NOVA IMS Information Management School

2024/2025

MACHINE LEARNING OPERATIONS

LOAN APPROVAL PREDICTION

MASTER IN DATA SCIENCE AND ADVANCED ANALYTICS

GROUP

- AFONSO LOPES, 20211540
- LEONOR MIRA, 20240658
- MARTIM TAVARES, 20240508
- RITA MATOS, 20211642
- RITA PALMA, 20240661

GITHUB LINK REPOSITORY

Table of Contents

1. Introduction

In this project, we explore the application of machine learning to the problem of loan approval prediction - a task of significant importance in the financial sector [1]. Deciding whether to approve or reject a loan is not only a matter of business efficiency but also one of regulatory compliance and customer trust. Historically, these decisions have been made manually by loan officers, but with the rising number of applications and increasing demand for faster, more digital services, there is a strong incentive to automate this process [2]. Machine learning offers a promising solution by enabling faster, more consistent, and potentially fairer decision-making.

Our goal is to simulate how such a solution could be developed and deployed in a production-like environment. Automating loan approval decisions can help financial institutions streamline operations, reduce turnaround times, and minimize human biases. At the same time, it enables them to scale their services more effectively without compromising decision quality.

This project is not only about building a predictive model to determine whether a loan should be approved, but also about establishing a complete MLOps workflow to support its deployment and ongoing performance. We place emphasis on experimentation tracking, version control, interpretability, and monitoring by integrating tools like MLflow for experiment management, SHAP for model explainability, and data validation and drift detection mechanisms to ensure robustness after deployment.

2. Methodology

Throughout the project, we adopted a structured, collaborative approach based on the Agile methodology, organizing our efforts into sprints. We held planning sessions to prioritize data pipeline implementation and assigned responsibilities for parallel progress.

As each pipeline was completed and validated, we built a functional data workflow, moving to a testing phase where tasks were divided among team members to ensure robustness and accuracy of various components.

Our process included thorough data cleaning and storing the cleaned data in a feature store for easy access. We performed data segmentation and feature extraction to enhance model performance, training multiple models to select the best one through rigorous testing.

To maintain long-term performance, we implemented data drift detection and monitoring. We used MLflow [3] to track experiments and manage models, with Python packages streamlining the workflow from preparation to deployment.

Each sprint began with meetings to set goals and roles, followed by daily stand-ups for communication. At the end of each sprint, we held reviews and retrospectives to assess progress and tackle challenges, promoting flexibility, teamwork, and continuous improvement.

2.1. Initial Data Exploration and Cleaning Strategy

The initial step in our project involved conducting an Exploratory Data Analysis (EDA) using a dedicated notebook. This phase allowed us to become familiar with the dataset's structure, distribution of variables, and potential data quality issues. During the exploration, we identified

several inconsistencies that required attention, including extra whitespace in column names and categorical values, inappropriate data types for certain features (*loan_term*, *no_of_dependents*), and the need to create interpretable binary variables for some fields. These findings directly informed the design of our first modular pipeline, *data_cleaning*, which addressed these issues systematically. The cleaning pipeline included steps such as trimming whitespace, standardizing categorical columns, mapping human-readable categories to numerical values, and engineering new binary features - specifically, *loan_approved* as a representation of the target variable *loan_status*, and *graduate* derived from the *education* field. In addition, the values in the *self_employed* column were converted from the string categories "Yes" and "No" to binary format (1 and 0), making them suitable for model training. This ensured that the dataset was in a consistent and machine-learning-friendly format before proceeding with the other pipelines.

2.2. Data Validation Using Great Expectations

To ensure high data quality and consistency across our pipeline, we adopted Great Expectations [4] to implement a comprehensive suite of data unit tests. This decision followed our initial data exploration phase and was aimed at reinforcing the reliability of the dataset throughout each stage of transformation and modeling.

Using Great Expectations, we defined rules to validate categorical variables such as *graduate* and *self_employed*, ensuring that their values matched a predefined set of valid categories. Additionally, for the *loan_approved* field, we enforced an extra validation to ensure that this variable could not contain null values. As the target variable of our problem, its preservation is essential, and the presence of missing values would compromise data integrity as well as the reliability of the model's outputs. In the case of continuous features, for example *income_annum* and *loan_amount*, we applied two key strategies. First, we constrained the value ranges based on the 1st and 99th percentiles to exclude extreme outliers while retaining realistic data points. Second, we introduced distribution-based validations using mean ± 1.5 times standard deviations to flag any statistically abnormal values. This combination helped strike a balance between capturing edge cases and filtering noise from the dataset. We also implemented data type validations to maintain schema consistency. Categorical and binary fields were expected to be integers, whereas monetary values were allowed to be either floats or integers.

