Master in Data Science and Advanced Analytics
NOVA Information Management School
Universidade Nova de Lisboa

MACHINE LEARNING OPERATIONS: HOTEL BOOKING CANCELLATION PREDICTION

Project Report

Group members:
Ilona Nacu, 20211602
Elizaveta Nosova, 20242132
João Venichand, 20211644
Rafael Silva, 20240511
Zofia Wojcik, 20240654

Github link: https://github.com/IlonaNacu/MLOps

Spring Semester 2025

ABSTRACT

This project implements a comprehensive MLOps pipeline for predicting hotel booking cancellations using machine learning techniques. The system leverages the Hotel Booking Cancellation Prediction dataset from Kaggle to build, deploy, and monitor a classification model that identifies bookings likely to be cancelled. The pipeline incorporates modern MLOps practices including data quality validation, model versioning with MLflow, automated testing, model serving, and data drift detection. The project demonstrates end-to-end machine learning operations suitable for production environments in the hospitality industry.

Keywords: MLOps, Hotel Booking, Cancellation Prediction, Machine Learning, Data Pipeline, Model Deployment

Table of Contents

1. Introduction and Project Motivation

1.1 Problem Statement

In this project we chose to work with a dataset about Hotel Booking Cancellations, acquired from Kaggle and with the characteristics typical of a real case of Booking Systems. We chose this dataset for several reasons:

- **Represents the Real World**: Simulates the real booking flow, including columns such as the number of adults, number of children, and the final booking status (Canceled or Not Canceled).
- **Practical Applicability**: Accurately predicting cancellations helps reduce revenue losses, optimize resource planning and improve occupancy rates.
- **Data Variability and Quality**: The dataset contains different types of data which makes it possible to apply various techniques, making it suitable for building a complete MLOps Pipeline.

1.2 Business Context

In the hotel industry, managing cancellations is critical to profitability. From a revenue management perspective, overbooking is generally preferred to under-occupancy. While overbooking can inconvenience guests, it is often a manageable situation, especially for hotel chains that can reallocate bookings internally. Last-minute cancellations lead to direct financial losses. Despite the presence of cancellation fees, hotels may not be able to resell the room in time, especially during off-peak periods. This mismatch between expected and actual occupancy disrupts planning and reduces profitability.

An accurate prediction system for early detection of booking cancellations enables hotels to proactively adjust allotments across sales channels and optimize occupancy. From a business perspective, identifying as many cancellations as possible is more valuable than avoiding occasional false alarms, as the cost of a missed cancellation typically outweighs that of a mistaken one.

1.3 Objectives

The main objective is to build a complete and modular Machine Learning Pipeline that represents the life cycle of a model in a real context, where we want to predict whether a booking will be canceled, based on the information that is provided in the dataset. This pipeline must include data ingestion, data unit tests, preprocessing, model versioning and selection, performance evaluation and explainability, pipeline testing, and Modularization with Kedro.

2. Project Planning and Outline

The development of the project followed an iterative and adaptive approach, reflecting an MLOps development cycle where rapid validation, progressive modularization, robustness and continuous improvement were the priorities over following a rigid plan. While we did not define sprints at the outset, the workflow naturally evolved in clear stages that we structured as sprints for easier understanding.

Sprint 1: Exploratory Analysis and Data Understanding

We started the project with exploratory data analysis, using a Jupyter Notebook dedicated to EDA. This step was essential for understanding the structure of the dataset, identifying relevant patterns, outliers, missing values, among others, and the insights extracted allowed us to define the basis of our pipeline.

Sprint 2: Ingestion and Data Validation
Next we developed the *ingestion_pipeline*, which is responsible for loading the data in a modular way, optional upload of raw features to *Hopswork* and is complemented by *data_unit_tests* implemented with *Great Expectations* to check automatically data quality validations. At this stage, we also implemented the *split_data* pipeline, which splits the data into training and test sets.

Sprint 3: Preprocessing
With the base of the data secured, we evaluated the impact of different preprocessing algorithms, such as encoding and normalization. This experimentation led to the construction of the *preprocessing_train* pipeline in which we applied the necessary treatments to the training data. It was joined by *upload_features*, which prepared the features for reuse using *Hopswork*, and *split_train*, which further splits the training set into training and validation subsets.

