Backorder Prediction

By

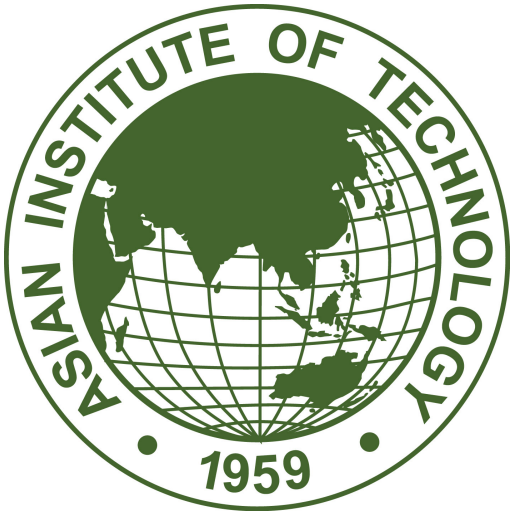
Group 13

Ashmita Phuyal st124454

Sitthiwat Damrongpreechar st123994

A Report for Computer Programming for

Data Science and Artificial Intelligence Project



Asian Institute of Technology

School of Engineering and Technology

Thailand

24th November, 2023

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Background and Motivation

When a customer orders a product, which is not available in the inventory or temporarily out of stock, and the customer decides to wait until the product is available and guaranteed to be dispatched, then this scenario is called backorder. Backorders if not properly handled will negatively affect a company's income, share price, and consumer trust which result in the loss of a client or selling order. This is a crucial component of supply chain management and inventory optimization, and it involves forecasting the likelihood of certain products being temporarily unavailable due to insufficient stock levels. Due to traditional inventory management systems, frequently caused stockouts, manufacturing delays, and dissatisfied clients have created the impact in business leading to heavy loss. With the advancement in technology and machine learning, it is possible to develop more accurate and proactive backorder prediction models.

The objective for the backorder prediction implementation is to identify and forecast potential instances of products facing backorders within the inventory. This strategic approach aims to enhance efficiency in inventory and supply chain management, enabling companies to proactively address potential stockouts and minimize the risk of understocking. By accurately predicting which products have the potential to become backorders, businesses can optimize their inventory levels, prevent customer dissatisfaction due to unfulfilled orders, and ultimately improve market competitiveness, earnings, and overall operational effectiveness. This proactive approach assists companies in steering clear of the costly and inefficient practices associated with both overstocking and understocking, ensuring a balanced inventory that aligns with consumer demand and eliminates unnecessary holding costs.

1.2 Business Overview

The challenge of precisely forecasting and managing inventory to avoid surpluses in order to minimize the costs associated with carrying excess inventory is the main issue with backorder prediction. Variable customer demand, unstable times for suppliers, and the balance between inventory levels and their cost constraints are the highlights of this issue. The main purpose is to

maintain high customer satisfaction while avoiding the expense of carrying excess inventory by preventing stockouts. It takes sophisticated data analysis, predictive modeling, and efficient supply chain process integration to achieve this balance, which makes it a difficult and crucial task for companies in a variety of sectors.

1.3 Impacts

Implementing a backorder prediction provides advantages on a large scale in the business areas. The end users like companies can effectively manage expectations of the customers by communicating proactively in order to meet the potential delivery delays. By reducing excess inventory, the companies can generate significant savings in terms of storage. As a result, businesses can ensure that products are available when needed, which could increase sales and overall revenue and can be able to better meet customer demand. The system enhances overall supply chain efficiency and fosters better supplier relations.

2. PROBLEM STATEMENT

The customers purchasing the product are unfamiliar about the behavior of the backorder policies. The e-commerce companies usually face the challenges of frequent backorders which create a huge impact on customer satisfaction, substantial costs, inventory management, and sales. The dynamic market conditions, fluctuation in customer demands, and diverse products are the main problems of backorder. In order to solve this problem, we need to develop a robust predicting system selecting the best possible models to predict the orders.

2.1 Objective

Creating a system that can precisely predict and handle situations in which product stock levels are anticipated to drop below customer demand is the main goal of backorder prediction. This involves the proactive identification of potential backorders, allowing businesses to take timely and informed actions to prevent or mitigate stockouts. The key objectives include:

- Preventing Stockouts: The main goal is to stop or reduce the number of times that products are out of stock when customers place orders. This will help to keep them happy and prevent revenue loss.
- Reducing Superfluous Stock: In addition to avoiding stockouts, the goal is to steer clear of overstocking, which can lead to needless holding expenses, space usage, and capital invested in inventory.
- Improving Inventory Management: Creating data-driven models and procedures to guarantee that the appropriate number of goods is available when needed, taking into account variables such as lead times for supplies and fluctuations in demand.
- Improving Customer Satisfaction: Keeping high levels of customer satisfaction requires managing customer expectations through efficient communication in backorder situations.

2.2 Scope

Backorder prediction covers a wide range of topics and factors in the context of inventory control and supply chain management. It has a dynamic application that evolves as technology advances and the market changes. It includes an in-depth approach to deal with the issues related to customer satisfaction and inventory management in a variety of industries. It comprises:

- Demand Forecasting: Using demand forecasting methods to project future trends and requirements from customers.
- Real-Time Data Integration: For more dynamic forecasts, the scope includes integrating real-time data sources, such as supplier performance metrics and market conditions.
- Cost optimization: Process of determining the best course of action for inventory management by weighing the costs of backorders against the costs of keeping inventory.
- Market dynamics : Understanding the effects of shifting consumer preferences and market conditions on backorder projections

3. RELATED WORKS

Prediction of backorder is a challenging task as it is uncertain and varies with time. In recent times, numerous research papers and studies have contributed to the advancement of backorder prediction in the fields of supply chain management, and inventory control. Several machine learning models have been developed to find the best prediction for the backorders.

In a recent study by Rodrigo, Eduardo , and Leonardo [1], the researcher provides insights into predictive modeling techniques for backorder prediction, emphasizing the use of machine learning algorithms to improve accuracy. The paper addresses the complexities of model building and the importance of precise backorder forecasts for companies looking to maximize consumer satisfaction and inventory control.

The latest research study have been conducted by Samiul Islam and Saman Hassanzadeh Amin using Distributed Random Forest (DRF) and Gradient Boosting Machine (GBM) [2] and have observed the performances of the machine learning models to showcase how these models can be used to predict the probable backorder products before actual sales take place. In this study, a ranged technique is proposed to solve this issue. Both machine learning models have been trained on actual and ranged data, and their respective performances have been compared. According to the comparison result, training the models with the ranged data improves their performance by about 20%.

In a broader context and based on real life, Hui Gao, Quanhui Ren and Chunfeng Lv [3] have explored the potential of neural networks and naive bayes, to predict backorder products. In this study, the likely backorder goods have been forecasted using two machine learning algorithms. Upon receiving the experimental results using three statistical criteria—accuracy, precision, misclassification rate, and ROC graphs, each of the results were examined to be proved that it attains a new level of state-of-the-art performance.

In order to improve inventory management, customer satisfaction, and operational efficiency, these research papers collectively offer insightful perspectives and methodologies to the field of

backorder prediction. They provide insights on light on a variety of topics, including machine learning techniques, data mining, neural network models, and real-time strategies.

4. DATASETS

The primary source of the dataset is from Kaggle Datasets [4] which contains information of a company's products for the 8 weeks prior to the week that is going to be predicted. The dataset consists of two main files: train.csv, which serves as the training set, and test.csv, used for testing purposes. These products are characterized by various attributes that are crucial for inventory management and risk assessment as they contain various attributes related to the company's products.

This train dataset contains information on various health-related attributes with 23 features of data and consists of records. Eight features—including the target variable—are categorical, while the remaining 15 are numerical, according to the dataset. The product identifier, or sku, appears in the first column and is distinct for every dataset.

5. METHODOLOGY

5.1 Exploratory Data Analysis (EDA)

- Data Considerations:

The dataset consists of 1,929,937 rows and 23 columns. These columns contain a mix of numerical and categorical features, with one column containing missing values that we will address in the imputation section.

Here's a breakdown of the data considerations:

- Data Shape: The dataset contains 1,929,937 rows and 23 columns.
- Column Types: Among the columns, 15 are numerical features, and 8 are categorical features.
- Categorical Features: The categorical features are 'sku', 'potential_issue', 'deck_risk', 'oe_constraint', 'ppap_risk', 'stop_auto_buy', 'rev_stop', and

'went_on_backorder.' The field: 'sku' is considered a product identifier and is not used in further analysis.

- Numerical Features: The numerical features include 'national_inv,' 'lead_time,' 'in_transit_qty,' 'forecast_3_month,' 'forecast_6_month,' 'forecast_9_month,' 'sales_1_month,' 'sales_3_month,' 'sales_6_month,' 'sales_9_month,' 'min_bank,' 'pieces_past_due,' 'perf_6_month_avg,' 'perf_12_month_avg,' and 'local_bo_qty.'
- Missing Values: The features 'perf_6_month_avg' and 'perf_12_month_avg' contain missing values denoted by -99.0. We have done imputation to remove the missing values from these two fields.
- Data Cleaning: The two rows with all missing values for all columns have been removed from the dataset. After the removal of these rows, there are no more missing values in the dataset. The data cleaning process has ensured the quality, integrity, and usefulness of the data for further analysis and modeling.
- Missing values found in lead_time, perf_6_month_avg, and perf_12_month_avg columns are resolved using the imputation process.

- Univariate Analysis:

The subplot we used showcases the data distribution for numerical and categorical features in the dataset. We used Kernel Density Estimate (KDE) Plots for numerical features and count plots for categorical features. Most of the numerical features are right-skewed, with a tail extending to the right. Hence, the majority of data points are concentrated on the lower end of the scale, while some data points have higher values, creating a long right tail. All categorical features contain binary classification data ('Yes', 'No'), exhibiting a considerable imbalance between classes. Even the target feature 'went_on_backorder' demonstrates a high imbalance. Univariate analysis underscores the substantial skewness and imbalance present in these categorical features, emphasizing the need for appropriate handling methods.

