NOVA IMS Information Management School

MLOps Project

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

Credit Fraud Detector Model Deployment

Group G
Diogo Pires, 20230534
Majd Al Ajlani, 20230767
Manuel Gonçalves, 20230466
Maria Batrakova, 20230739
Vítor Souto Neves, 20230548

June, 2024

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Table Of Contents

Project link: https://github.com/MajdAlAjlaniiii/Credit_Fraud_Detector

1. Project Overview

Developed as part of our Machine Learning Operations (MLOps) course, this project aims to simulate a real-world process of developing, deploying, and maintaining machine learning (ML) models. The goal is to provide a comprehensive understanding of an entire lifecycle of a machine learning project, from data exploration and preprocessing, to model training, model selection, evaluation, deployment, and management. Through this project, we aim to achieve practical insights and actionable outcomes that can be translated into real-world applications, emphasizing the importance of proper planning and risk mitigation strategies.

1.1. Dataset

For this project, we chose the *"Credit Card Fraud Detection"* dataset from Kaggle [1]. This dataset includes anonymised credit card transactions labeled as either fraudulent or genuine. The dataset contains numerical features obtained through a Principal Component Analysis (PCA). These features, named *V1* through *V28*, are the principal components, while *"Time"* and *"Amount"* are the only features not transformed by PCA. *"Time"* represents the seconds elapsed between each transaction and the first transaction in the dataset, and *"Amount"* indicates the transaction amount. The response variable, *"Class"*, indicates whether a transaction is fraudulent (1) or genuine (0).

1.2. Business Problem

Fraud detection is a critical challenge faced by financial institutions, where the goal is to identify and prevent fraudulent transactions while at the same time minimizing the impact on legitimate transactions. The inherent class imbalance in fraud detection cases (which reveals itself in our case), where fraudulent transactions (positive cases) are significantly outnumbered by genuine transactions (negative cases), poses a unique challenge for machine learning models. We chose the **Weighted Recall** as the target metric since we want to capture as many fraudulent transactions as possible while considering the balance between different classes. By focusing on this metric, we ensure that the model's performance is evaluated in a way that accounts for both the minority (fraudulent) and majority (non-fraudulent) classes. This approach allows us to build a more robust fraud detection system that effectively minimizes false negatives, thereby safeguarding the institution against fraudulent activities. By balancing the model appropriately, we can manage the trade-off with false positives, ensuring a smooth user experience and maintaining trust in the system.

1.3. List of packages and versions

The full list of packages used for this project, including their version, can be found in the *requirements.txt* file of the project, accessed using the provided GitHub link.

2. Project planning

We divided this report into five Sprints. Each Sprint took, on average, about one week, with some requiring more time and others, involving simpler processes, taking less time.

2.1. Sprint One: Data Exploration and Preprocessing & Model Training

As mentioned, we used the *"Credit Card Fraud Detection"* dataset from Kaggle. Because the goal of this project was to implement a fully functional MLOps pipeline, for the model development phase, we took inspiration from a notebook [2] posted on Kaggle that used this dataset to test and implement several methods of data exploration and preprocessing, as well as model testing and fine-tuning. Our main goal was to direct our focus on the subsequent stages of the MLOps project, specifically the deployment and monitoring phases, and code modularization.

We're dealing with an incredibly unbalanced dataset, which is expected as mentioned in the Business Problem section. To tackle this issue, we tested both random undersampling and SMOTE (oversampling technique) to have an equal distribution between fraudulent and non-fraudulent cases (Fig. 1). This resulted in a huge loss of data. Were this a real-life ML project we would have to deal with this another way to avoid such data loss, but given the focus of this project, we still decided to keep this approach, and continued using the subset with random undersampling.