Sprint 4: Model Training and Evaluation
With the data ready, we created the pipelines *model_train* and *model_selection* which allowed us to train, evaluate and compare different models in a structured way, with the support of MLflow for versioning. In this phase, we also applied model explainability techniques (SHAP, Permutation Importance and PDP) to ensure transparency.

Sprint 5: Model Serving
With a champion model selected, we created the *preprocess_test* and *model_predict* to ensure that the model is compatible with unseen data and works as expected.

Sprint 6: Feature Selection and Robustness Testing
Finally, we implemented the *feature_selection* pipeline to analyze the impact of the most relevant variables and explore model simplifications. We also included data drift tests to assess the robustness of the model in scenarios where the data changes.

Throughout the entire process, we introduced unit tests for critical functions and pipelines, guaranteeing modularization with the use of *Kedro*, to strengthen the system's reliability. The organization of the folders can be seen in Figure 1. The visualization of the data, pipelines, nodes and all the connections between them can be checked in Figure 2.

3. EXPLORATORY DATA ANALYSIS (EDA)

We started with an exploratory data analysis (EDA), using visualizations to identify distributions, patterns and potential problems with the data.

Dataset characteristics:

- Provenance: hotel's Property Management System
- Size: 36285 rows
- Features: 16
- Target: *Booking status*, imbalanced (33% Cancelled vs 67% Not Cancelled)
- Issues: data types, imbalance in categorical/discrete features, in few cases: outliers invalid dates, logically invalid entries

A histogram of the date of reservation, segmented by booking status and visible in Figure 3, revealed the presence of a few outliers reservations that were promptly removed from the dataset (specifically 3 bookings made before June 2017 booked more than 7 months in advance). There is also a clear tendency for cancellations in certain periods, suggesting a possible temporal influence on user behaviour. This motivated our choice to split the data chronologically in further steps, as class balance patterns are relatively stable over time and, in practice, predictions would be made in the order bookings occur.

We identified outliers that were discovered based on logical inconsistencies such as the total number of nights (*weekend nights* + *week nights*) being 0 or the total number of people being zero (*adults* + *children*). These cases represent invalid bookings reflecting dataset quality, and in real-case scenario could signalize technical issues of the booking system.

Furthermore, we also found that several categorical features represented unbalanced distributions such as *market segment type* and *room type*. In the binary features *repeated* and *car parking space,* as well as discrete features *P-C* and *P-not-C* (number of client's cancelled/uncanceled previous bookings) only one category/value appeared in more than 99% of bookings, making them uninformative. These insights were important for informing preprocessing decisions, namely in the aggregation of poorly represented categories, elimination of certain variables, among other possible transformations.

Through boxplots we discovered the presence of outliers in several variables, including *lead time* and *average price*, as shown in Figure 4. A correlation analysis showed no significant correlation between the features and with the target. Finally, data exploration led us to identify possible seasonal patterns.

4. Data Processing and Feature Engineering

At the ingestion stage, raw data was uploaded, with the option of uploading raw features to the feature store. To make the upload feasible, some preliminary cleaning steps were mandatory, such as converting string date of reservation to datetime, dropping rows with invalid dates and replacing whitespace character in column names.

Data unit tests on ingested data checked the existence of target column and unique identifier of bookings, expected values of categorical features and value ranges of numerical features. Although none of blocking assertions, such as lack of target variable or identifier and bookings for past dates (negative *lead_time*) stopped the pipeline, the resulting data report revealed some upper outliers as per *average_price* and *lead_time* range expectations.

In the preprocessing stage after 75%-25% chronological train-test split, we performed data cleaning based on EDA and data unit tests results: fixing variables datatypes, eliminating invalid bookings for 0 guests and/or 0 nights, removing *average_price* and *lead_time* outliers and dropping uninformative columns. Additionally, a report on main statistics of cleaned features was recorded.