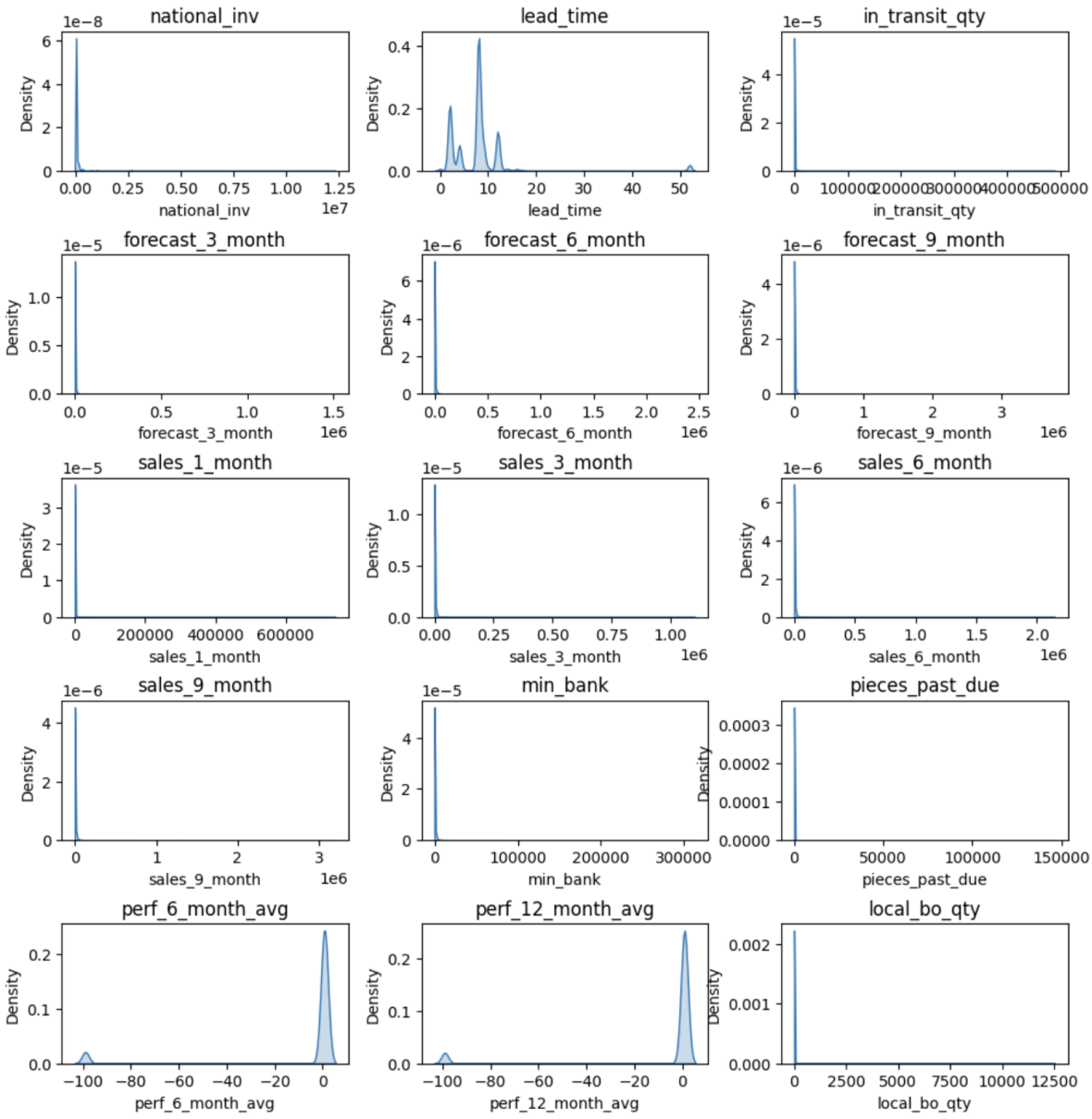


Figure 1: KDE plot for numerical features

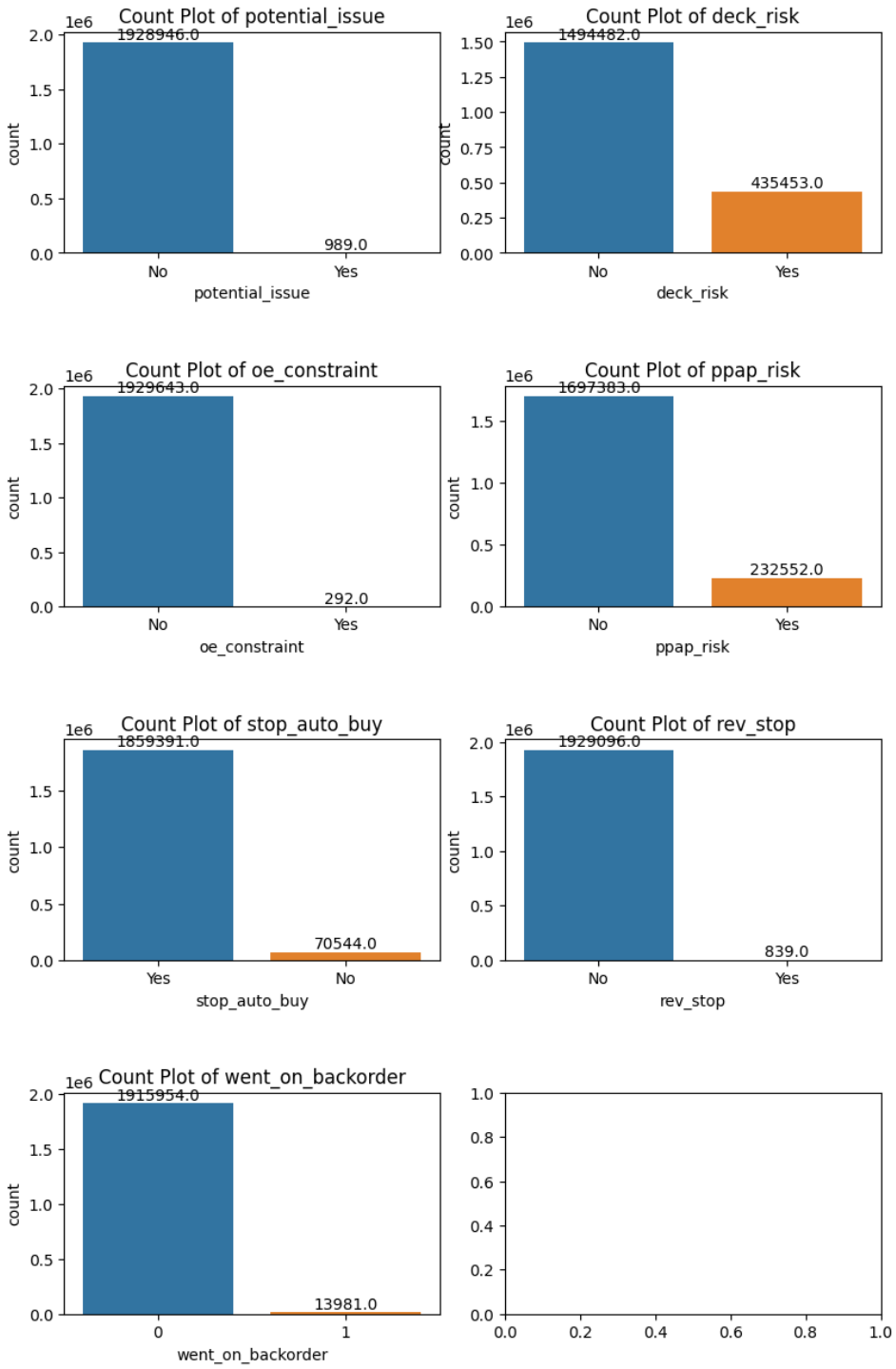


Figure 2: Countplot for categorical features

- Multivariate Analysis
    - Box Plot

The box plots illustrate the distribution of numerical variables from the dataset based on the values of the categorical variable 'went_on_backorder.' This provides insights into the presence of outliers in the data. The boxplots reveal that for all features, a substantial number of data points extend beyond the whiskers on both ends, indicating the presence of numerous outliers. This suggests that the whiskers, typically representing a range of 1.5 times the interquartile range (IQR), encompass a significant amount of data, highlighting the prevalence of outliers in the dataset.

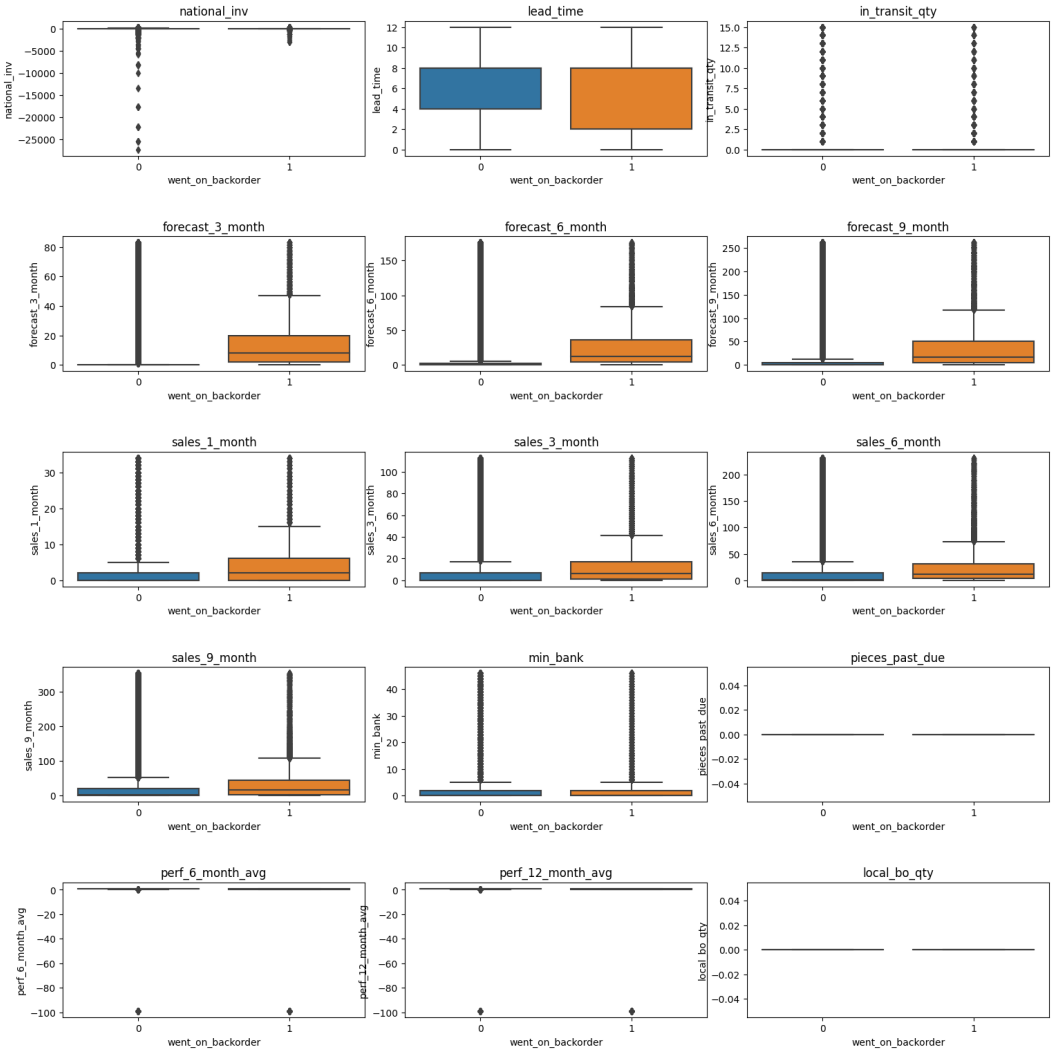


Figure 3: Box Plot for Numerical Features

- Correlation Matrix and Predictive Power Score (PPS)

The correlation matrix heatmap focuses on linear relationships and is useful for feature selection. The below heat map shows that the significant correlations in your data are positive and have the following results:

- Forecast_3_month, forecast_6_month, and forecast_9_month have strong correlations with each other with a correlation coefficient as 0.99. The strong correlation coefficient can come from the forecasting which is linear regression prediction.
- sales_1_month, sales_3_month, sales_6_month, and sales_9_month are strongly correlated with each other, with correlation coefficients ranging from 0.82 to 0.98. These are reasonable due to the similar features involved in sale quantity with different periods of time.
- The two variables: perf_6_month_avg and perf_12_month_avg are highly correlated with a coefficient of 0.97. And same with the sales features, the high correlation coefficient is reasonable due to the similar features involved with suppliers (sources).