These validations were applied not only during the initial stage but were also re-run after the feature engineering process to validate any newly created variables.

2.3. Data Ingestion and Feature Store Integration with Hopsworks

During the data ingestion phase, the raw dataset was initially preprocessed by converting the datetime column into a standardized timestamp format and removing any duplicate records to ensure consistency. Features were then organized into three main categories - numerical, categorical, and target - to support structured and modular processing in later stages of the pipeline. Basic checks were applied during ingestion to prevent clearly invalid values or features from being stored in the feature store. More comprehensive validations were handled separately through data unit tests in Section 2.2.

Once cleaned and structured, the features were ingested into the Hopsworks Feature Store [5], where each group was mapped to a dedicated feature group. For every group, *loan_id* was designated as the primary key, while the *datetime* field served as the event time. Metadata including feature descriptions, value ranges, and units were registered alongside the features to enhance discoverability and context for downstream users.

After ingestion, Hopsworks automatically generated summary statistics - including minimum and maximum values, mean, and standard deviation - as well as visualizations such as histograms for continuous variables and bar charts for categorical ones. A data preview was also made available through the platform. These built-in metrics, along with custom profiling, enable ongoing monitoring for data drift and anomalies, ensuring that the stored features remain trustworthy over time.

2.4. Reference-Analysis Data Split

The dataset was randomly split into two groups: 80% as reference data and 20% as analysis data. The reference data serves as the primary dataset and is further divided into training and validation sets, while the analysis data is reserved for additional analysis and testing. A fixed random state was used to ensure the split is reproducible.

2.5. Data Preprocessing for Training and Batch Inference

The preprocessing phase involved a series of steps, including outlier handling, feature engineering and data scaling. Missing values were not addressed because the original dataset contained no missing entries, as confirmed during the data exploration phase.

First, we identified outliers in three variables, *residential_assets_value, commercial_assets_value* and *bank_asset_value,* using the Interquartile Range (IQR). All points that fell outside the acceptable range were removed.

Several new features were created to capture financial characteristics and risk profiles, enhancing the dataset's value. Table 1 shows all the newly engineered features and their respective descriptions.

Additionally, the column *datetime* was dropped since it was only created for a previous step, and it wouldn't provide any predictive power to the classification models.

Finally, all numeric features were standardized using the Standard Scaler and categorical features were one hot encoded. The resulting scaler and encoder were saved as pickle files for later use.

The same preprocessing steps were applied to the batch dataset, using the statistics derived exclusively from the training data.

2.6. Train-Validation Data Split

The reference dataset was split into training and validation sets using stratified sampling to preserve the distribution of the target variable. This approach ensures that both target classes are adequately represented in each subset, which is essential for reliable model training and evaluation.

2.7. Feature Selection

The feature selection was performed in two stages. In the first stage, we identified and removed highly correlated features (*total_assets, loan_term_group_medium* and *income_annum*), to reduce multicollinearity and minimize redundant information in the dataset.

In the second stage, we applied Recursive Feature Elimination (RFE), using a *RandomForestClassifier* as the base estimator. This approach allowed the ranking of all features based on their importance. From an original set of 39 variables, RFE identified 18 of them as the most impactful features for predictive modeling.

The final feature set was logged for reproducibility, documentation and further analysis, ensuring a more focused and robust input for subsequent modeling steps.

2.8. Model Selection and Optuna Hyperparameter Tuning

After feature selection, we evaluated four classification algorithms: Logistic Regression, Gaussian Naïve Bayes, Random Forest and Gradient Boosting. All models were trained on the training dataset, and their performance was compared using accuracy. The model that performed best was the *GradientBoostingClassifier,* achieving a score of 0.998, and was selected for further optimization.

To enhance its performance, we conducted hyperparameter tuning for our champion model using Optuna [6], an efficient and automated optimization framework.

Throughout this process, MLflow was used fmor experiment tracking, allowing us to systematically record model parameters, metrics and results for reproducibility and future analysis.