In the next step, we performed feature engineering on clean train data. First, we encapsulated observed seasonal patterns in a new feature *season.* Based on date of reservation and lead time, we defined date of arrival and assigned it to high *season* for summer months and December and *low* season otherwise. Then we grouped underrepresented categories of *market_segment_type* and *room_type* and into *Other* category to reduce cardinality. We created two versions of preprocessed data: with encoded categorical features and unmodified numerical features for tree-based classifiers and with encoded categorical features and scaled

numerical features for logistic regression. *OneHotEncoder* and *StandardScaler* were used for this purpose. Afterwards we introduced an optional stage of uploading both versions of preprocessed data to the feature store for further reuse.

The preprocessing of test data would follow the same steps as train data, with the only exception that missing values, if any, are to be imputed using clean data statistics report: with median values for numerical features (to account for discrete variables) and with the most frequent values for categorical features.

5. MODEL DEVELOPMENT AND RESULTS

5.1 MODEL SELECTION PROCESS

We started with a baseline model, *Logistic Regression* with default parameters, and with feature selection as optional, to establish a performance benchmark, in the *model_train* pipeline. This pipeline only works this way on the first run because after we get a new champion model, as described below, it will always use this model, as it simulates that it's the current model in production.

Next, we used the *model_selection* pipeline, which includes *GridSearch* to tune hyperparameters and identify the best-performing model. The *GridSearch* is optional, and it can be altered to not run by changing its parameters in the *parameters.yml* file. The models obtained are also added to *MLflow*.

Models Evaluated:

- **Logistic Regression:** A simple baseline model used to assess linear separability of the data. It required feature scaling due to its sensitivity to feature magnitude. Feature selection was set to be optional.
- **Random Forest:** An ensemble model that usually handles non-linear relationships well and requires minimal preprocessing. It delivered balanced performance across all metrics.
- **XGBoost:** A gradient boosting model that builds an ensemble of decision trees in a sequential manner, known for high performance in structured data. It provided competitive results but was more prone to overfitting.

If *GridSearch* is set to not run, these three models are run and compared with their default parameters.

Once a superior model was found, it replaced the baseline and became the current "champion." In our setup, the champion model simulates a production-ready model and is deployed immediately. So, when the *model_train* pipeline is ran again, instead of running the baseline, the new champion is always the one being run. In this pipleine, the model is registered in *MLFlow* as the champion. We then apply it to unseen test data, in *model_predict* pipeline to simulate the behavior on unseen data.

5.2 FINAL MODEL PERFORMANCE

Performance Metrics:

- **Accuracy:** Measures overall correctness. Not prioritized due to class imbalance in the dataset. It was used as a baseline for comparison.

- **Precision:** Important to measure how many predicted positives were actually correct, useful when false positives are costly.
- **Recall:** Reflects the model's ability to detect all actual positives. Useful when missing a positive case is critical. More useful for our case than precision, but we kept both as they are more informative together.
- **AUC-ROC:** Evaluates the model's ability to distinguish between classes across different thresholds. Used as a secondary diagnostic to understand discrimination ability.
- **F1-Score:** The harmonic mean of precision and recall. Chosen as the decisive metric because it's ideal for imbalanced classification tasks.

These performance metrics were used both for *model_train* and for *model_selection* pipelines, but in both, more consideration was given to the F1 score.

Selected Model: Based on the comparison of F1 scores across all tested models, Random Forest emerged as the best-performing model. It is currently the champion model, offering strong predictive performance with minimal preprocessing and stable results across validation folds. On the training pipeline, it obtained an F1 Score of 0.82.

5.3 Feature Importance and Model Explainability

To ensure transparency and interpretability of the predictions made by our final model we applied two explainability techniques:

- **Permutation Importance** revealed that the most influential features were *lead_time*, *special_requests* and *average_price*, as shown in Figure 5. This method measures how much the model's performance decreases when the values are randomly shuffled, with greater drops indicating more important variables.
- **Partial Dependence Plots (PDP)** further helped us to understand how these key features influence the model, as displayed in Figure 6. For instance, the probability of cancellation sharply increases for bookings with a lead time above 140 days; a higher number of special requests correlates with a lower likelihood of cancellation; and as the average price increases, so does the predicted probability of cancellation.