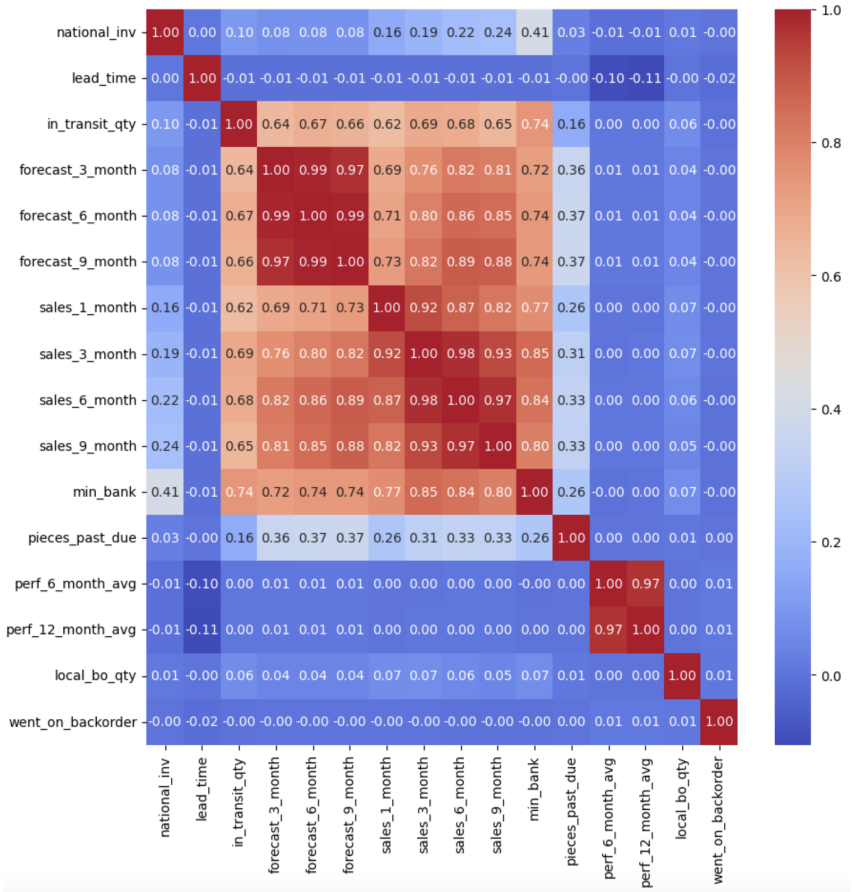


Figure 4: The Correlation HeatMap

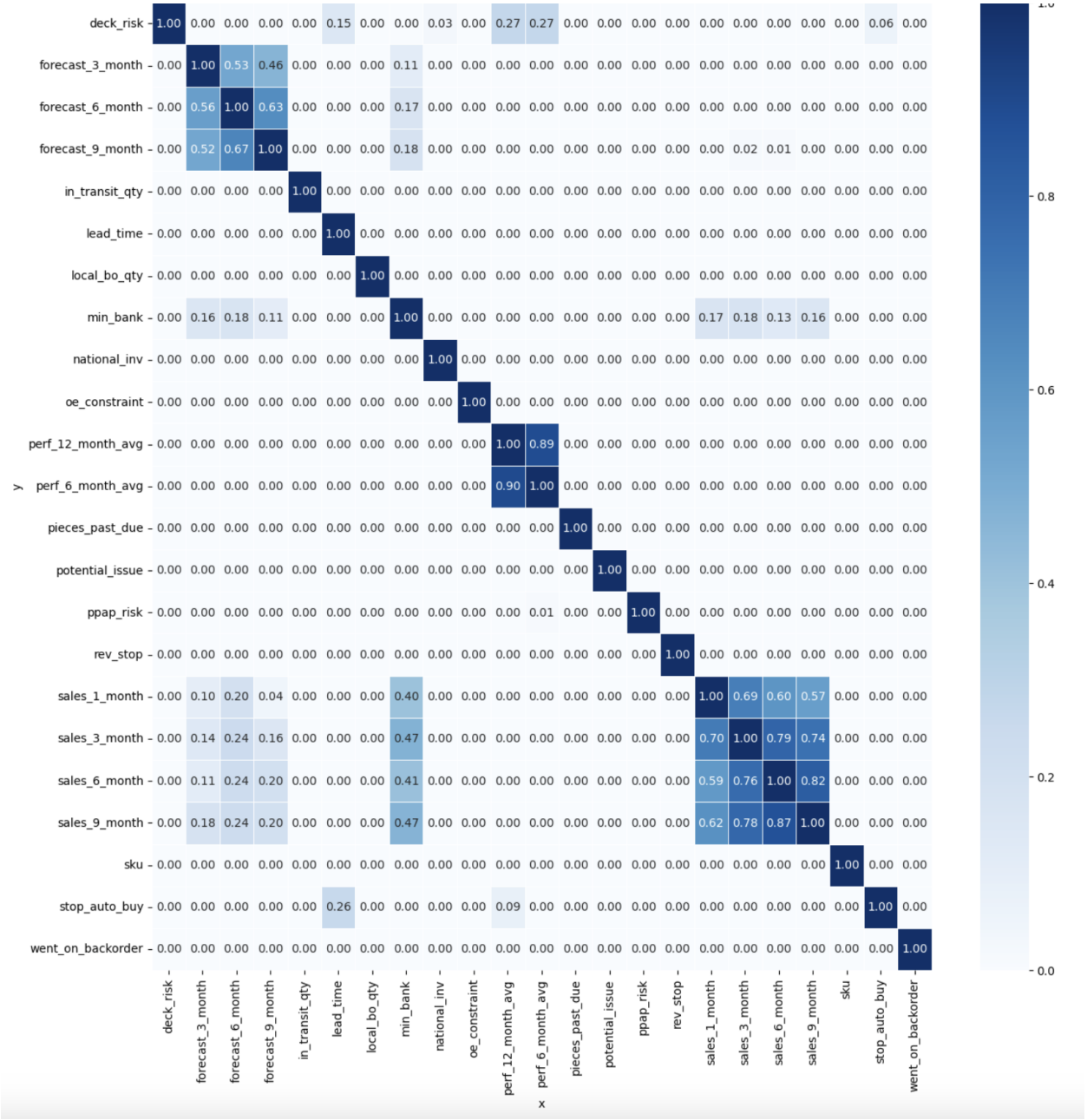


Figure 5: The HeatMap of Power Predictive Score

- Class Imbalance

The above exploratory data analysis has concluded that the dataset is extremely right-skewed. Almost all the numerical columns had extreme skewness on the positive side indicating them as outliers. The data is highly imbalanced as we can see that the majority class is for the went_on_backorder = No, while the minority class: "Yes" for went_on_backorder is negligible.

When comparing the other categorical features with the target variable ‘went_on_backorder’, it has been found many variables don't seem useful as they have an equal proportion of the classes and some have less impact with the backorder. In order to eradicate the issue, we have introduced an undersampling technique to balance the classes. Similarly, to handle the positive skewness(potential outliers) in data, we have used Robust Scaling to standardize the data as it eliminates the outliers while scaling the data.

5.2 Features Selection

From the correlation matrix, there is no correlation with the target features. The features to be dropped are as follows:

- 'forecast_3_month',
- 'forecast_6_month',
- 'forecast_9_month',
- 'sku'.

The high correlation coefficient of 1.0 among these forecast-related features implies possible redundancy or a linear prediction relationship. Linear models might not exhibit strong performance due to the presence of correlated features. Additionally, not all merchants may possess the capability to predict 'forecast_3_month', 'forecast_6_month', and 'forecast_9_month'. Furthermore, 'sku' represents the product ID and will be removed as well.

5.3 Preprocessing

Several crucial steps are carried out in the data preprocessing process for modeling. First, an Undersampling technique is carried out in order to balance the classes. Then, feature scaling is done with standardization techniques in order to ensure that numerical variables are on a common scale. Secondly, we convert categorical variables into a numerical representation that machine learning algorithms can comprehend, feature encoding is required. For this, we have used label encoding based on the data. Then, feature selection uses methods such as feature importance ranking or domain expertise to determine which variables are most pertinent to the particular predictive task. Last but not least, data splitting is an essential stage that involves

splitting the dataset into training and testing sets (such as an 80-20 or 70-30 split) to assess the model's performance and make sure it can adapt to new situations.

- Data Splitting - Due to a large dataset of backorder, we've chosen to perform a 70:30 data split.
- Undersampling - We have balanced the dataset using undersampling techniques in the training data to prevent the model from being biased towards the majority class and balance the class distribution in the training data.
- Imputation - We have performed imputation to handle missing and special values (-99.0) in your dataset, specifically for the features 'lead_time,' 'perf_6_month_avg,' 'perf_12_month_avg,' and 'national_inv.' Here's a summary of the imputation process:
    - We're using the `SimpleImputer` from scikit-learn with the ' `strategy='most_frequent'` to fill missing values (NaN) in the 'lead_time' feature.
    - Observing the outliers percentage we The high outliers For the 'perf_6_month_avg', 'perf_12_month_avg', and 'national_inv' features, you're also using the `SimpleImputer` with the 'most_frequent' strategy to replace the special value -99.0 with the most frequent value.
- Standardization and Scaling - We have applied Min-Max and Robust Scaling for the feature importance to the set of numerical features in our backorder dataset and follow the below steps:

- We first identified and selected the numerical features for scaling. We then applied the Min-Max scaling to both the training and testing datasets (`X_train` and `X_test`) by using MinMaxScaler ensuring that the features are scaled between 0 and 1, preserving the relative relationships among data points. Similarly, RobustScaler is employed to normalize the features and is fitted to the training set using the fit_transform method, capturing the necessary scaling parameters. Subsequently, the testing set is transformed using the learned parameters. RobustScaler is advantageous in handling outliers, as it scales data based on the interquartile range, making it robust to extreme values.

- Encoding - We encoded the categorical features to facilitate their integration into a machine learning pipeline excluding the target variable ('went_on_backorder'), and compiled them into the list `encode_cols` for subsequent encoding operations. Then, we

used the `OrdinalEncoder` to perform label encoding on each categorical feature within `encode_cols`. This systematic encoding process is applied uniformly to both the training (`X_train`) and testing (`X_test`) datasets.

To streamline and automate the encoding workflow, a `ColumnTransformer` named `encoder` is established. This transformer is configured to handle the encoding of categorical features using the `OrdinalEncoder`, while the `remainder='passthrough' ensures that non-categorical features retain their original values without undergoing transformation.

Pipeline - We have created a data preprocessing pipeline, combining several steps of 'imputation',' scaling', and 'encoding' into a single pipeline. This preprocessing pipeline allows us to prepare our data for machine learning, ensuring that it is in a suitable format for model training and evaluation. The pipeline process is employed to consolidate all of the preprocessing and modeling steps into a single workflow. This pipeline facilitates the ease of maintenance, reproducibility, and deployment of the model. It also helps prevent data leakage, which can occur accidentally during preprocessing.

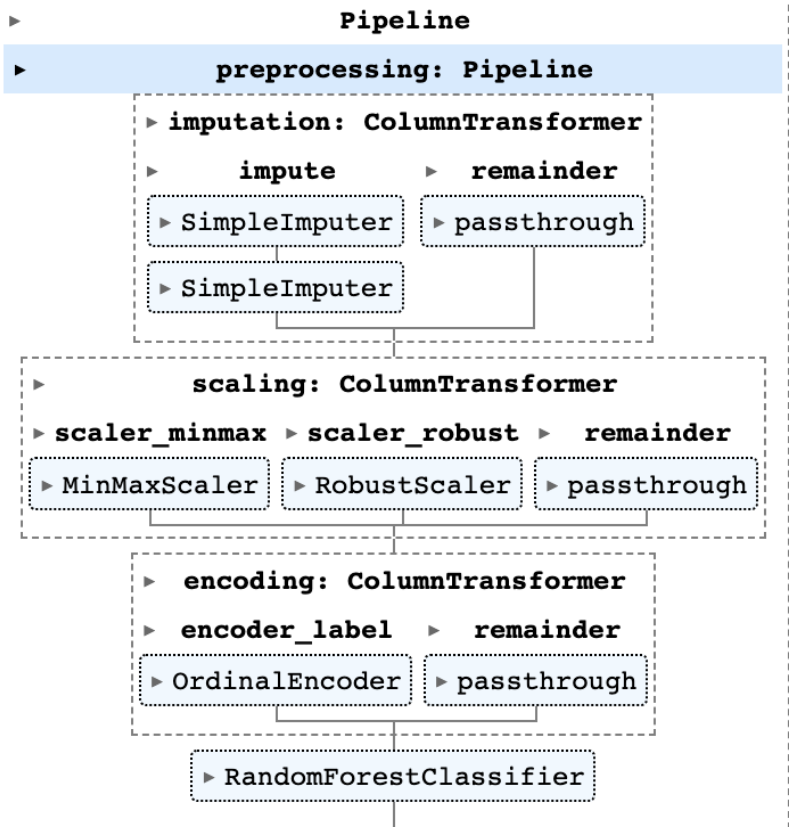


Figure 6: The Fitted Pipeline

5.4 Machine Learning Training and Testing

We have experimented with multiple models (5 models) to address the complexities and find which models will give better results in terms of accuracy, f1-score, and recall and performed the following steps:

- Cross-Validation: Cross-validation is employed to assess the performance of various machine learning algorithms on a training dataset which partitions the dataset into multiple folds to iteratively train and evaluate the model on different combinations of training and validation sets. The choice of algorithms includes Logistic Regression, Gaussian Naive Bayes, Random Forest Classifier, Support Vector Classifier (SVC), and Gradient Boosting Classifier.