2.9. Final Model Training

After choosing and tuning *GradientBoostingClassifier*, the final model was trained using the processed training data, having the best columns as input for the model to train only with the features that showed higher relevance. The performance metrics used on both training and validation datasets were accuracy, f1-score, precision and recall, which were enough to assess the model's performance, having all computed metrics logged in for tracking (Figure 2). The results obtained were very impressive, indicating the model performed extremely well on unseen data. From these values we conclude there are no signs of overfitting, since the train and test scores are almost identical; all the metrics are close to 1.0 on both sets, meaning the model performs very well on all classes; the classifier is consistent and reliable.

We used SHAP on the trained model to better visualize how each feature was contributing to the final predictions. These individual insights were visualized in Figure 1 and allowed better understanding of how each variable was affecting the results and their relevance.

2.10. Prediction with Trained Model

The prediction phase is responsible for applying the trained model to new data and generating predictions, including validations that were tracked and a performance analysis that was

graphed. This step loads the previous *GradientBoostingClassifier* model and applies it to the processed batch data using only the selected features, returning the same input with the added predicted labels as a new column.

All descriptive statistics generated are automatically logged into MLflow to ensure every step can be analyzed and the results summarized. Every evaluation metric was also saved and logged for further analysis.

Additionally, a confusion matrix is generated for a visual representation of the predictions quality, demonstrating any misclassification. As shown in Figure 3, the model correctly classified practically all of the instances, resulting in only one misclassified sample. This reinforces the model's high performance in predicting both classes.

2.11. Production-Ready Model Deployment

The deployment stage marks the final transition from development to a production-ready model. At this point, the focus shifts from experimentation to stability, reproducibility, and maintainability. We ensured that all necessary components - trained model, selected features, and evaluation metrics - were properly serialized, stored, and versioned to support real-world use. This structured approach enables smooth inference, consistent preprocessing, and reliable performance tracking.

The model was serialized with joblib for efficient loading during batch or real-time predictions. Feature lists were saved with Python's pickle module to preserve their exact structure for inference integrity. Evaluation metrics like accuracy and F1 score were stored in JSON format to support transparent reporting and comparison across model versions.

Artifact paths were managed via configuration parameters, allowing flexible deployment across environments. By explicitly saving each component, the solution remains reproducible, auditable, and adaptable to changes. This design supports integration with serving platforms, CI/CD pipelines, and retraining workflows, while also easing maintenance tasks like rollback or re-evaluation [7].

Overall, the deployment strategy prioritizes reliability and long-term usability, ensuring consistent and responsible model serving in production.

2.12. Data Drift Detection

To evaluate the stability of input data over time, we applied two complementary statistical methods to detect distributional changes between the reference and current datasets [8].

The first method uses the Population Stability Index (PSI), which quantifies shifts in feature distributions. PSI was calculated using fixed bins derived from the reference dataset and adjusted to handle zero frequencies. A bar chart visualizing the PSI scores across all monitored features is generated and saved (Figure 4). This plot provides a clear overview of which features have deviated most from the reference distribution.

To capture drift over time, we employed Jensen-Shannon (JS) divergence. The current dataset was split into sequential chunks, and for each feature, the distribution in each chunk was compared to the reference. A separate line plot is produced for each feature (Figure 5), showing

JS divergence across chunks. This enables visual detection of both gradual and abrupt shifts in feature behavior.

An alerting mechanism was incorporated: if JS divergence for any chunk exceeds a defined threshold, it is flagged in the corresponding plot. These alerts serve as immediate visual cues, drawing attention to points in time where significant drift may occur. The sensitivity of this threshold is configurable based on monitoring requirements.

Together, the PSI bar chart and JS divergence plots offer both a global summary and a temporal view of data drift, enabling effective monitoring and early detection within the pipeline.

2.13. Pytest Coverage

Pytest coverage measures the extent to which the source code is exercised by automated tests. It is an important metric to assess the quality and reliability of the codebase by identifying untested parts that could harbor bugs or defects.

In this project, we implemented comprehensive testing to ensure robustness and correctness across all modules and pipelines. Achieving an overall coverage of 92% demonstrates a strong level of confidence in the stability and maintainability of the code, indicating that the majority of critical functionalities are well tested.

