] GitHub repository

7. ANNEX

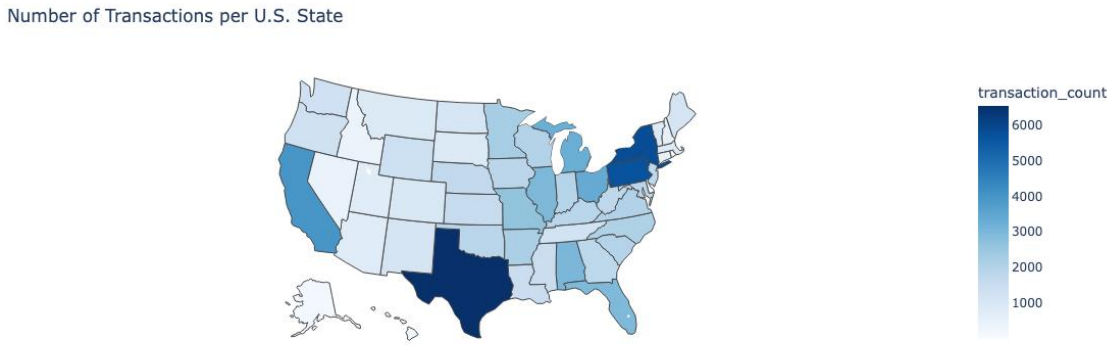

Figure 1 – Number of Transactions per U.S. State

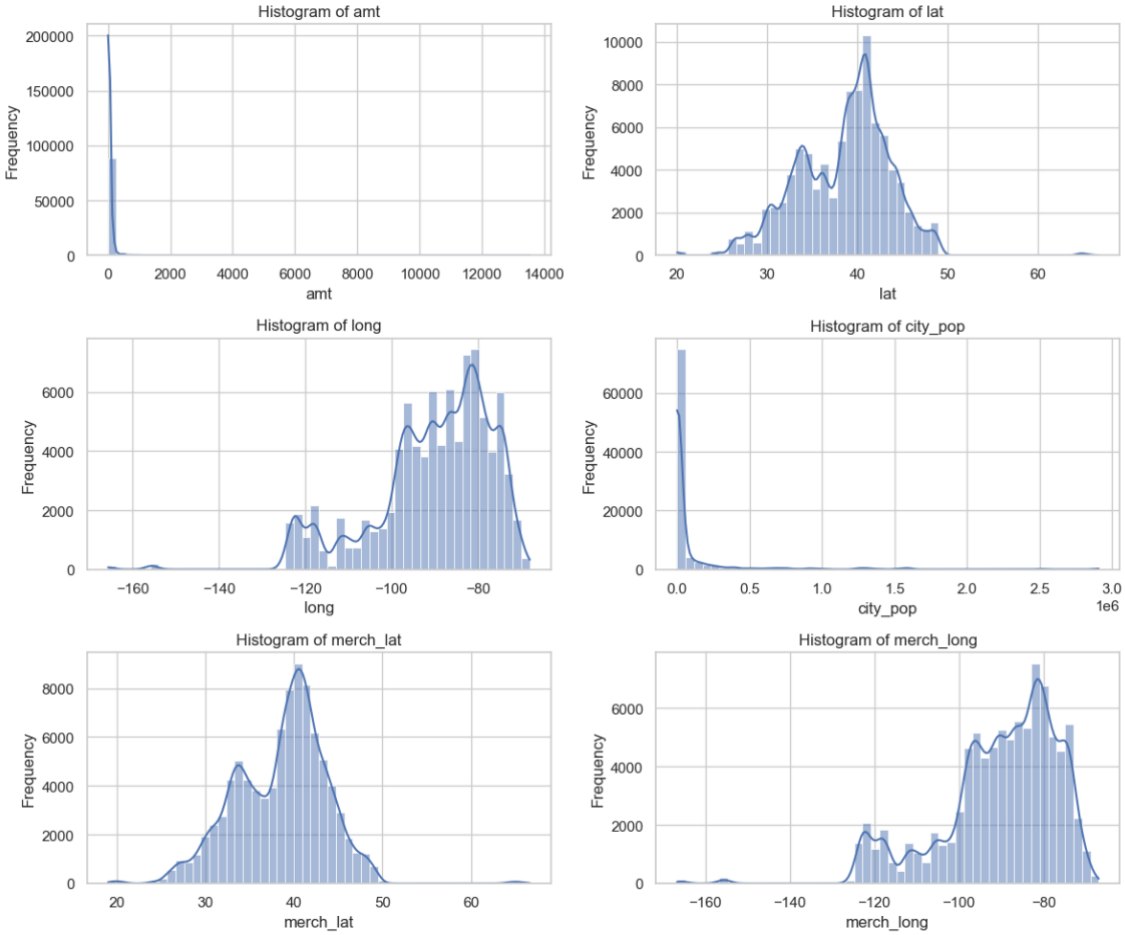

Figure 2 – Histograms of numerical features

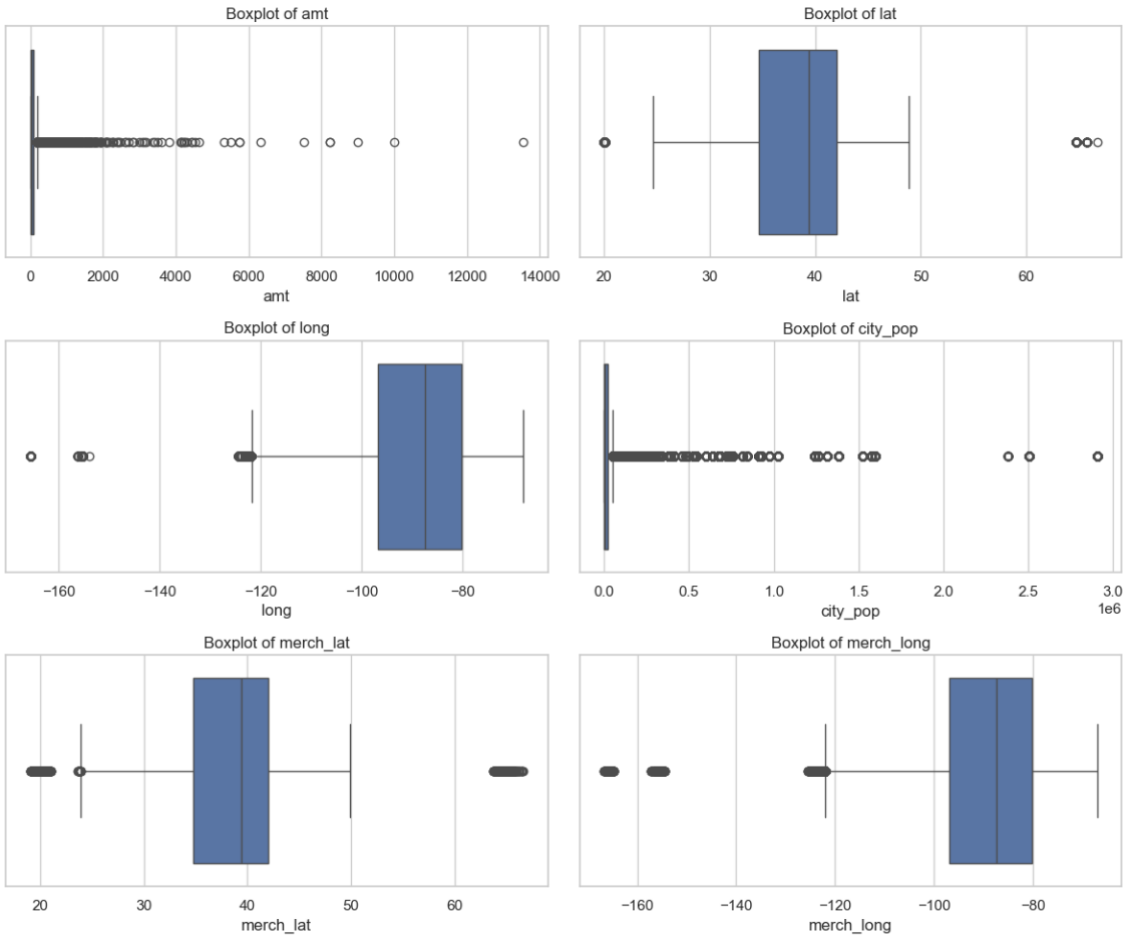

Figure 3 – Boxplots of numerical features

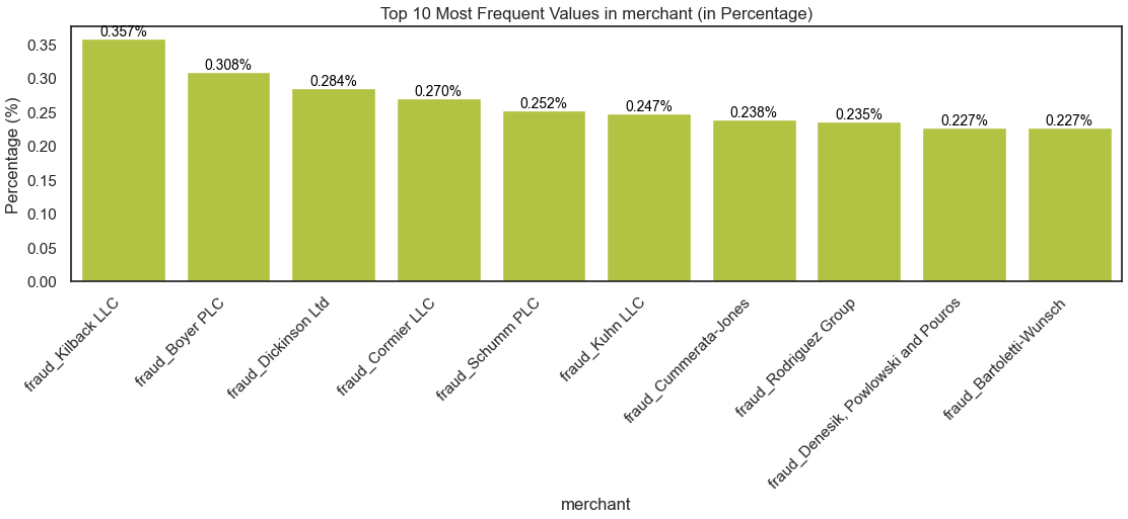