For each algorithm, several performance metrics are introduced across the five folds:

1. Accuracy: Represents the overall correctness of the model's predictions. The mean accuracy and standard deviation are printed, providing insights into the consistency of model performance across different folds.
2. F1 Score: Balances precision and recall, offering a harmonic mean between the two. The mean F1 score and its standard deviation are reported.
3. Recall: Emphasizes the model's ability to capture all positive instances. Mean recall and standard deviation are printed, indicating the model's consistency in identifying positive cases.
4. Precision: Focuses on the correctness of positive predictions. The mean precision and its standard deviation are displayed.

For our backorder datasets, in terms of all the pre-defined metrics, the Random Forest Classifier stands out with higher mean accuracy, suggesting their effectiveness in this classification task.

- GridSearchCV - We have used cross-validation along with grid search to tune hyperparameters for the RandomForestClassifier. The grid search is conducted using five-fold cross-validation and multiple scoring metrics, including accuracy, precision, recall, F1 score, and ROC AUC. The best parameters for the RandomForestClassifier are determined based on optimizing the F1 score.

The results of the grid search indicate the best parameters as {'max_depth': 20, 'n_estimators': 200}. The cross-validation scores for various metrics (accuracy, precision, recall, F1 score, and ROC AUC) are also displayed for each combination of hyperparameters, and the model achieves an F1 score of 0.896 on the test set. The entire process is encapsulated in a pipeline, including preprocessing steps and the RandomForestClassifier with the selected hyperparameters.

- Training and Testing Model on Pipeline

The machine learning model is trained using a pipeline that includes preprocessing steps and a RandomForestClassifier with previously tuned hyperparameters. The model is trained on the training set (`X_train1`, `y_train1`), and predictions are made on the test set (`X_test1`). The accuracy of the model on the test set is then computed. In this specific case, the model achieves an accuracy of 87.58%, indicating that approximately 87.58% of the predictions on the test set are correct. This accuracy score provides an overall assessment of the model's performance in terms of correct predictions on the backorder dataset.

5.5 Evaluation

The reliability of the F1 score, precision, and recall values was compromised due to the imbalanced nature of the dataset. Therefore, we have introduced the confusion matrix and ROC-AUC score for comparison. Since we have an imbalanced distribution of classes, but we resampled the training data and when comparing the result of the confusion matrix and ROC-AUC score, RF exhibited the highest accuracy. This approach provides a comprehensive comparison, considering both accuracy and ROC-AUC score, which is particularly relevant for imbalanced datasets.

- Confusion Matrix: The confusion matrix evaluates the model's performance on test sets by calculating precision, recall, and F1-score. In the evaluation of the test set, the model demonstrates high recall (91%), low precision (5%), and low F1-score (10%), indicating that the model has high performance in correctly identifying the target class (1, ‘Yes’).
- This issue may stem from the imbalanced target features, as evidenced by the low F1 score. Possible solutions include feature engineering, tree-based feature selection, and selecting some more strategies for sampling techniques. However, the high recall and low

precision results are still satisfactory due to the objective of backorder prediction, which involves rare cases.

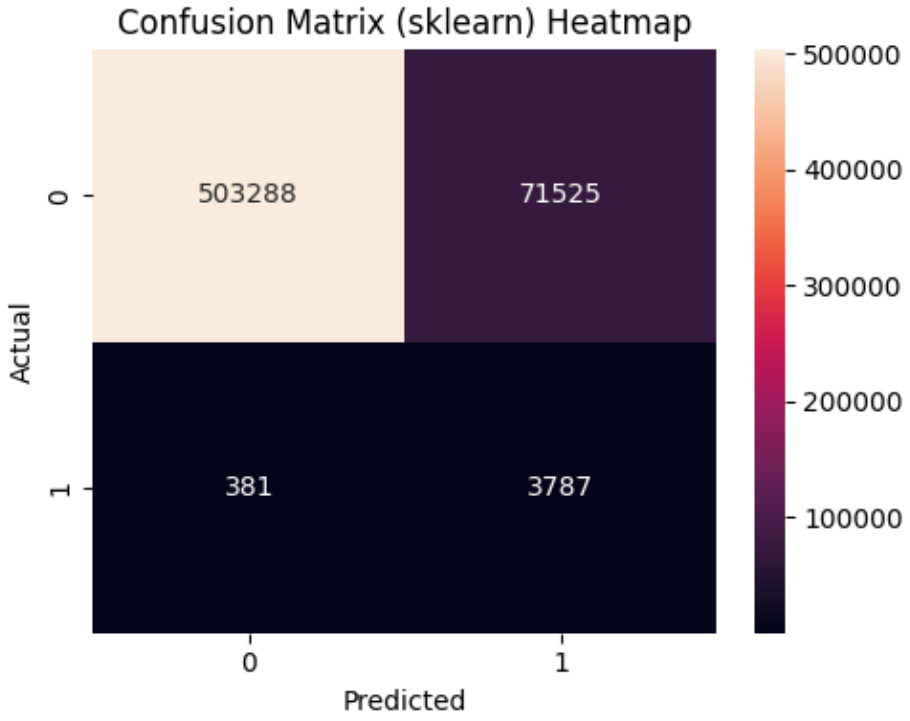


Figure 7: Confusion Matrix

- ROC Curve: The ROC Area Under the Curve (AUC) served as the validation metric due to the significant class imbalance. With an AUC value of 0.95, which exceeds 0.5, it signifies that the predictive performance is substantially better than random guessing. By looking at ROC curves, we can determine a cutoff threshold for classification after fitting the models, rather than naively assuming a threshold of 0.5.

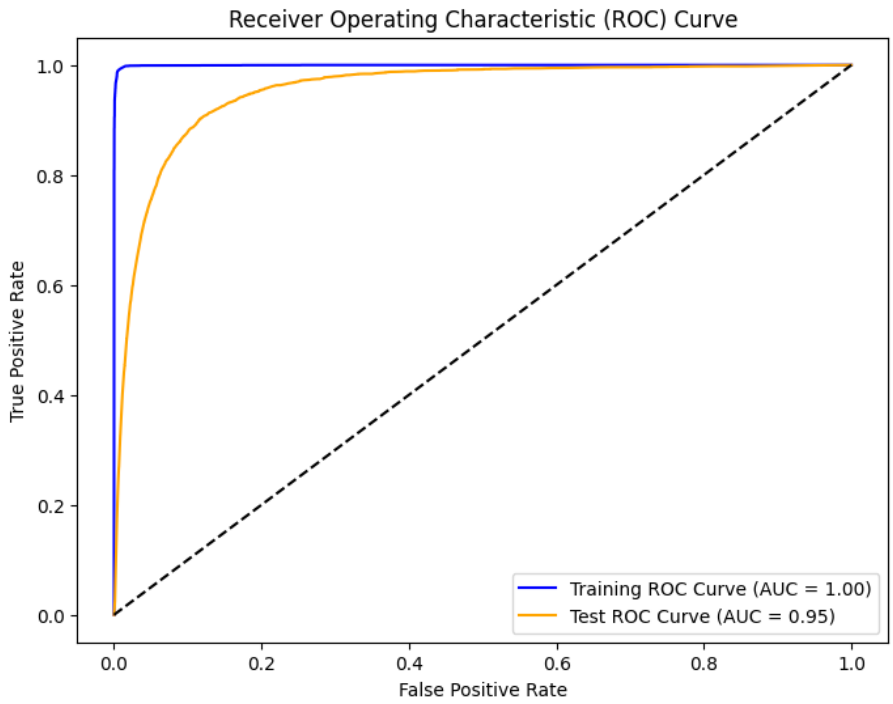


Figure 8: ROC Curve

- Feature Importance: The feature importances are determined from the best estimator. The importance values are sorted in descending order, providing insights into the significance of each feature. The feature ranking reveals that "national_inv" holds the highest importance, followed by "sales_3_month," "sales_9_month," "sales_6_month," and other relevant features. The importance values indicate the contribution of each feature to the model's predictive performance, aiding in feature selection and interpretation.
- The 'national_inv' feature holds the highest importance because it directly correlates with the current inventory level for the product. This factor significantly influences the likelihood of a backorder. A lower current inventory level tends to increase the probability of a product becoming backorder. Additionally, features ranked 2nd to 4th are associated with the sales quantity of the product. These metrics directly reflect the demand for the product. When demand exceeds availability, it often leads to backorders.

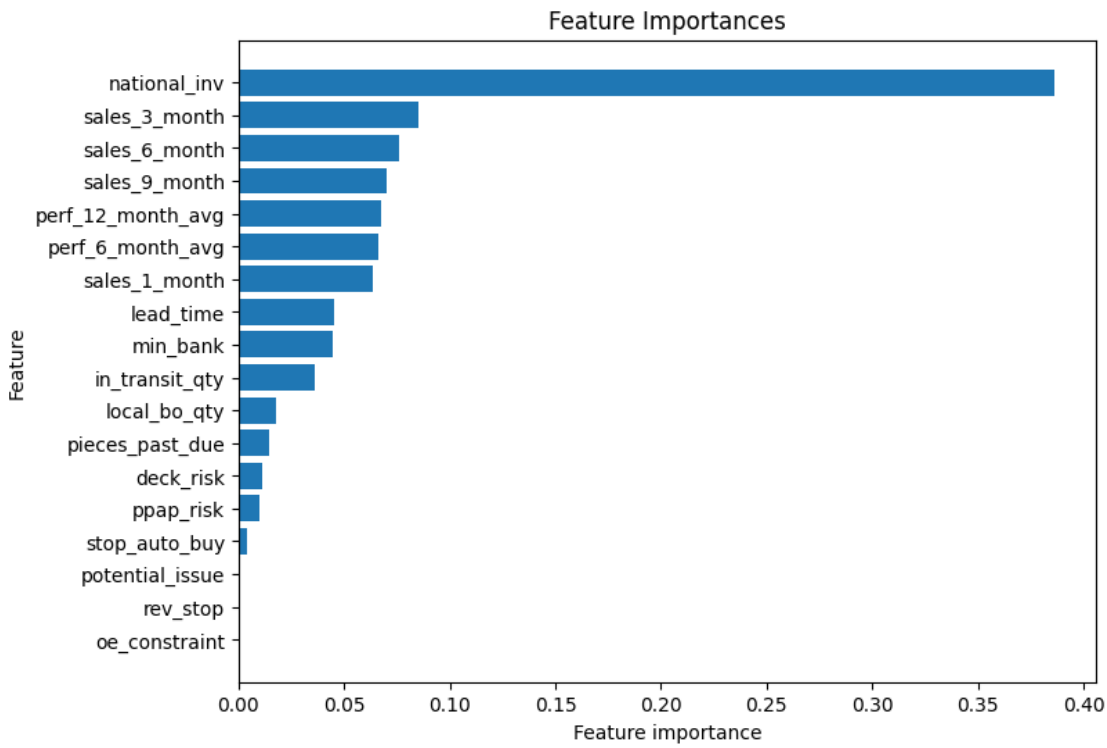


Figure 9: Feature Importance

5.6 Deployment

- Inference

The pipeline, including the preprocessing steps and the RandomForestClassifier model, is saved as a joblib file named 'pipeline_model.joblib.' Subsequently, the saved pipeline is loaded, and its performance is evaluated on the test set, confirming that the accuracy in inference matches the original model's accuracy. Additionally, a single sample is extracted from the test set and converted into a DataFrame. The loaded pipeline is then used to make predictions on this single sample, and the result is presented as the potential backorder status, indicating whether the product is predicted to be a backorder ('Yes') or not ('No'). This process demonstrates the successful saving, loading, and application of the machine learning pipeline for making predictions on new data.

- Web application

For the deployment of our application, we used Dash for the frontend providing an interactive and visually appealing user interface. Flask was chosen for the backend, for a flexible framework to handle server-side operations and logic. The deployment itself was hosted on an AWS server, leveraging the power and reliability of Amazon Web Services to create and deploy the web application.