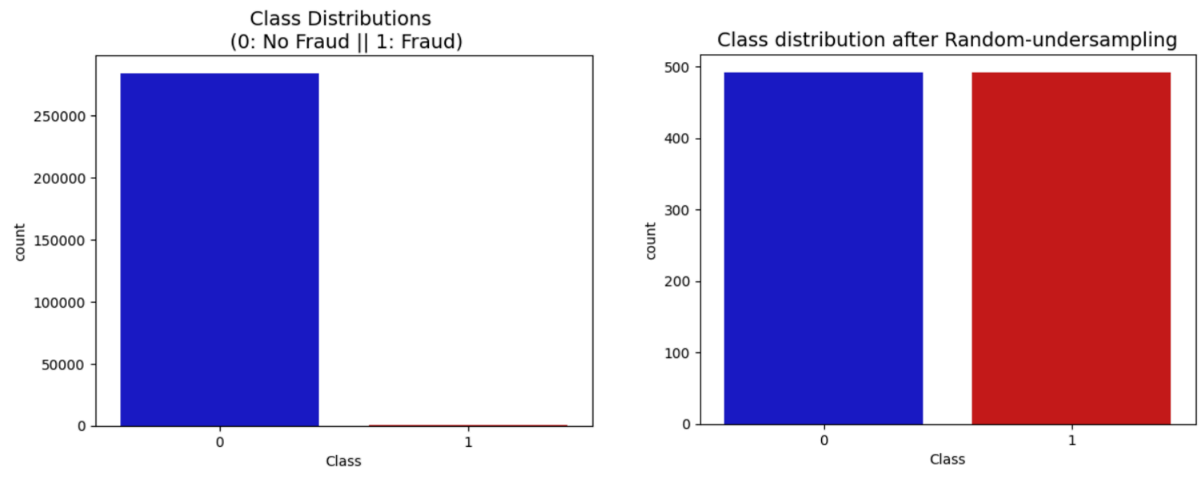

Fig. 1 - Class distributions: original dataset (left); subset after random undersampling (right).

Using normalization techniques, we scaled the *"Amount"* and *"Time"* features using the Standard Scaler and Robust Scaler, respectively. Correlation tests were performed, and we removed outliers of particular features (*V14*, *V12*, and *V10*) using the interquartile range (IQR) strategy. After preprocessing, we split our data using stratified k-fold cross-validation and began model testing, using four different models: **Logistic Regression** (LR); **K-Nearest Neighbors** (KNN); **Support Vector Classifier** (SVC); and **Decision Trees** (DT). Hyperparameter tuning was implemented as well, using Grid Search to test multiple hyperparameters on all the classifiers tested.

2.2. Sprint Two: Kedro Pipeline Setup and MLFlow Integration

We set up our MLOps pipelines using **Kedro** and integrated it with **MLFlow** for experiment tracking. Using Kedro's modular structure, we developed well-organized pipelines with multiple nodes (Fig. 2). We designed a pipeline separated from the main one, that is responsible for data loading and preprocessing. The **Load and Preprocess Data Node**, as the name implies, takes the original dataset and transforms the data into preprocessed data using the methods described in Sprint One. This data is then uploaded to our feature store in Hopsworks, facilitating data management. It's also in this pipeline that we perform the **Data Unit** and **Data Drift** tests.

The main pipeline starts with the **Download Feature Store** node, which is responsible for fetching our preprocessed data from the feature store. Next, the **Split Data Node** splits the data into training and testing sets (*X Train, Y Train, X Test, Y Test*). The training sets are used to train and fine-tune the models, outputting a set of **Trained Models**, where we can select our **Champion Model**. This champion model is subsequently used to **Calculate and Plot SHAP values** for explainability and interpretability purposes. Finally, the testing sets are used to evaluate our models, which outputs a set of **Evaluation Reports**, providing metrics and insights into model performance.