3. Evaluation Strategy and Success Criteria

To evaluate the success of our solution, we focus on how effectively the model generalizes to unseen data. While we do assess overall test performance, we go beyond a single performance metric to ensure that the model is not only accurate but also robust, interpretable, and production ready.

Although our dataset presents a mild class imbalance between approved and rejected loan applications, it is not severe enough to require complex rebalancing techniques. However, relying solely on overall accuracy can be misleading in such scenarios. Therefore, we complement standard evaluation with more balanced metrics. We prioritized the F1 score, which provides a harmonic mean of precision and recall - especially useful when both false positives and false negatives carry meaningful consequences. Given the binary nature of our classification task, we use the standard F1 score. We also examine precision and recall separately to better understand the trade-offs between approving unqualified applicants (false positives) and rejecting qualified ones (false negatives). To gain a deeper insight into model behavior, we include a confusion matrix, which clearly visualizes the distribution of prediction outcomes across all classes.

Beyond predictive performance, success in our project also means delivering a model that is trustworthy and explainable. To support this, we integrate feature importance analysis and SHAP values to explain individual predictions - a key consideration in regulated industries like finance. Lastly, we consider long-term model reliability, incorporating data drift evaluation to monitor changes in input distributions that could degrade performance after deployment.

4. Scalability Assessment and Future Development

This project demonstrates a modular architecture built with Kedro and tracked using MLflow. The structure allows for clear separation of concerns, reproducibility of results, and ease of experimentation, which are essential for maintaining reliability and auditability as the project grows. All data processing and modeling steps are implemented using Pandas and Scikit-learn. While this is effective for development and controlled environments, it introduces scalability limitations. In a production context with larger datasets or concurrent workloads, these tools may struggle to maintain performance. Their use, however, allows for rapid iteration and transparency in the early stages, and the modular design of the pipeline would support a transition to distributed frameworks like Spark without significant refactoring. MLflow enables consistent experiment tracking, parameter logging, and model versioning. Although configured locally for now, the same configuration can be ported seamlessly to a managed MLflow server or integrated into existing infrastructure to support collaboration and traceability in production. Model explainability is addressed using SHAP, which offers granular insights into feature contributions. While resource-intensive, it provides valuable transparency and can be adapted for batch-based or sampled execution when deployed at scale.

Overall, the technologies chosen align well with the objectives of a proof of concept and provide a strong foundation for extension into production, where enhancements can be made incrementally without requiring architectural redesign.

5. Conclusion

This project simulated the real-world deployment of a machine learning model for loan approval prediction. We built and end-to-end pipeline that follows a Machine Learning Operations oriented workflow, including key steps, such as, data cleaning, feature engineering, model tracking, deployment and drift monitoring.

The model achieved strong predictive performance and integrated essential tools: MLflow for traceability, SHAP for explainability and PSI/JS for monitoring data drift. These elements ensure transparency, accountability and long-term reliability, crucial in financial decision-making contexts. The modular design using Kedro enables scalable development, simplified experimentation a clear structure for future enhancements. Although developed as a proof of concept, this framework offers a robust foundation for real-world deployment and further development in more demanding environments.

6. References

[1] "Loan-Approval-Prediction-Dataset." Accessed: Jun. 28, 2025. [Online]. Available: https://www.kaggle.com/datasets/architsharma01/loan-approval-prediction-dataset

[2] "» Innovation in Banking." Accessed: Jun. 28, 2025. [Online]. Available: https://academy-of-business.com/innovation-in-banking/

[3] "Explaining MLOps using MLflow Tool - Analytics Vidhya." Accessed: Jun. 28, 2025. [Online]. Available: https://www.analyticsvidhya.com/blog/2022/12/explaining-mlops-using-mlflow-tool/

[4] "How does Great Expectations fit into ML Ops? • Great Expectations." Accessed: Jun. 28, 2025. [Online]. Available: https://greatexpectations.io/blog/ml-ops-great-expectations/

[5] "MLOps with a Feature Store - Hopsworks." Accessed: Jun. 28, 2025. [Online]. Available: https://www.hopsworks.ai/post/mlops-with-a-feature-store

[6] "MLOps with Optuna | Towards Data Science." Accessed: Jun. 28, 2025. [Online]. Available: https://towardsdatascience.com/mlops-with-optuna-b7c52d931b4b/

[7] "CI/CD for Machine Learning - MLOps Guide." Accessed: Jun. 28, 2025. [Online]. Available: https://mlops-guide.github.io/MLOps/CICDML/

[8] F. Bayram, B. S. Ahmed, and A. Kassler, "From concept drift to model degradation: An overview on performance-aware drift detectors," *Knowl Based Syst*, vol. 245, Jun. 2022, doi: 10.1016/j.knosys.2022.108632.