6. Data Drift

Our data drift detection implementation leverages **NannyML's UnivariateDriftCalculator** to provide comprehensive monitoring of feature distributions between training (reference) and production (current) datasets. The system automatically categorizes features and applies appropriate statistical tests based on data types. All features from model training are used for univariate drift detection, controlled by the parameter *use_all_model_features* set to *true*. Drift detection leverages a timestamp column (*date_of_reservation*) to identify shifts over time. The modularity of the project allows to switch detection methods, adjust thresholds, and integrate the results into real-life monitoring system

Different statistical methods are based on feature types:

- Categorical features: Chi-squared tests.
- Discrete (integer) features: For simplicity also treated as categorical, using Chi-squared tests.
- Continuous features: Jensen-Shannon divergence

Alerts for data drift are triggered if the calculated metrics surpass a defined threshold (*alert_threshold* set to 0.05). Drift analysis results can be visualized and saved. Refer to Figure 7. and Figure 8. for distribution of integer and binary column respectively.

7. Conclusions and Future Work

7.1 Project Summary

This project was created as a proof of concept, but nevertheless incorporates techniques and practices aligned with the production level of MLOps. The pipeline is structured using *Kedro*, ensuring modularity and reproducibility, as well as having several unit tests for both data and pipeline functions, reinforcing the reliability and maintainability of the code.

Additionally, it uses *MLflow* for versioning of the models, ensuring traceability of parameters and metrics and the champion model produced is also explained using Permutation Importance and PDP to ensure transparency of predictions. All of these components together are crucial for real-world deployments.

7.2 MLOps Implementation Success

All key components expected in a MLOps pipeline were successfully integrated:

- *Great Expectations* for automated data quality checks.
- *MLflow* for experiment tracking and model versioning.
- *Kedro* modularization for scalable pipeline deployment.
- Model explainability using PDP and Permutation Importance.
- Data drift detection with *NannyML* based on statistical monitoring.

7.3 Future Improvements

If available, enriching the input with data from other sources could bring more insights. For example, information on payment, records from CRM systems, call-center logs, or internal risk score estimations could be incorporated at the ingestion stage. To ensure non-decreasing data quality, data unit tests can be expanded, including creation of custom expectations.

For further increase in performance, a more fine-grained feature engineering can be conducted. The new introduced feature *season* did not appear to significantly affect the predictions, judging by explainability exploration. A higher cardinality could be applied, such as separating Christmas holidays from summer season, and, if the location of the hotel is known, local holiday bridges like Easter break. Additionally, other features can be added, such as day of the week of booking or arrival, interaction features and, with extra data, customer profiling.

For future improvements regarding the modeling stage, one suggestion is to initially tag the best-performing trained model as a *challenger*, rather than immediately registering it as the *champion*. Only after verifying that it truly outperforms the current champion should it be promoted. Expanding the hyperparameter grid

in *GridSearch* and testing a wider range of models could also enhance model selection in future iterations and including overfitting analysis, instead of only using the F1 score metric, to guarantee that our model is generalizing well. Additionally, rather than relying solely on automatic logging in *MLflow*, explicitly specifying which metrics and artifacts to log, along with clear descriptions, would make the tracking process more robust and transparent.

One risk identified is the lack of scalability and reliability for very large datasets due to the use of *Pandas*. To solve this problem we would need to move certain components of the *Pandas* pipelines to distribution systems such as *Spark* and for real-world deployment we would need containerization and orchestration tools (such as *Docker* and *Kubernetes*, respectively).

Our data drift implementation is designed to monitor and detect shifts in data distributions within a static data environment. Since our data does not stream in real time, we utilize the data used to train the model as the reference dataset, against which batches of current data are compared for drift analysis. This could be expanded to handle streaming data sources, allowing for real-time drift analysis. Also, for more reliable results a thorough investigation into alarm threshold for each feature should be applied. Most importantly, this should be done on features like *lead_time* or *special_requests*, which have high permutation importance. Another direction for improvement would be applying trigger-based retraining for automated retraining pipelines.