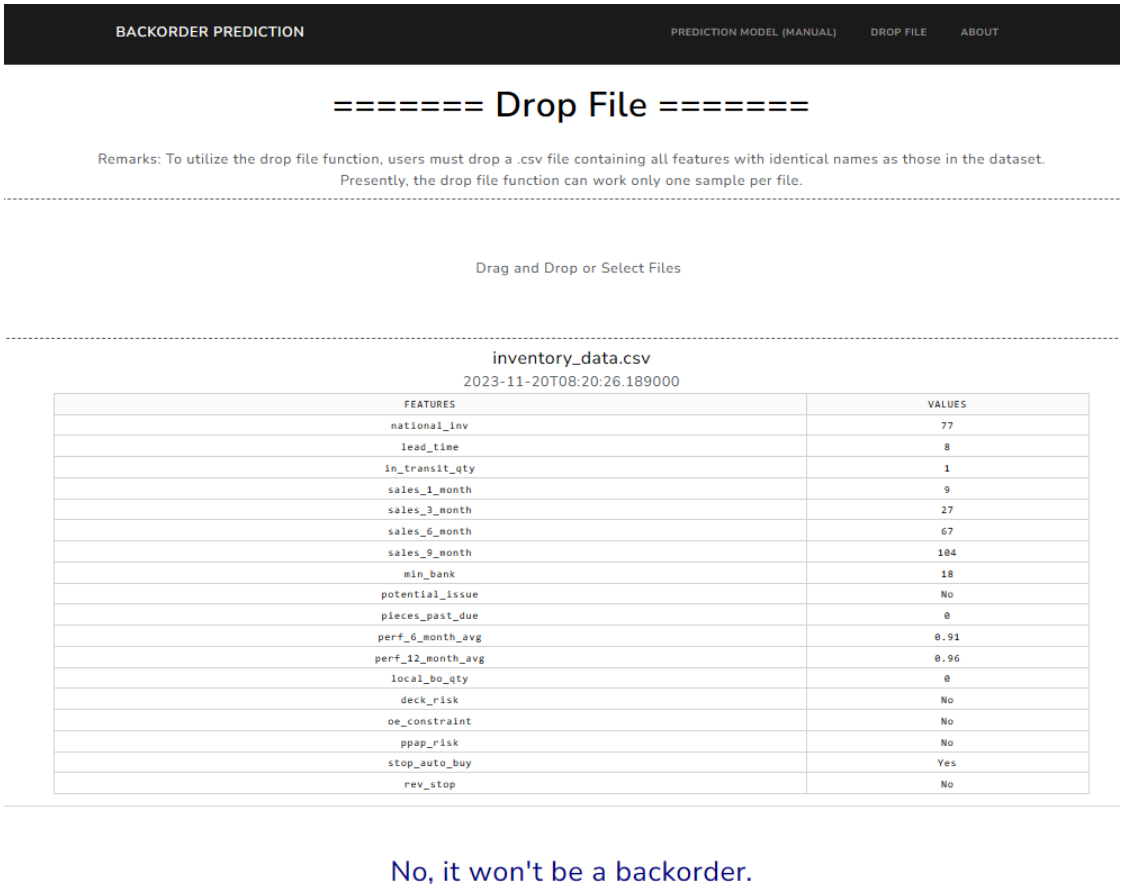


Figure 10: Example of Web Applications

6. RESULTS AND DISCUSSIONS

In the preliminary results, the dataset was utilized to conduct exploratory data analysis, perform basic preprocessing, and make simple model predictions. These preliminary results were achieved using Jupyter Notebook with the Python programming language. The steps for the preliminary results are as follows.

1. Import Libraries:

   In the first step, the libraries that were used to perform the exploratory data analysis, preprocessing, pipelining, and prediction (as illustrated in Figure 11). The main libraries are pandas, numpy, sklearn, seaborn, ppscore, and matplotlib.

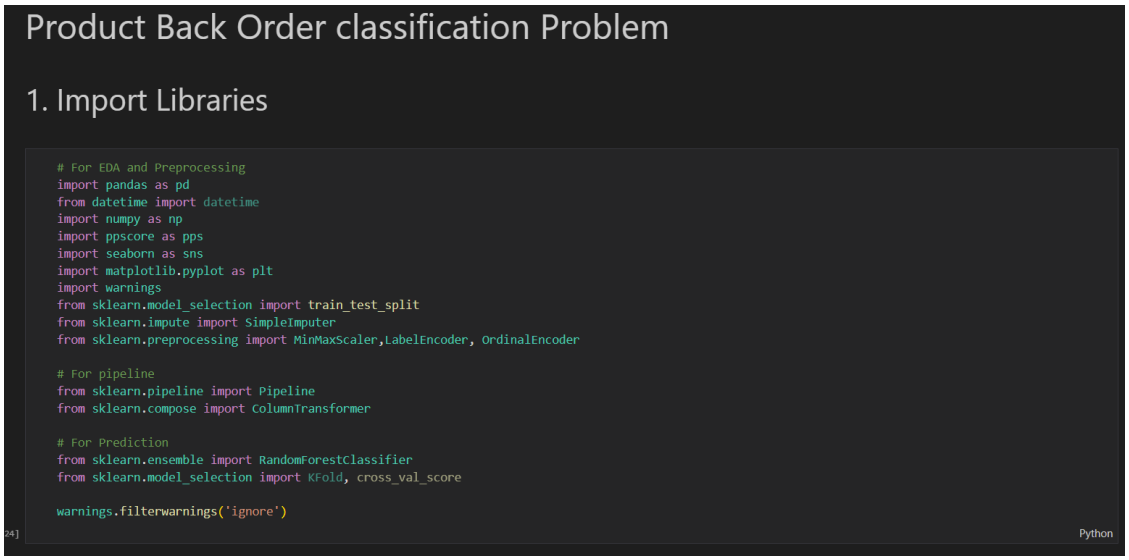


Figure 11: The imported libraries that used to perform the preliminary result.

2. Loading Dataset:

   The two files of the dataset are loaded as the Pandas DataFrame using Pandas functions and then combined for exploratory data analysis (as illustrated in Figure 12).

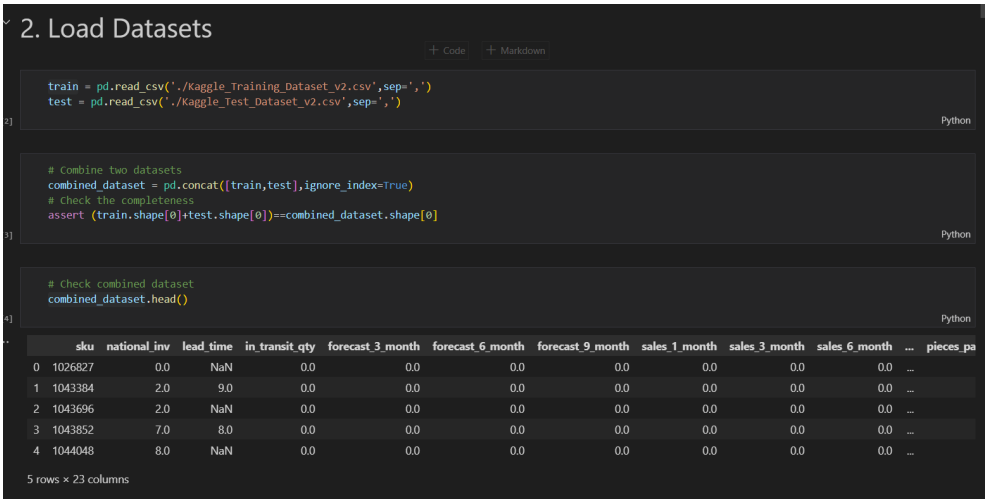


Figure 12: Loaded and Combined datasets

3. Exploratory Data Analysis (EDA)

- Dataset Considerations: The dataset is inspected in terms of its shape, data types, column names, and statistics. Numerical and categorical columns for later use are then determined. During the statistical inspection (using the describe() function), it's observed that the features 'perf_6_month_avg' and 'perf_12_month_avg' have missing values, which appear to be filled with -99.0. Furthermore, two rows in the dataset contain missing values, and they are dropped. Finally, the target feature is encoded using Label Encoder for use in both Univariate and Multivariate analyses. The illustrations for Dataset Considerations range from Figure 13 to Figure 21.

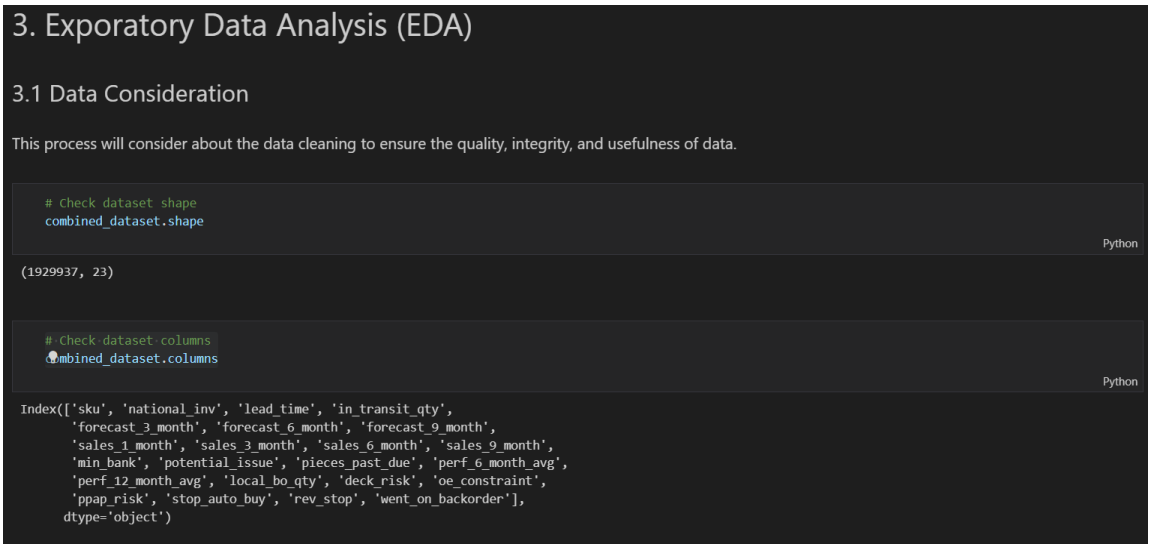


Figure 13: Shape and Columns checking

```python
# Check dataset features type
combined_dataset.dtypes
```

Python

```
sku                 object
national_inv       float64
lead_time          float64
in_transit_qty     float64
forecast_3_month   float64
forecast_6_month   float64
forecast_9_month   float64
sales_1_month      float64
sales_3_month      float64
sales_6_month      float64
sales_9_month      float64
min_bank           float64
potential_issue     object
pieces_past_due    float64
perf_6_month_avg   float64
perf_12_month_avg  float64
local_bo_qty       float64
deck_risk           object
oe_constraint       object
ppap_risk           object
stop_auto_buy       object
rev_stop            object
went_on_backorder   object
dtype: object
```

Figure 14: Check the dataset's features type

```python
# Checking numerical and categorical features
numerical_cols = []
categorical_cols = []

for column in combined_dataset.columns:
    if pd.api.types.is_numeric_dtype(combined_dataset[column]):
        numerical_cols.append(column)
    else:
        categorical_cols.append(column)

print(f'Numerical Features: {len(numerical_cols)} ,{numerical_cols}')
print(f'Categorical Features: {len(categorical_cols)} ,{categorical_cols}')
```

✓ 0.0s Python

```
Numerical Features: 15 ,['national_inv', 'lead_time', 'in_transit_qty', 'forecast_3_month', 'forecast_6_month', 'forecast_9_month', 'sales_1_month', 'sales
Categorical Features: 8 ,['sku', 'potential_issue', 'deck_risk', 'oe_constraint', 'ppap_risk', 'stop_auto_buy', 'rev_stop', 'went_on_backorder']
```

```python
# Redefined categorial features (we not used 'sku' which is product ID features)
categorical_cols.remove('sku')
```