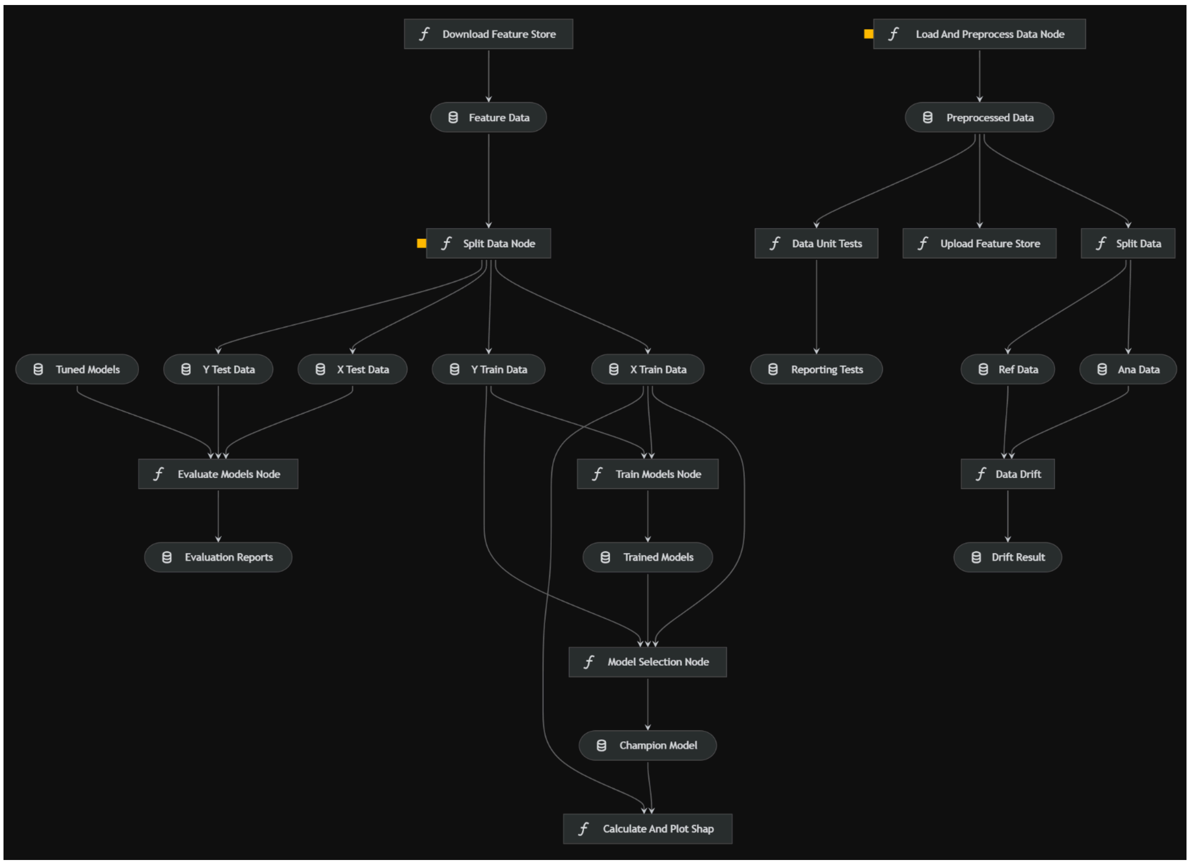

Fig. 2 - Kedro pipeline's structure.

During the training and tuning phases, each experiment's parameters, metrics, and artifacts were logged using **MLFlow**. This allowed us to track the performance of all our different models and configurations systematically. MLFlow's model registry was also used to store and manage the versions of our models, ensuring that we could easily retrieve and deploy the best-performing model.

2.3. Sprint Three: Feature Store Implementation

As mentioned in Sprint Two, we have a pipeline for uploading and downloading features from a feature store. For Sprint Three, we focused on the implementation of a feature store using **Hopsworks**, a modular data platform that facilitates collaboration by providing a governed platform for developing, managing, and sharing ML assets such as features, models, training data, batch scoring data, and logs [3].

We began by setting up the environment and connecting to the **Hopsworks feature store** using credentials from environment variables. We prepared the data by formatting the DataFrame, resetting its index if necessary, and sanitizing column names. Next, we created or retrieved feature groups in the feature store, defining their names, versions, primary keys, and descriptions. The preprocessed data was then inserted into these feature groups.

Finally, we split the DataFrame into features and target, defined metadata for both, and uploaded them separately to the feature store. This approach ensured efficient and collaborative management of our machine learning features and targets, establishing a strong foundation for future development and analysis.

2.4. Sprint Four: Unit Tests to Functions and Data Unit Tests

Four tests were conducted for this Sprint. First, a unit test was developed for the *load_and_preprocess_data* function using ***Pytest***. The test utilizes a *sample_data* fixture to create a mock DataFrame resembling the structure of the credit card transaction data. The test function saves this data to a temporary CSV file and passes it to the preprocessing function. The test verifies several aspects of the preprocessing:

- Removal of original *"Time"* and *"Amount*" columns;
- Addition of *"scaled_time"* and *"scaled_amount"* columns;
- Correct scaling of time and amount values using *RobustScaler*;
- Preservation of other feature columns (*V1* to *V28*);
- Retention of the "Class" column intact.