Overall, this project successfully demonstrates how MLOps practices can be implemented in a proof-of-concept setting. Through the integration of modular pipeline design, automated data validation, model and data versioning, explainability, and drift detection mechanisms, the project lays a solid foundation for reliable machine learning deployment. While there are areas for growth—such as incorporating more diverse data sources, expanding feature engineering, enhancing model evaluation procedures, and improving infrastructure scalability—this work establishes a clear and practical roadmap for transitioning into a fully productionized ML system.

8. REFERENCES

[1] Dataset Source: Hotel Booking Cancellation Prediction. Kaggle. Available: https://www.kaggle.com/datasets/youssefaboelwafa/hotel-booking-cancellation-prediction

9. APPENDIX

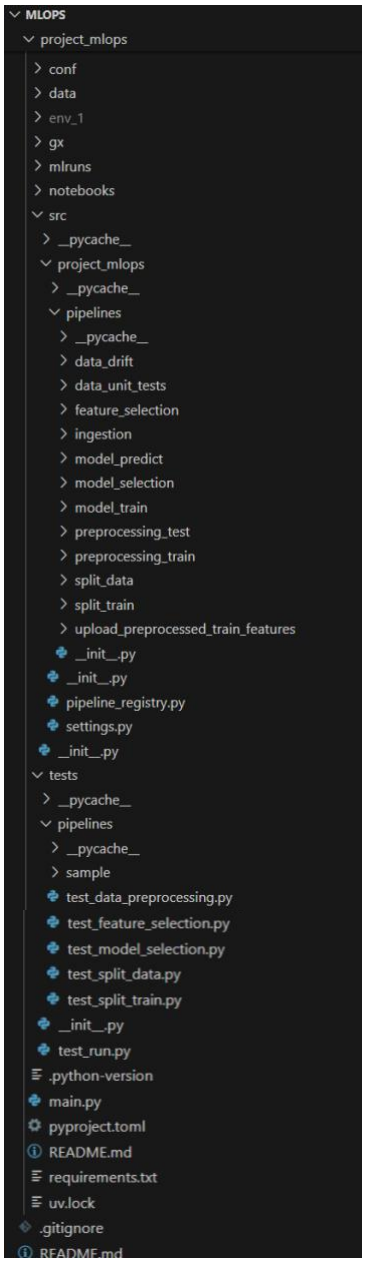

Figure 1: Folder Organization

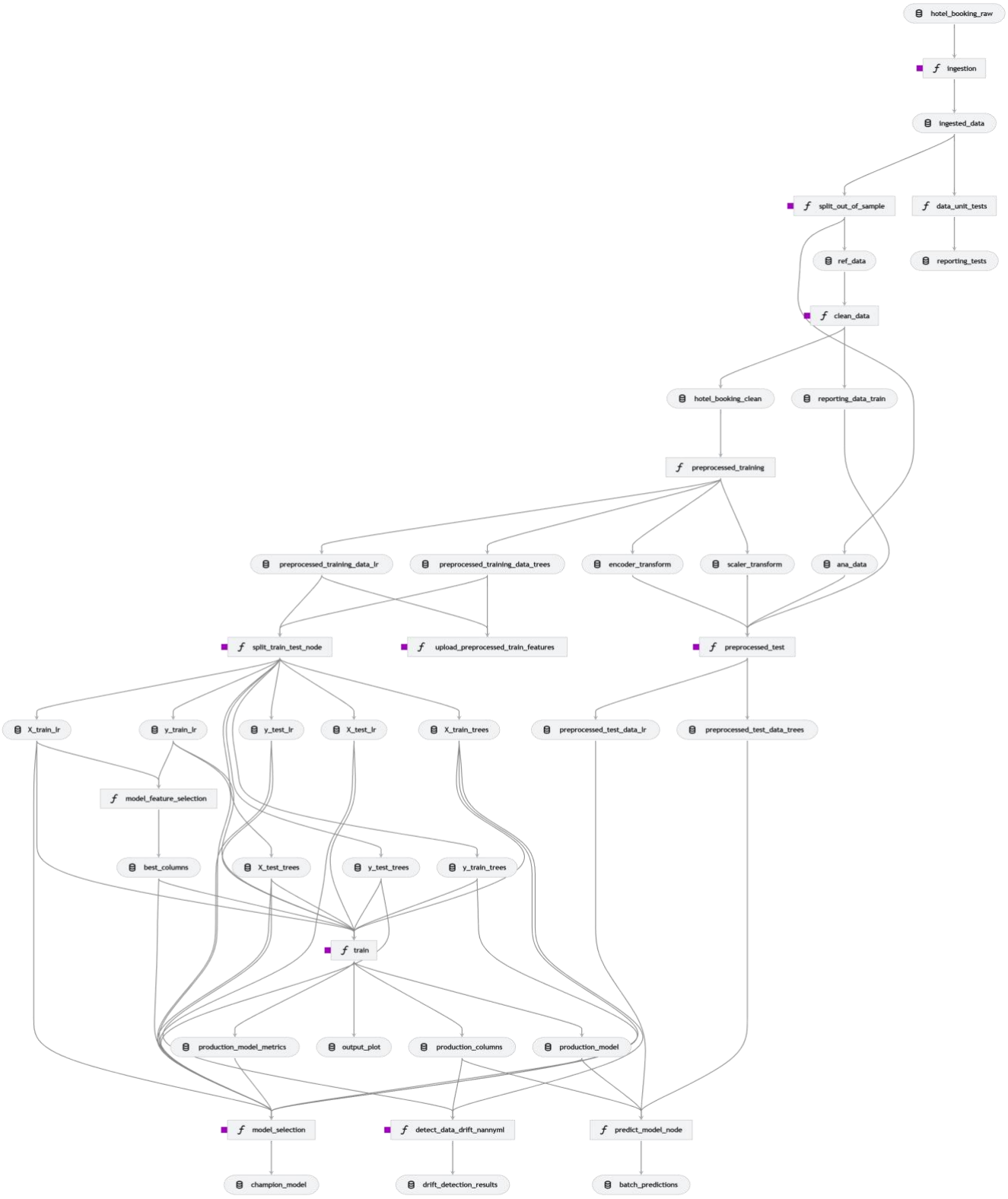

Figure 2: Kedro pipeline visualization

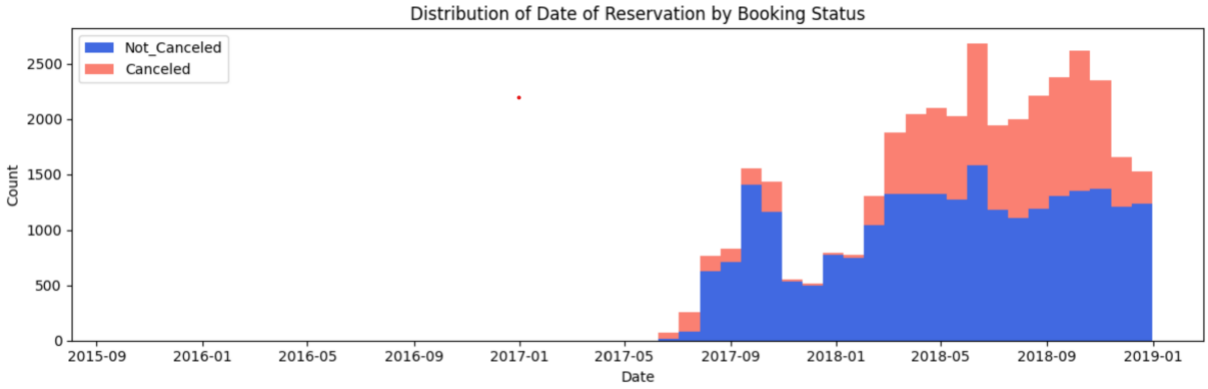

Figure 3: Histogram of Reservation Dates by Booking Status.