✓ 0.0s Python

Figure 15: Separated the numerical and categorical features

```python
# Describe the combined dataset
combined_dataset.describe()
```

Python

	sales_1_month	sales_3_month	sales_6_month	sales_9_month	min_bank	pieces_past_due	perf_6_month_avg	perf_12_month_avg	local_bo_qty	went_on_backorder
5	1.929935e+06	1.929935e+06	1.929935e+06	1.929935e+06	1.929935e+06	1.929935e+06	1.929935e+06	1.929935e+06	1.929935e+06	1.929937e+06
2	5.536816e+01	1.746639e+02	3.415653e+02	5.235771e+02	5.277637e+01	2.016193e+00	-6.899870e+00	-6.462343e+00	6.537039e-01	7.246351e-03
1	1.884377e+03	5.188856e+03	9.585030e+03	1.473327e+04	1.257968e+03	2.296112e+02	2.659988e+01	2.588343e+01	3.543230e+01	8.482875e-02
)	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	-9.900000e+01	-9.900000e+01	0.000000e+00	0.000000e+00
)	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	6.300000e-01	6.600000e-01	0.000000e+00	0.000000e+00
)	0.000000e+00	1.000000e+00	2.000000e+00	4.000000e+00	0.000000e+00	0.000000e+00	8.200000e-01	8.100000e-01	0.000000e+00	0.000000e+00
1	4.000000e+00	1.500000e+01	3.100000e+01	4.700000e+01	3.000000e+00	0.000000e+00	9.600000e-01	9.500000e-01	0.000000e+00	0.000000e+00
5	7.417740e+05	1.105478e+06	2.146625e+06	3.205172e+06	3.133190e+05	1.464960e+05	1.000000e+00	1.000000e+00	1.253000e+04	2.000000e+00

Features 'perf_6_month_avg' and 'perf_12_month_avg' have the -99.0 which seem to be missing values and filled with -99.0

Figure 16: Statistics details in each feature of the datasets

```
# Counts -99.0 in feature 'perf_6_month_avg'
combined_dataset['perf_6_month_avg'].value_counts()
```

Python

```
 0.99     163323
 1.00     150339
-99.00    148579
 0.73     128818
 0.98      97390
           ...
 0.20        921
 0.03        829
 0.04        724
 0.01        648
 0.29        572
Name: perf_6_month_avg, Length: 102, dtype: int64
```

```
# Counts -99.0 in feature 'perf_12_month_avg'
combined_dataset['perf_12_month_avg'].value_counts()
```

Python

```
 0.99     152682
-99.00    140025
 0.78     131353
 0.98     106119
 0.97      74113
           ...
 0.23        895
 0.06        873
 0.05        743
 0.03        639
 0.02        437
Name: perf_12_month_avg, Length: 102, dtype: int64
```

Figure 17: Checking the -99.0 values in feature ‘perf_6_month_avg’ and ‘perf_12_month_avg’

```
# Check Missing Values in combined_dataset
combined_dataset.isna().sum()
```

✓ 1.1s

Python

```
sku                     0
national_inv            2
lead_time          115619
in_transit_qty          2
forecast_3_month        2
forecast_6_month        2
forecast_9_month        2
sales_1_month           2
sales_3_month           2
sales_6_month           2
sales_9_month           2
min_bank                2
potential_issue         2
pieces_past_due         2
perf_6_month_avg        2
perf_12_month_avg       2
local_bo_qty            2
deck_risk               2
oe_constraint           2
ppap_risk               2
stop_auto_buy           2
rev_stop                2
went_on_backorder       2
dtype: int64
```

There might be 2 rows that contains all missing values. Let's check that..

Figure 18: Checking the number of missing values in each feature.

```
# Check for all missing rows
combined_dataset[combined_dataset['went_on_backorder'].isna()]
```

✓ 0.1s Python

	sku	national_inv	lead_time	in_transit_qty	forecast_3_month	forecast_6_month	forecast_9_month	sales_1_month	sales_3_month	sales_6_month	...	pi
1687860	(1687860 rows)	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	
1929936	(242075 rows)	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	

2 rows × 23 columns

Figure 19: Checking the rows that contain missing values.

```
# Drop 2 missing rows
combined_dataset.drop(combined_dataset[combined_dataset['sku'] == '(1687860 rows)'].index,inplace=True)
combined_dataset.drop(combined_dataset[combined_dataset['sku'] == '(242075 rows)'].index,inplace=True)
```

✓ 0.6s Python

```
# Recheck missing rows again
combined_dataset.isna().sum()
```

Python

```
sku                   0
national_inv          0
lead_time        115617
in_transit_qty        0
forecast_3_month      0
forecast_6_month      0
forecast_9_month      0
sales_1_month         0
sales_3_month         0
sales_6_month         0
sales_9_month         0
min_bank              0
potential_issue       0
pieces_past_due       0
perf_6_month_avg      0
perf_12_month_avg     0
local_bo_qty          0
deck_risk             0
oe_constraint         0
ppap_risk             0
stop_auto_buy         0
rev_stop              0
went_on_backorder     0
dtype: int64
```

Figure 20: Drop the rows that contain missing values and recheck again.

```
# LabelEncoder the target features
target_col = LabelEncoder().fit_transform(combined_dataset['went_on_backorder'])
combined_dataset['went_on_backorder']= target_col
```

Python

Figure 21: Label encoding the target feature for later on analysis.

- Univariate Analysis and Multivariate Analysis:

  The Kernel Density Estimate plots (KDE plots) and Count plots are used to conduct univariate analysis. For multivariate analysis, a correlation heatmap, power predictive score heatmap, and outliers function are created. The corresponding codes are provided in figures, ranging from Figure 22 to Figure 28.

3.2 Univariate Analysis

```python
# Check the skrew of data
# Create subplots for kde plots
fig, axes = plt.subplots(nrows=5, ncols=3, figsize=(10, 10))
fig.tight_layout()

# Iterate through the list and create kde plots
for i, k in enumerate(numerical_cols):
    row, col = divmod(i, 3)
    sns.kdeplot(data = combined_dataset,x=k, fill=True, ax=axes[row, col])
    axes[row, col].set_title(k)
plt.subplots_adjust(hspace=0.6)
plt.show()
```

Python

Figure 22: Plotting the numerical features as the KDE plots.

```python
# Countplot for categorical features
fig, axes = plt.subplots(nrows=4, ncols=2, figsize=(8, 12))
fig.tight_layout()

# Iterate through the list of categorical features and create count plots
for i, feature in enumerate(categorical_cols):
    row, col = divmod(i, 2)
    ax = sns.countplot(data=combined_dataset, x=feature, ax=axes[row, col])
    # Add count annotations above each bar
    for p in ax.patches:
        ax.annotate(f'{p.get_height()}', (p.get_x() + p.get_width() / 2., p.get_height()), ha='center', va='bottom')
    axes[row, col].set_title(f'Count Plot of {feature}')
plt.subplots_adjust(hspace=0.6)
plt.show()
```

Python

Figure 23: Plotting the categorical features as the count plots.

```python
# Create subplots for box plots
fig, axes = plt.subplots(nrows=5, ncols=3, figsize=(15, 15))
fig.tight_layout()

# Iterate through numerical columns and create box plots within the percentile range
for i, k in enumerate(numerical_cols):
    row, col = divmod(i, 3)
    zero_percentile_k = combined_dataset[k].quantile(0)
    ninety_percentile_k = combined_dataset[k].quantile(0.9)

    # Filter 'k' column based on its own percentile range
    filtered_data_k = combined_dataset[(combined_dataset[k] >= zero_percentile_k) &
                                       (combined_dataset[k] <= ninety_percentile_k)]

    sns.boxplot(x='went_on_backorder', y=k, data=filtered_data_k, ax=axes[row, col])
    axes[row, col].set_title(k)
plt.subplots_adjust(hspace=0.6)
plt.show()
```

Python

Figure 24: Plotting the features as the box plot.

3.3 Multivariate Analysis

```python
#Calculate the outliers
def outlier_count(col, data):

    # calculate your 25% quatile and 75% quatile
    q75, q25 = np.percentile(data[col], [75, 25])

    # calculate your inter quatile
    iqr = q75 - q25

    # min_val and max_val
    min_val = q25 - (iqr*1.5)
    max_val = q75 + (iqr*1.5)

    # count number of outliers, which are the data that are less than min_val or more than max_val calculated above
    outlier_count = len(np.where((data[col] > max_val) | (data[col] < min_val))[0])

    # calculate the percentage of the outliers
    outlier_percent = round(outlier_count/len(data[col])*100, 2)

    if(outlier_count > 0):
        print("\n"+15*'-' + col + 15*'-'+"\n")
        print('Number of outliers: {}'.format(outlier_count))
        print('Percent of data that is outlier: {}%'.format(outlier_percent))
```

Python

```python
#Loop the outliers function to find the percent of data that is outliers
for col in combined_dataset[numerical_cols]:
    outlier_count(col,combined_dataset)
```

Python

Figure 25: Defining outliers percentage function for measuring the percentages of outliers in numerical features.

```
...
---------------national_inv---------------

Number of outliers: 290377
Percent of data that is outlier: 15.05%
90 percentile value is 360.0
91 percentile value is 417.0
92 percentile value is 490.0
93 percentile value is 577.0
94 percentile value is 722.0
95 percentile value is 924.0
96 percentile value is 1225.0
97 percentile value is 1821.0
98 percentile value is 3015.0
99 percentile value is 5487.0
100 percentile value is 12334404.0
99.0percentile value is 5487.0
99.1percentile value is 6102.0
99.2percentile value is 6910.0
99.3percentile value is 7984.0
99.4percentile value is 9360.0
99.5percentile value is 11079.0
99.6percentile value is 13650.0
99.7percentile value is 17309.0
99.8percentile value is 24574.0
...
99.7percentile value is 18.0
99.8percentile value is 35.0
99.9percentile value is 99.0
100 percentile value is  12530.0
```

Figure 26: The result examples of the percentages of outliers in each numerical features.

```python
# Correlation Matrix
plt.figure(figsize=(10,10))
sns.heatmap(combined_dataset.corr(), fmt=".2f", annot=True, cmap="coolwarm")
```

Python

Figure 27: Plotting the heatmap of the correlation matrix

```python
# Predictive Power Score
#Create another dataframe to used with pps (just in case that adjusting the features for this)
df_pps=combined_dataset.copy()
matrix_df_pps = pps.matrix(df_pps)[['x', 'y', 'ppscore']].pivot(columns='x', index='y', values='ppscore')
plt.figure(figsize=(15,15))
sns.heatmap(matrix_df_pps, vmin=0, vmax=1, fmt=".2f", cmap="Blues", linewidths=0.5, annot=True)
```

Python

Figure 28: Plotting the heatmap of the predictive power score

4. Preprocessing

- Resampling

The dataset : combined_dataset is divided into two subsets based on the values of the target variable went_on_backorder. The majority class (orders that did not go on backorder) is stored in majority_class, and the minority class (orders that did go on backorder) is stored in minority_class. The majority class is further split into training and testing sets using train_test_split. Random undersampling is applied to the training set of the majority class (majority_train). This process involves randomly removing instances from the majority class until its size matches that of the minority class. This is done to balance the class distribution in the training set and mitigate the effects of class imbalance. The undersampled majority class and the original minority class are concatenated to create a new dataset (undersampled_dataset).

```python
# Separate majority and minority classes
majority_class = combined_dataset[combined_dataset['went_on_backorder'] == 0]
minority_class = (variable) majority_test: Any set['went_on_backorder'] == 1]

majority_train, majority_test = train_test_split(majority_class, test_size=0.2, random_state

# Undersampling majority class in the training set
majority_train_undersampled = resample(majority_train, replace=False, n_samples=len(minority
undersampled_dataset = pd.concat([majority_train_undersampled, minority_class])
undersampled_dataset = undersampled_dataset.reset_index(drop=True)

# Check the new class distribution
undersampled_dataset['went_on_backorder'].value_counts()
```

Fig 29:Undersampling the training datasets

- Train and Test Datasets Splitting:

The dataset is separated into X and y variables based on the target and non-target features. The train_test_split function is utilized to split the data into training and testing sets with a 70:30 ratio. Furthermore, the shapes of all the test and training datasets are verified. The code for this process is illustrated in Figure 30.