For the second test, *test_shuffle_and_undersample*, evaluates the effectiveness of the undersampling process in balancing class distribution, verifies that changes have been made to the original DataFrame, and confirms that the resulting dataset size matches expectations.

The third function, *test_remove_outliers*, assesses the *remove_outliers* function. It introduces artificial outliers into the dataset, then verifies their successful removal using the IQR method, particularly focusing on the *V14* column for fraud cases.

The final test, *test_preprocess_data*, validates the *preprocess_data* function, which integrates multiple preprocessing steps. It checks for appropriate class balancing, verifies the expected record count, and ensures effective outlier removal from specified columns (*V14, V12, and V10*), also using the IQR method.

Additionally, data unit tests were implemented using **Great Expectations** to validate the integrity and correctness of the data post-preprocessing. These tests include:

- Ensuring *"scaled_time"* and *"scaled_amount"* columns have values within expected ranges;
- Verifying that the *"Class"* column contains only the values 0 and 1;
- Confirming non-null values in critical columns *("scaled_time", "scaled_amount", "Class"*, and all *V columns*);
- Checking that all values in the *V columns* are within the expected range.

The *get_validation_results* function extracts validation results, including success status, expectation types, and observed values, ensuring comprehensive data validation. A *SimpleCheckpoint* is run to validate the data against the defined expectations, and assertions are made to ensure the data types of *“scaled_time”, “scaled_amount”*, and *“Class”* are as expected.

By combining these unit tests and data unit tests, this Sprint ensures both the functionality and integrity of the data processing pipeline, providing robust validation and confidence in the preprocessing steps.

2.5. Sprint Five: Model Monitoring and Explainability

We implemented **data drift tests** to monitor and detect any significant changes in the data distribution over time, this is essential in any MLOps pipeline to ensure the model's performance remains consistent and reliable. Our approach for analyzing data drift was employed using the resources from the library ***NannyML***. The first approach was through **Univariate Drift Detection**, employing different continuous and categorical methods according to the nature of the features. For this approach, we display an example using **PSI** (Fig. 3) where through the introduction of artificial noise the features "*scaled_time*" and "*scaled_amount*" undergo significant population change as evidenced by their PSI value of over 0.2. Then, using a **Multivariate Drift Detection** method with Domain Classifier, to analyze possible changes in feature relationships over time.

SHAP values were calculated and a summary plot was generated for the “champion model” (Fig. 3). SHAP (SHapley Additive exPlanations) values are a set of values assigned to each feature in ML models, that provide insights into the contribution of each feature to the model's prediction for a specific instance [4]. With SHAP, we are able to explain our model, and understand its predictions based on the features it deems important.