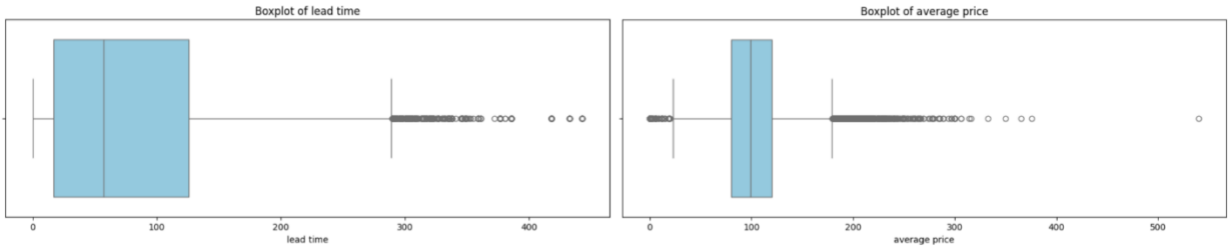

Figure 4: Boxplots of Lead Time and Average Price.

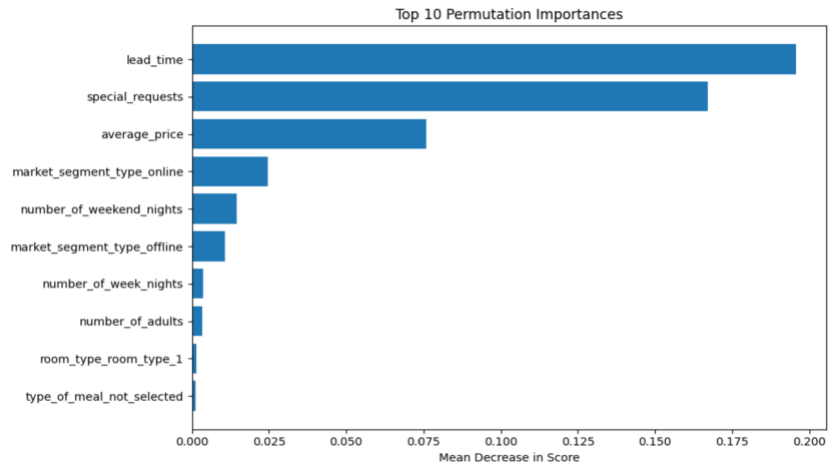

Figure 5: Top 10 Features by Permutation Importance

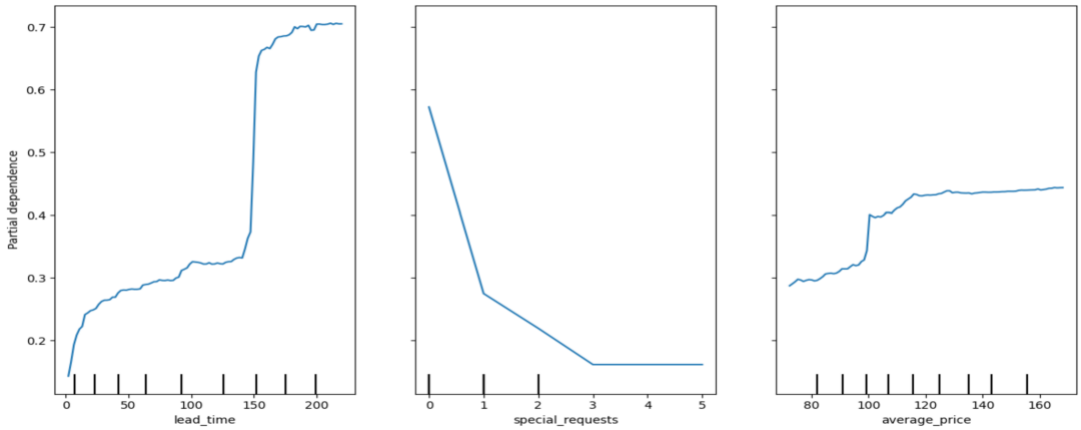

Figure 6: Partial Dependence Plots of Key Features.