Figure 4 – Distribution of top 20 most common merchants

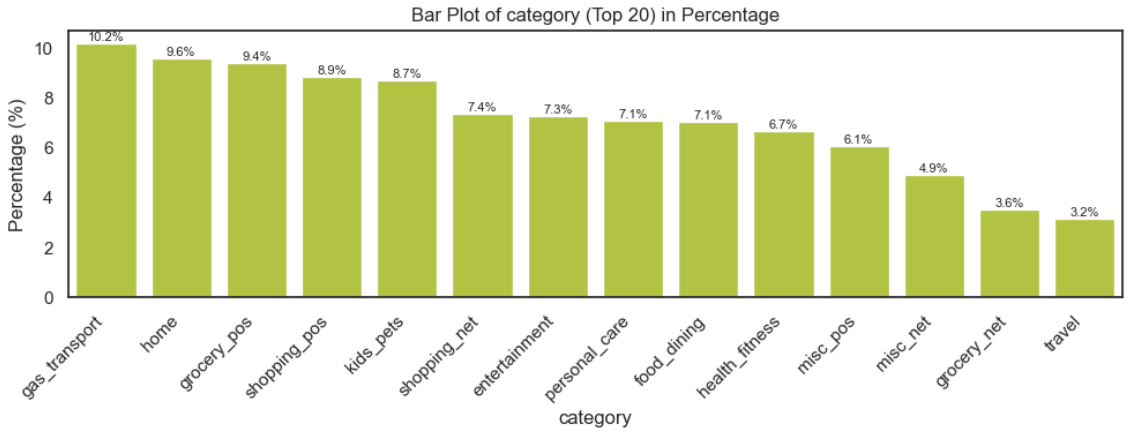

Figure 5 – Distribution of top 20 most common categories

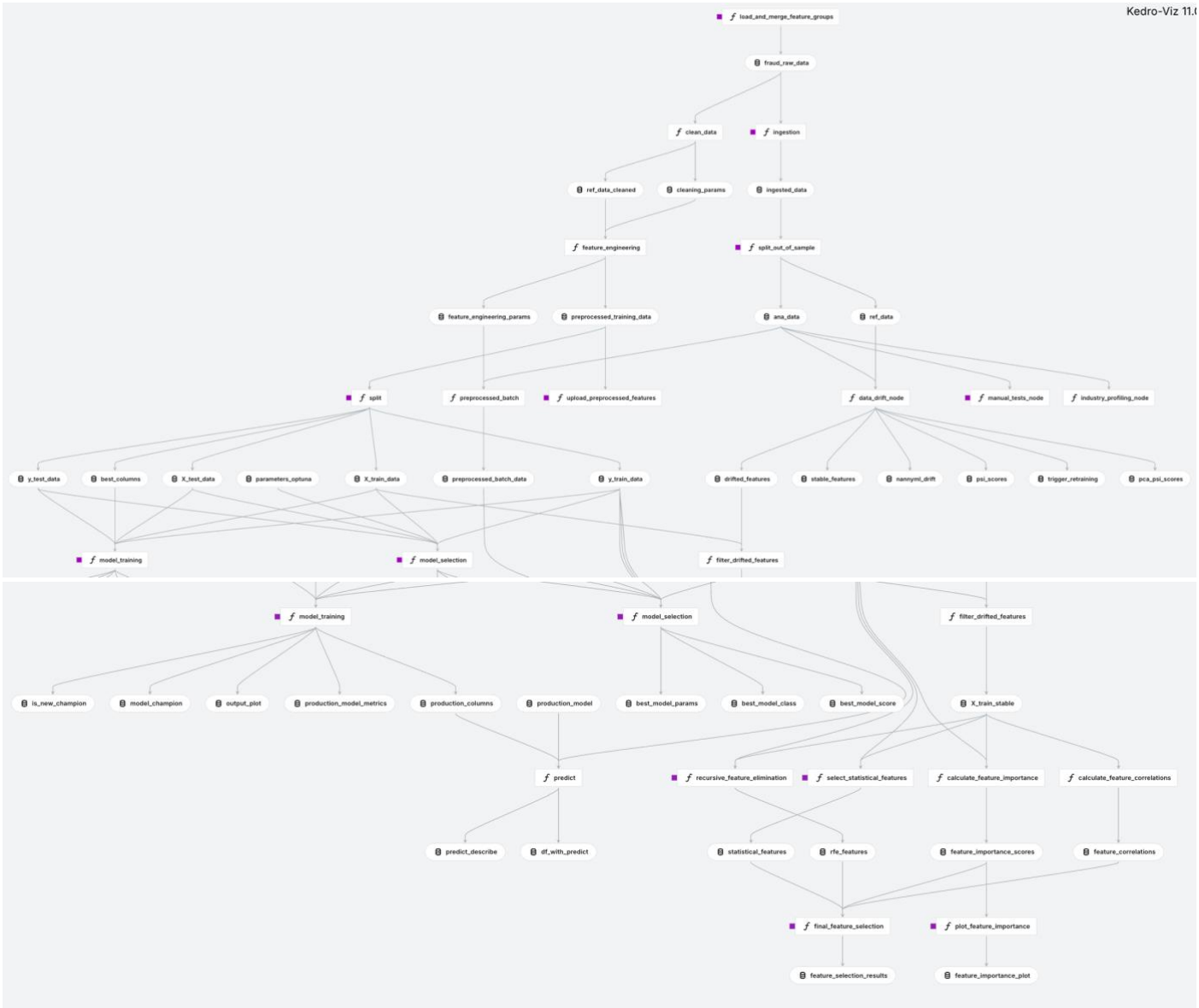

Figure 6 – Kedro viz output

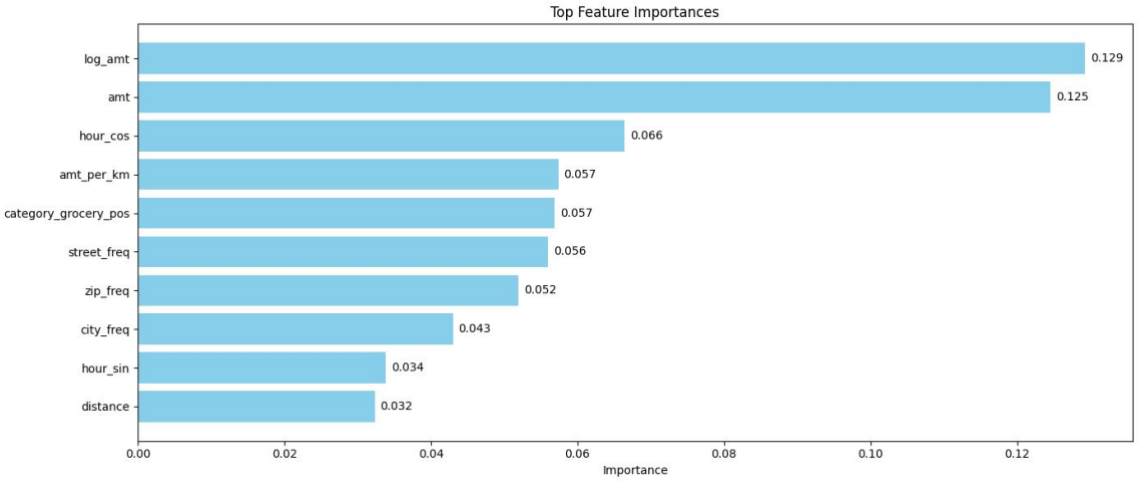

Figure 7 – Top 10 most important features based on Random Forest

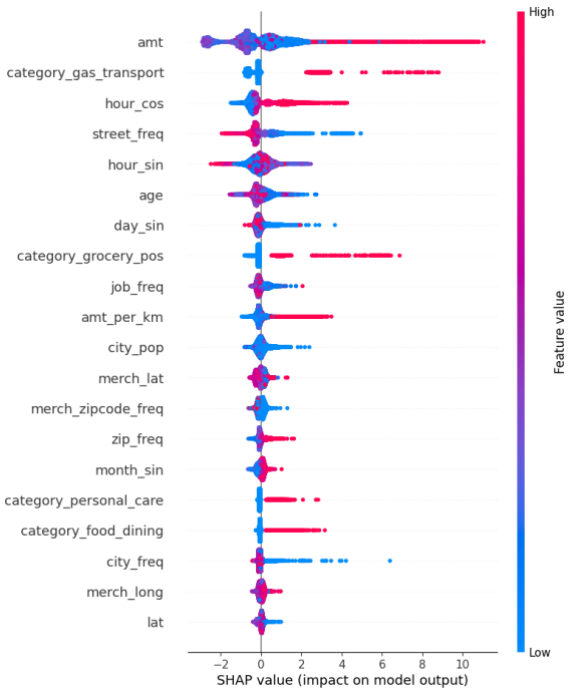

Figure 8 – SHAP Summary Plot for Feature Importance Interpretation

Type	Feature	Description
Temporal	trans_date_trans_time	Timestamp of the transaction
	unix_time	Unix timestamp corresponding to the transaction
Categorical	dob	Date of birth of the cardholder
	cc_num	Credit card number
	first	First name of the cardholder
	last	Last name of the cardholder
	gender	Gender of the cardholder
	street	Street address of the cardholder
	city	City of residence
	state	State of residence
	zip	Zip code of the cardholder's address
	merchant	Name of the merchant
	category	Merchant category
	job	Occupation of the cardholder
	trans_num	Unique transaction identifier
Numerical	merch_zipcode	Zip code of the merchant
	amt	Transaction amount
	lat	Latitude of the cardholder's location
	long	Longitude of the cardholder's location
	city_pop	Population of the cardholder's city
	merch_lat	Latitude of the merchant's location
	merch_long	Longitude of the merchant's location
Target	is_fraud	Binary Indicator

Table 1 - Description of dataset features