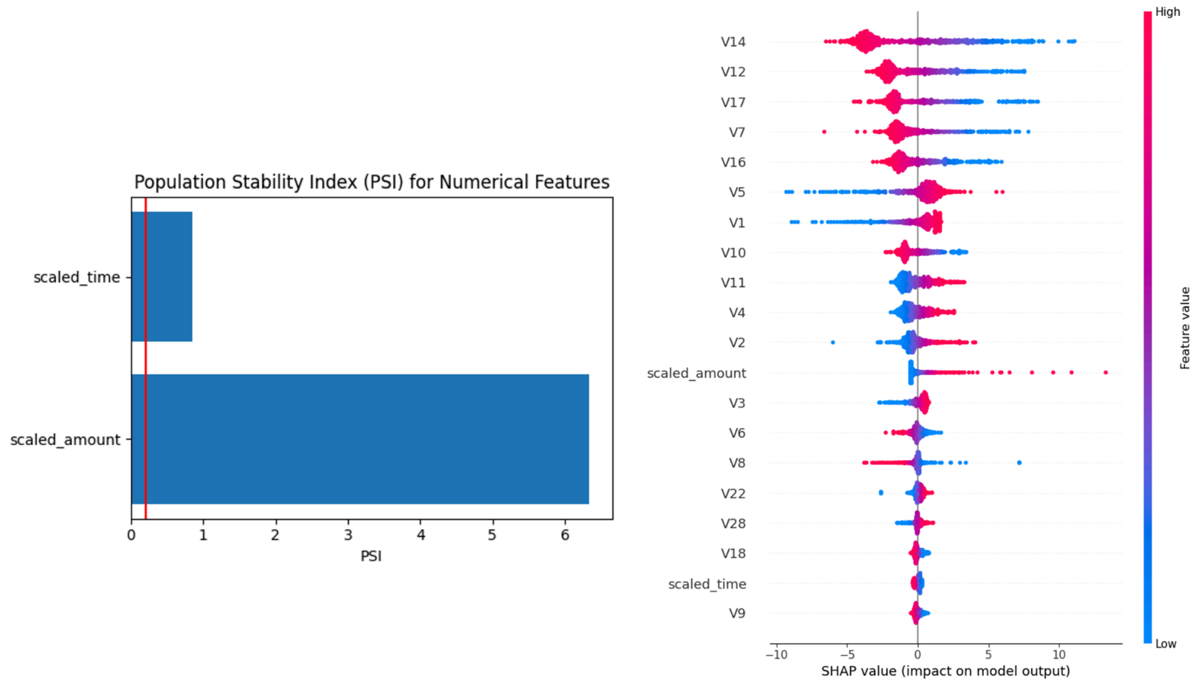

Fig. 3 - PSI for numerical features (left); SHAP value of each feature's impact on the model's prediction (right).

With this graph we can see that the five most important features are *V14*, *V12*, *V17*, *V7*, and *V16*, where high values (red) increase the model's output significantly, while low values (blue) decrease it.

Streamlit is an open-source Python framework that allows the creation of interactive web applications, quickly and easily. It was designed to help data scientists turn data scripts into shareable web apps [5]. We decided to use Streamlit to create a visually appealing and interactive representation of our entire workflow, including raw and preprocessed data, pipeline visualizations (using *Kedro Viz*), and to provide an interface for running the pipelines. This approach enhanced the accessibility and usability of our project, making it easier to understand and interact with the various components.

3. Results and Conclusions

With this project we managed to successfully implement a functional MLOps pipeline, covering all stages from model development, to deployment and management. The Kedro framework was instrumental in organizing our pipelines and modularizing the code, which enhanced the maintainability and scalability of the project. Hopsworks allowed us to create a feature store for storing and retrieving our preprocessed features. With MLFlow, we tracked the execution of our pipelines, conducted grid search experiments for hyperparameter tuning, and compared different models using our chosen evaluation metrics. The inclusion of SHAP values, data drift tests, and data unit tests further enhanced the explainability, reliability, and integrity of our models. Finally, Streamlit allowed us to develop an interactive user-friendly application to visualize our models results and performance metrics.

To conclude, the combination of all these tools significantly improved our workflow efficiency and model performance monitoring.

4. Improvements and Future work

Given the nature of this project, as mentioned, our primary focus was on the deployment and monitoring phases of the MLOps cycle. However, a real-life project would require more extensive attention to the development phase, conducting a more thorough data exploration, testing and implementing a wider range of preprocessing techniques, and experimenting with more classification models, as well as applying strategies to prevent and avoid data loss, overfitting, and data leakage.

There are also a few areas for improvement and expansion to enhance the robustness and scalability of MLOps pipelines, such as Containerization and Orchestration with Docker and Kubernetes, automated testing and CI/CD**.** In the case of dynamic data (e.g. streaming or live data), we could also improve monitoring and logging to track performance in real-time, and in the case of Big Data, developing scalable infrastructure using tools like Hadoop or Spark for efficient data handling.

5. References

[1] “Credit Card Fraud Detection”. (2018). Retrieved May 25, 2024 from https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

[2] Janio Martinez Bachmann. “Credit Fraud || Dealing with Imbalanced Datasets”. (2019). Retrieved May 25, 2024 from https://www.kaggle.com/code/janiobachmann/credit-fraud-dealing-with-imbalanced-datasets

[3] Hopsworks Documentation. Retrieved June 15, 2024 from https://docs.hopsworks.ai/latest/

[4] An introduction to explainable AI with Shapley values. Retrieved June 25, 2024 from https://shap.readthedocs.io/en/latest/example_notebooks/overviews/An%20introduction%20to%20explainable%20AI%20with%20Shapley%20values.html

[5] Streamlit documentation. Retrieved June 27, 2024 from https://docs.streamlit.io