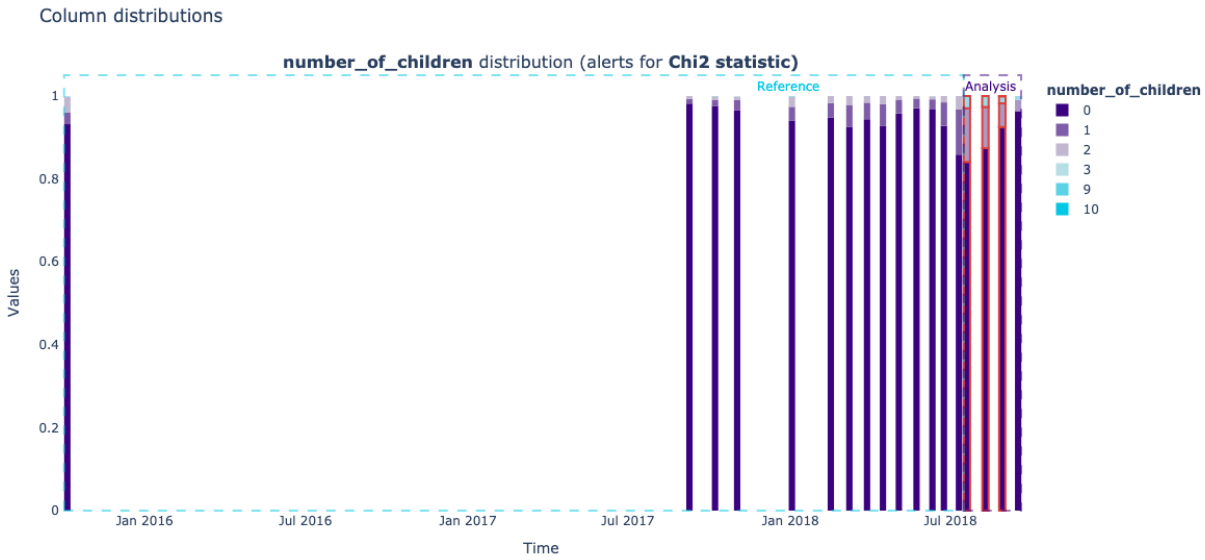

Figure 7: NannyML Plot for Data Drift of a Discrete Column Type.

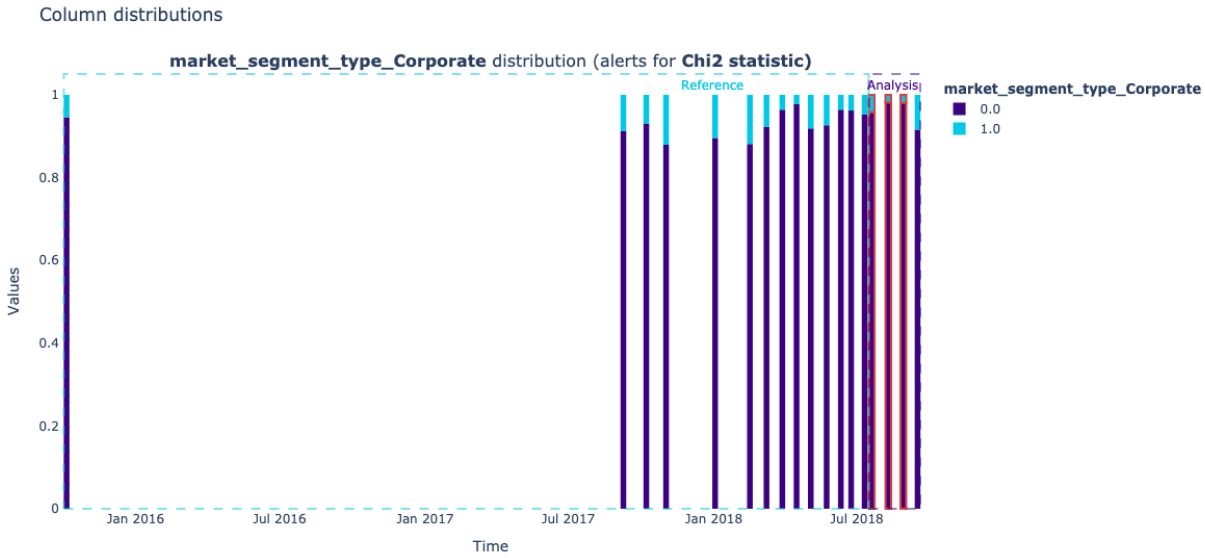

Figure 8: NannyML Plot for Data Drift of a Discrete Column Type.