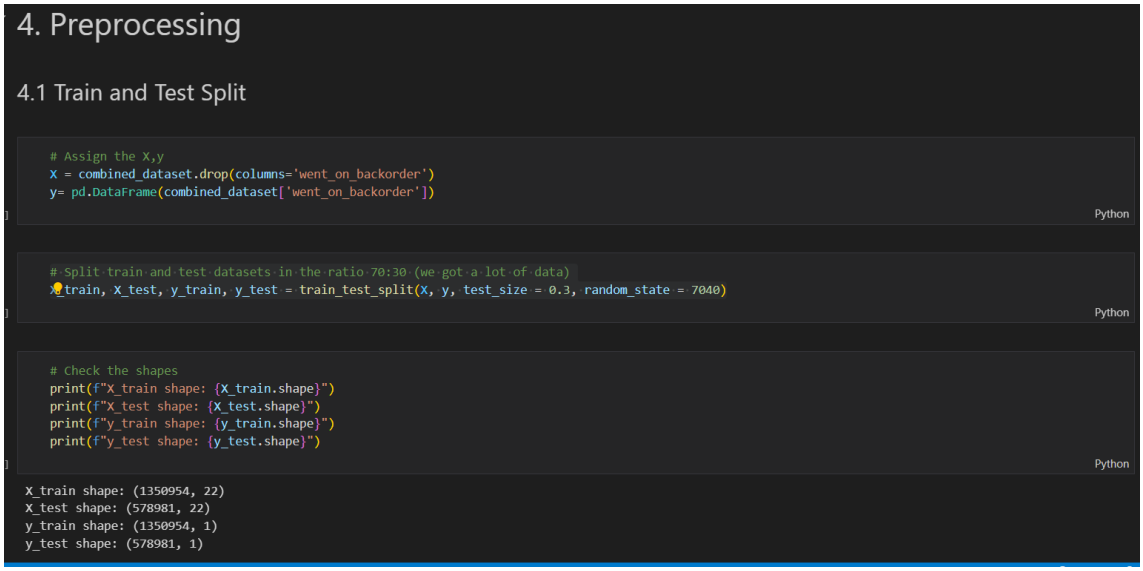


Figure 30: Splitting training and testing datasets then verifying their shape.

- Imputation:

Imputation begins by checking all missing values, including the value '-99.00'. The SimpleImputer function from the scikit-learn library is then used for imputation. After fitting and transforming the datasets, the validation of imputation is verified. Finally, a pipeline for imputations is created and contained within the columns transformer. The code for this process is shown in Figure 31 to Figure 36.

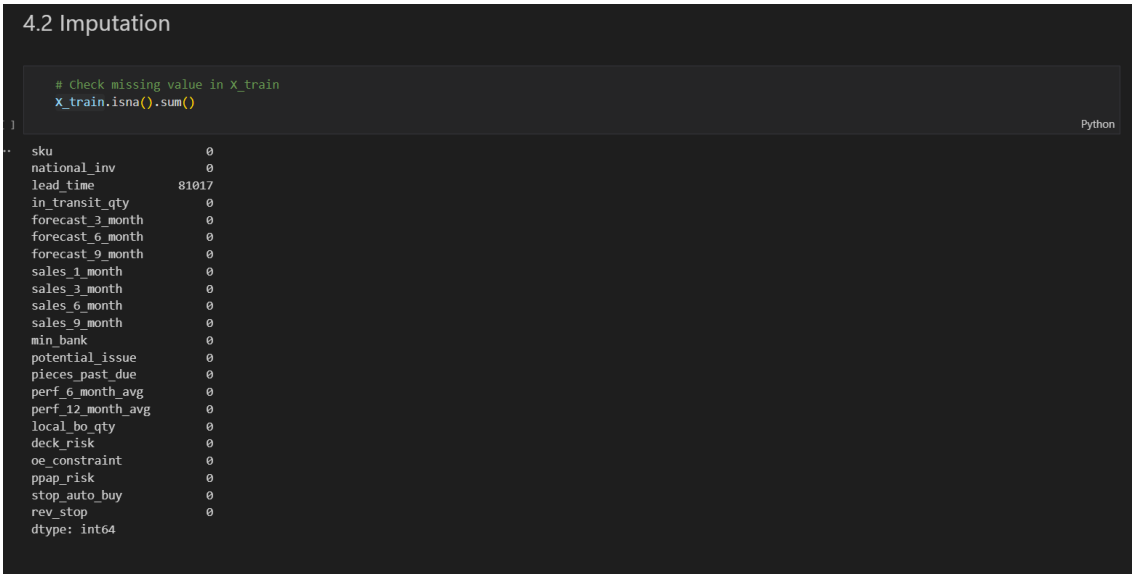


Figure 31: Checking missing values (NaN) in all features.

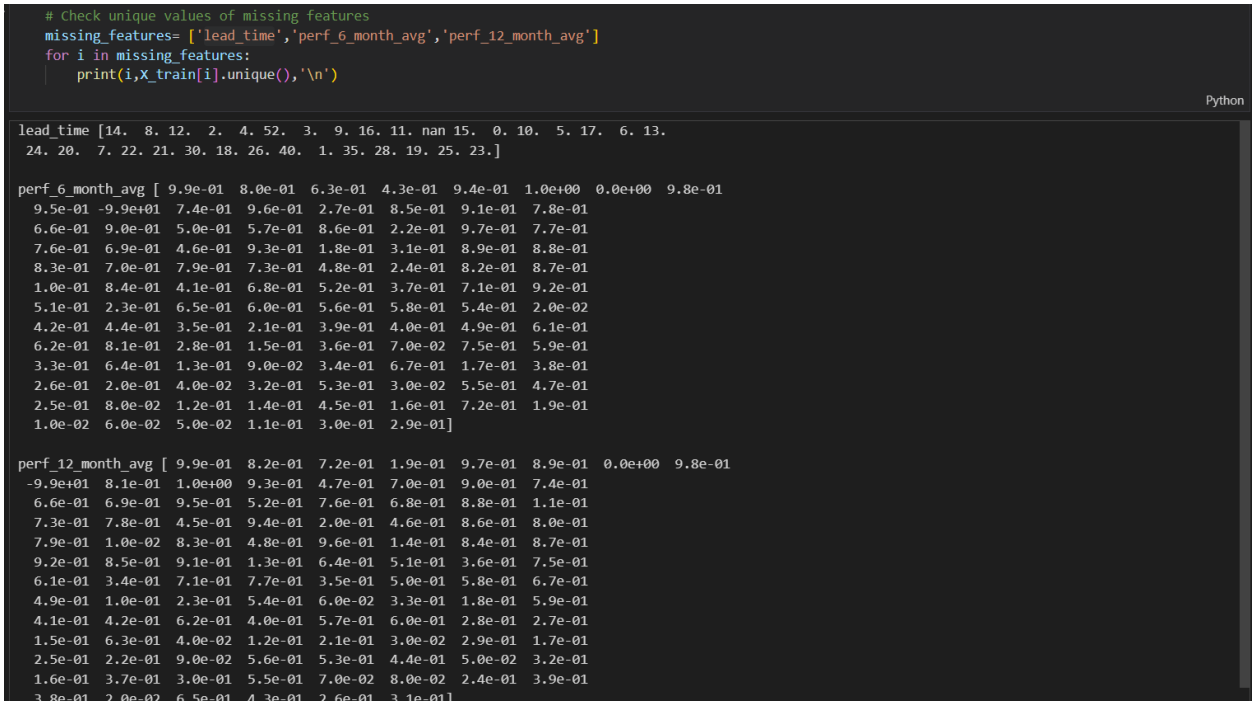


Figure 32: Checking the unique values to find the abnormal values.

Imputation:
'lead_time' = np.nan
'perf_6_month_avg' = -99.0
'perf_12_month_avg' = -99.0

```python
# There are any "-99.0" left in other features?
for feature in X_train.columns:
    unique_values = X_train[feature].unique()
    if -99.0 in unique_values:
        print(f"Feature {feature} contains -99.0 ")
```

```
Feature national_inv contains -99.0
Feature perf_6_month_avg contains -99.0
Feature perf_12_month_avg contains -99.0
```

Imputation(updated):
'lead_time' = np.nan
'perf_6_month_avg' = -99.0
'perf_12_month_avg' = -99.0
'national_inv' = -99.0

Figure 33: Rechecking the '-99.00' values in other features.

```python
# updated missing features
missing_features.append('national_inv')
missing_features
```

```
['lead_time', 'perf_6_month_avg', 'perf_12_month_avg', 'national_inv']
```

```python
# Create the imputers for nan and -99
imp_null = SimpleImputer(missing_values=np.nan, strategy= 'most_frequent')
imp_neg99 = SimpleImputer(missing_values= -99.0, strategy= 'most_frequent')
```

```python
# fit_transform imputers for X_train and transform imputers for X_test
X_train['lead_time']= imp_null.fit_transform(X_train[['lead_time']])
X_test['lead_time']= imp_null.transform(X_test[['lead_time']])

X_train['national_inv'] = imp_neg99.fit_transform(X_train[['national_inv']])
X_test['national_inv']= imp_neg99.transform(X_test[['national_inv']])

X_train['perf_6_month_avg'] = imp_neg99.fit_transform(X_train[['perf_6_month_avg']])
X_test['perf_6_month_avg']= imp_neg99.transform(X_test[['perf_6_month_avg']])

X_train['perf_12_month_avg'] = imp_neg99.fit_transform(X_train[['perf_12_month_avg']])
X_test['perf_12_month_avg']= imp_neg99.transform(X_test[['perf_12_month_avg']])
```

Figure 34: Create the SimpleImputer then fit or transform into datasets.

```python
# Check that imputer works
print(f"X_train lead_time: {X_train['lead_time'].isna().sum()}")
print(f"X_train national_inv: {len(X_train[X_train['national_inv'] == -99])}")
print(f"X_train perf_6_month_avg: {len(X_train[X_train['perf_6_month_avg'] == -99])}")
print(f"X_train perf_12_month_avg: {len(X_train[X_train['perf_12_month_avg'] == -99])}\n")

print(f"X_test lead_time: {X_train['lead_time'].isna().sum()}")
print(f"X_test national_inv: {len(X_train[X_train['national_inv'] == -99])}")
print(f"X_test perf_6_month_avg: {len(X_train[X_train['perf_6_month_avg'] == -99])}")
print(f"X_test perf_12_month_avg: {len(X_train[X_train['perf_12_month_avg'] == -99])}")
```

Python

```
X_train lead_time: 0
X_train national_inv: 0
X_train perf_6_month_avg: 0
X_train perf_12_month_avg: 0

X_test lead_time: 0
X_test national_inv: 0
X_test perf_6_month_avg: 0
X_test perf_12_month_avg: 0
```

Figure 35: Validate the imputation

```python
# Create Pipeline for imputation
impute = Pipeline(
    steps=[
        ('impute_null',imp_null),
        ('impute_(-99.0)',imp_neg99)
    ]
)
```

Python

```python
# Create ColumnTransformer for imputation
imputer = ColumnTransformer(
    transformers=[
        ("impute", impute, missing_features)
    ], remainder='passthrough', verbose_feature_names_out=False
).set_output(transform='pandas')
```

Python

Figure 36: Create the Pipeline for imputation.

- Standardization and Scaling:

The MinMax and RobustScaler scaler is the chosen scaling method. After its creation, it is applied to fit and transform only the numerical features. The scaling is then validated, and a pipeline for standardization and scaling is established. The code for this process is presented in Figure 37 to Figure 39.