7. Appendix

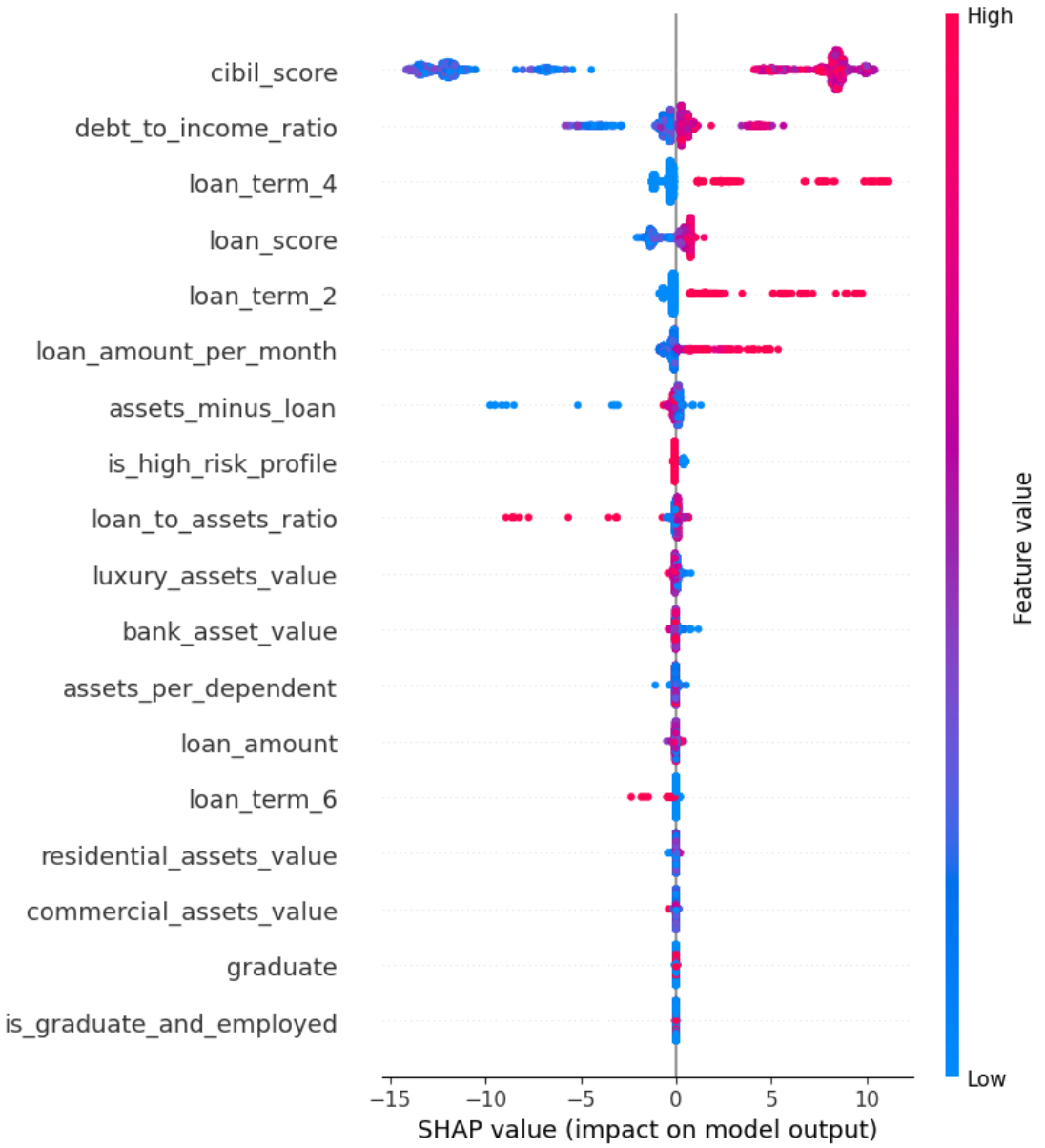

Figure 1 - SHAP Summary Plot

Metric	Value
precision_train	1
accuracy_train	1
precision_test	0.9984544049459041
recall_test	0.9973684210526316
recall_train	1
accuracy_test	0.9980487804878049
f1_test	0.9979067449782302
f1_train	1

Figure 2 - Final Model's Metrics

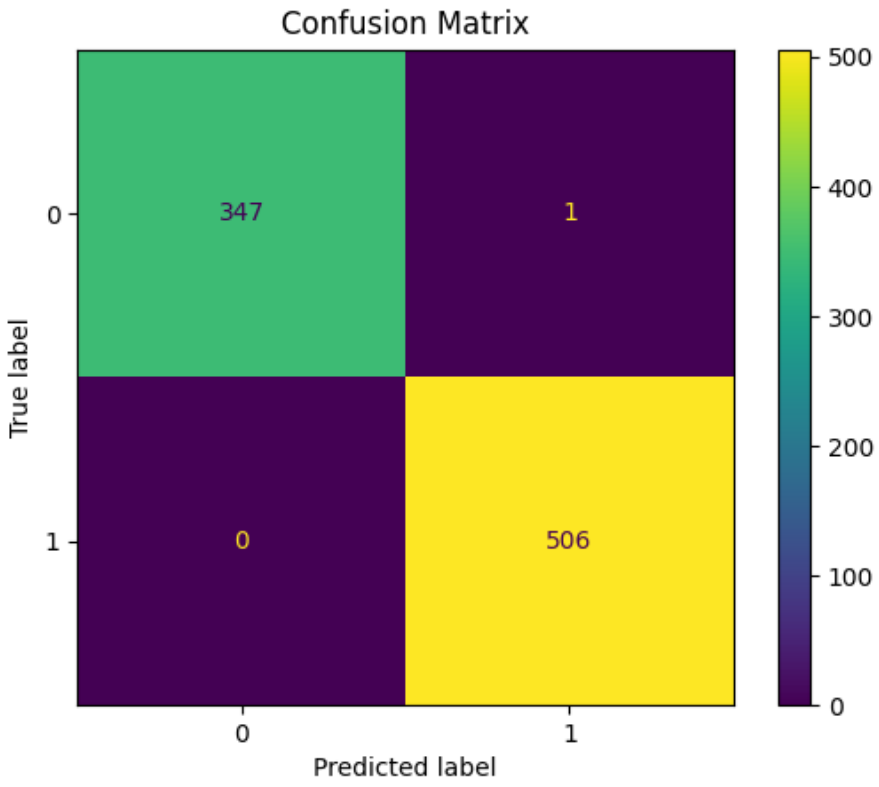

Figure 3 - Final Model's Confusion Matrix

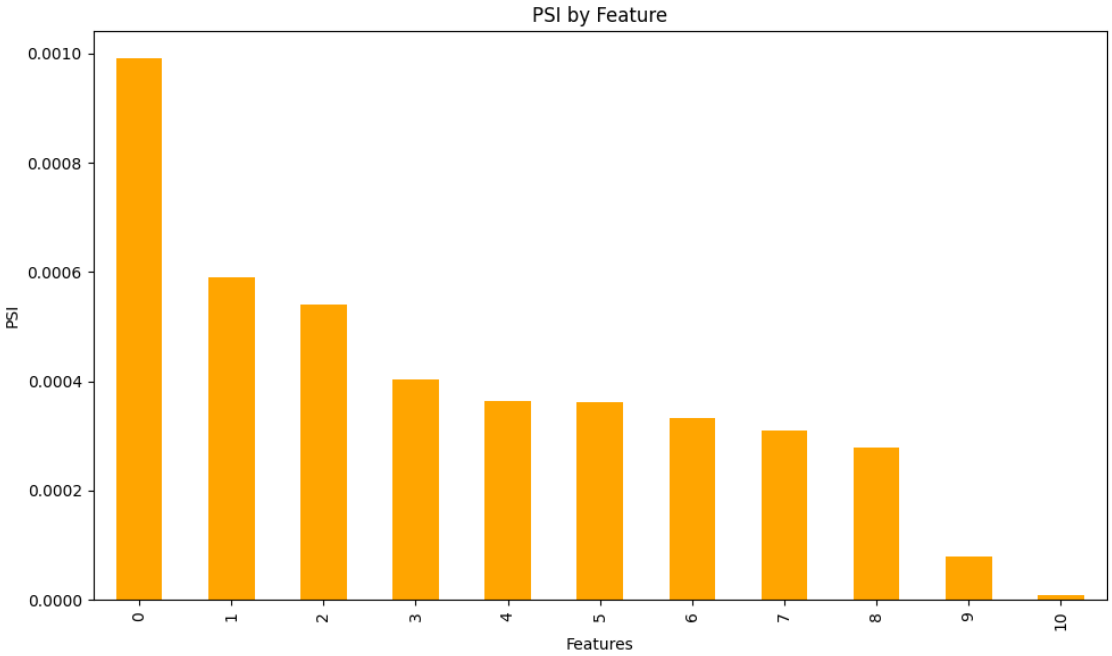

Figure 4 - Population Stability Index by Feature

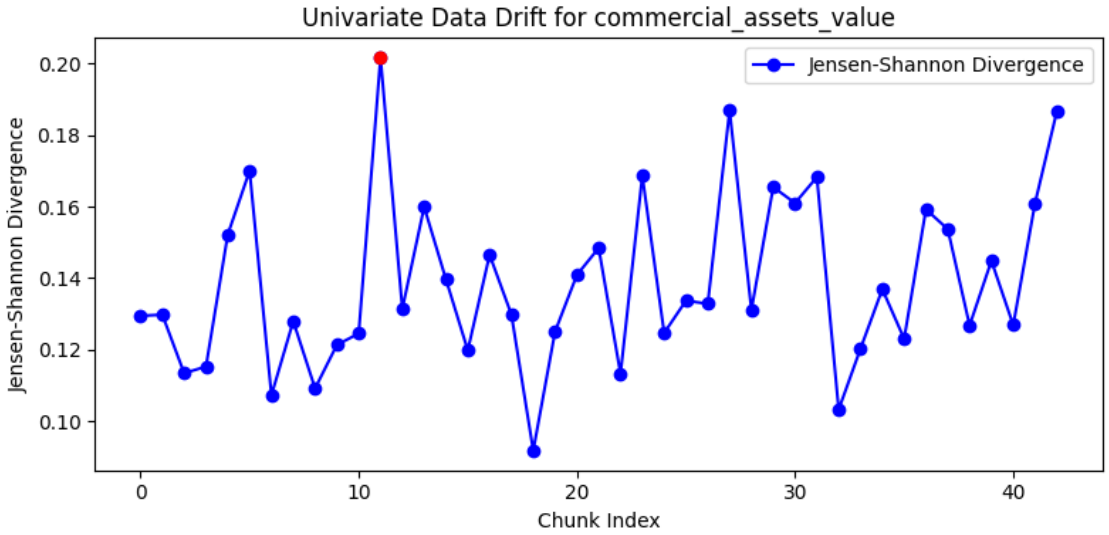

Figure 5 - Univariate Data Drift Graph Example

Feature Name	Feature Description
total_assets	Provides an overall measure of the individual's financial strength
is_graduate_and_employed	Combines education and stable income status. Being a graduate and an employee may signal higher income stability and lower risk
loan_to_assets_ratio	Measures how large the loan is relative to the total assets. A higher ratio may imply higher risk, as the loan is a bigger burden compared to the individual's resources.
cibil_score_bin	Categorizes the credit score which can help the model capture patterns associated with risk.
debt_to_income_ratio	Measures affordability, a higher ratio might mean a higher risk.
loan_score	Encapsulates many risk indicators, such as assets, CIBIL score, income, dependents and status, giving the model a quick heuristic of risk level.
assets_per_dependent	Accounts for financial responsibility, more assets per dependent imply better financial support, making the loan less risky
loan_amount_per_mounth	Helps assess the repayment burden
loan_term_group	Allows the model to learn common patterns associated with loan duration. Short and long loans have different risk profiles.
assets_minus_loan	Indicates the net worth after the loan is granted, a higher value indicates the individual is in a better position to handle repayment
is_high_risk_profile	Explicitly marks individuals with low credit scores or a high debt to income ratio, enables the model to quickly spot risky applicants.

Table 1 - Engineered Features and Descriptions