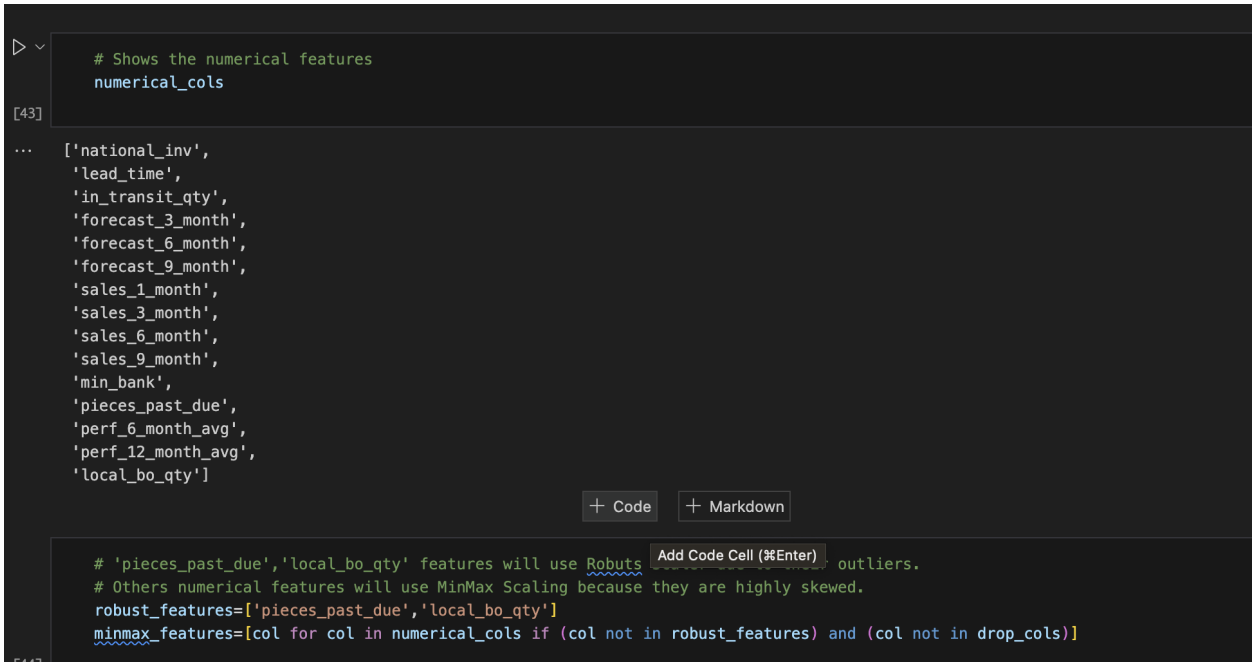


Figure 37: Checks the numerical values and stores them in the MinMax and Robust features variable.

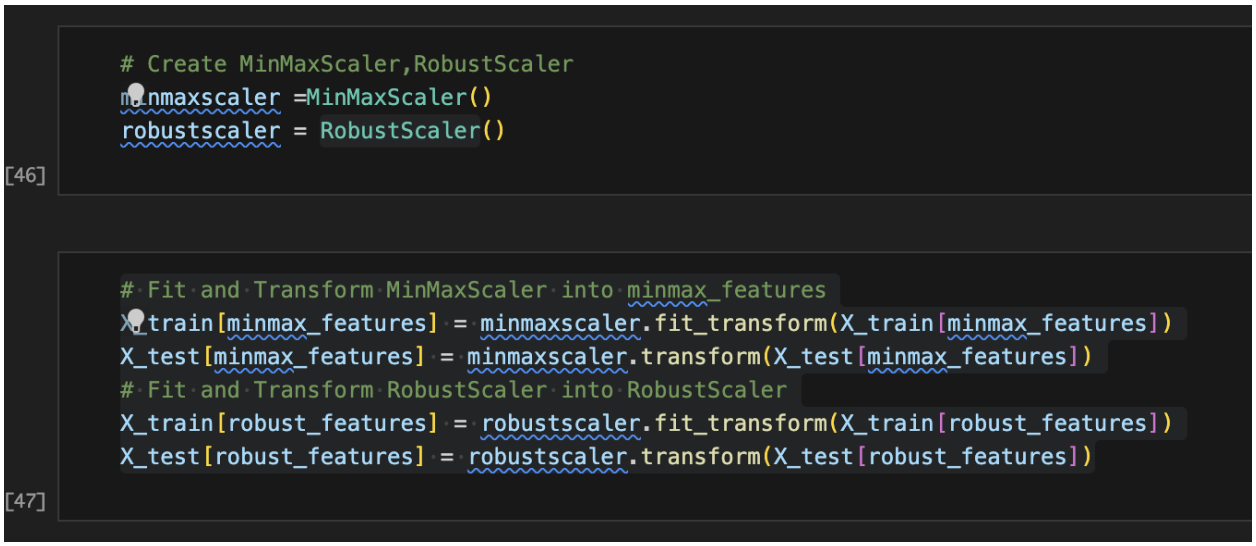


Figure 38: Create, fit, and transform the MinMax and Robust scaling then validate the result.

Figure 39: Create the Pipeline for MinMax and Robust scaling

- Encoding:

Label encoding is employed for the categorical features to perform encoding. The process begins by selecting the features for encoding and creating the Label Encoder. Each categorical feature is iterated through and encoded. The Label Encoder is then validated. A pipeline for encoding is established using an ordinal encoder, which follows the same principles as a label encoder. The code for the encoding process is depicted in Figure 40 to Figure 42.

Figure 40: Perform the label encoding

```
# Check the label encoding result
X_train.head()
```
Python

h	sales_6_month	...	potential_issue	pieces_past_due	perf_6_month_avg	perf_12_month_avg	local_bo_qty	deck_risk	oe_constraint	ppap_risk	stop_auto_buy	rev_stop
3	1.863390e-06	...	0	0.0	0.99	0.99	0.0	0	0	0	1	0
0	0.000000e+00	...	0	0.0	0.80	0.82	0.0	0	0	0	1	0
0	9.316951e-07	...	0	0.0	0.63	0.72	0.0	0	0	0	1	0
0	0.000000e+00	...	0	0.0	0.43	0.19	0.0	1	0	1	1	0
6	1.118034e-05	...	0	0.0	0.94	0.97	0.0	0	0	0	1	0

Figures 41: Validate encoding results

```
# Create ColumnTransformer for encoding
# All category features are using LableEncoder
encoder = ColumnTransformer(
    transformers= [
        ('encoder_label',OrdinalEncoder(),encode_cols)
    ]
    ,remainder='passthrough'
)
```
Python

Figure 42: Create the Pipeline for encoding using the Ordinal Encoder

- Pipeline and Preprocessing Result:

In this section, all the pipelines for the preprocessing steps are combined with the RandomForest Classifier algorithm to make a sample prediction. Furthermore, the results of the preprocessing are validated. The code for this section is illustrated in Figure 43 and Figure 44.

```python
# Create Pipeline for all steps
pipeline = Pipeline([
    ('imputation',imputer),
    ('scaling',scaler),
    ('encoding',encoder),
    # Try with RandomForest Classifier algorithm
    ('prediction',RandomForestClassifier())
])
```

Python

Figure 43: Create the Pipeline for preprocessing and prediction.

Prepocessing Result

```python
X_train.head()
```

Python

	sku	national_inv	lead_time	in_transit_qty	forecast_3_month	forecast_6_month	forecast_9_month	sales_1_month	sales_3_month	sales_6_month	...	po
519658	1867616	0.002057	0.269231	0.00000	6.619921e-07	0.000001	0.000001	0.000001	0.000003	1.863390e-06	...	
1184550	1553554	0.002056	0.153846	0.00000	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000e+00	...	
440461	1788443	0.002059	0.230769	0.00000	0.000000e+00	0.000000	0.000000	0.000000	0.000000	9.316951e-07	...	
227290	1338851	0.002056	0.038462	0.00001	0.000000e+00	0.000000	0.000000	0.000000	0.000000	0.000000e+00	...	
1190017	1559412	0.002059	0.153846	0.00000	0.000000e+00	0.000008	0.000011	0.000000	0.000006	1.118034e-05	...	

5 rows × 22 columns

```python
X_test.head()
```

Python

h	sales_6_month	...	potential_issue	pieces_past_due	perf_6_month_avg	perf_12_month_avg	local_bo_qty	deck_risk	oe_constraint	ppap_risk	stop_auto_buy	rev_stop
0	0.000000e+00	...	0	0.0	1.00	0.82	0.0	0	0	0	1	0
4	6.955104e-04	...	0	0.0	0.87	0.90	0.0	0	0	0	1	0
0	0.000000e+00	...	0	0.0	0.68	0.66	0.0	0	0	1	1	0
7	4.658476e-07	...	0	0.0	0.59	0.61	0.0	0	0	0	1	0
0	4.658476e-07	...	0	0.0	0.79	0.79	0.0	0	0	0	1	0

Figure 44: Validation of X_train and X_test datasets

5. Modeling

- Cross Validation

In this section, we have used different models to compare the algorithm performance across different classification metrics, focusing on the selection of the most appropriate model based on the specific evaluation criteria.

5.1 Cross Validation

```python
# Put the models into a list
algorithms = [LogisticRegression(), GaussianNB(), RandomForestClassifier(),SVC(),GradientBoostingClassifier()]

# The names of the models
algorithm_names = ["LogisticRegression", "GaussianNB", "RandomForestClassifier","SVC","GradientBoosting"]
# Create splits
kfold = KFold(n_splits = 5, shuffle = True, random_state=999)
```

Python

Figure 45: Selection of Models and KFold Cross-Validation Setup

Accuracy score

```python
# Print the 'accuracy' score in each algorithm
for model,modelname in zip(algorithms,algorithm_names):
    score = cross_val_score(model, X_train, y_train, cv=kfold, scoring='accuracy')
    print(modelname,": Scores(accuracy): ", score, "- Scores mean: ", score.mean(), "- Scores std (lower better): ", score.std())
```

Python

```
LogisticRegression : Scores(accuracy):  [0.58023434 0.61248408 0.58547771 0.59694268 0.59363057] - Scores mean:  0.5937538734096713 - Scores std (lower better):  0.011065956881646364
GaussianNB : Scores(accuracy):  [0.52903719 0.52815287 0.53121019 0.52585987 0.53273885] - Scores mean:  0.5293997942834151 - Scores std (lower better):  0.002393500004650214
RandomForestClassifier : Scores(accuracy):  [0.88970963 0.89579618 0.89019108 0.88585987 0.8856051 ] - Scores mean:  0.8898423714839174 - Scores std (lower better):  0.0037030079987225763
SVC : Scores(accuracy):  [0.5397351  0.55770701 0.54598726 0.53910828 0.5411465 ] - Scores mean:  0.544736828784747 - Scores std (lower better):  0.006919211133273972
GradientBoosting : Scores(accuracy):  [0.86118186 0.87414013 0.86980892 0.86675159 0.87006369] - Scores mean:  0.8683892391406628 - Scores std (lower better):  0.0043016307999491675
```

Figure 46 : Calculating accuracy for all the models

F1_score

```python
# Print the 'f1' score in each algorithm
for model,modelname in zip(algorithms,algorithm_names):
    score = cross_val_score(model, X_train, y_train, cv=kfold, scoring='f1')
    print(model," Scores(f1): ", score, "- Scores mean: ", score.mean(), "- Scores std (lower better): ", score.std())
```

Python

```
LogisticRegression()  Scores(f1):  [0.55675094 0.59057873 0.56946282 0.57813333 0.56856911] - Scores mean:  0.5726989880999358 - Scores std (lower better):  0.011234645679585768
GaussianNB()  Scores(f1):  [0.67669173 0.67542937 0.67933078 0.67390923 0.68126521] - Scores mean:  0.6773252640548455 - Scores std (lower better):  0.0026536865819268324
RandomForestClassifier()  Scores(f1):  [0.89333    0.89465649 0.89143426 0.89128257 0.88828203] - Scores mean:  0.8917970689890955 - Scores std (lower better):  0.002157685625666634
SVC()  Scores(f1):  [0.17072051 0.22012579 0.19220308 0.16751035 0.19919964] - Scores mean:  0.18995187625964222 - Scores std (lower better):  0.019362443967904448
GradientBoostingClassifier()  Scores(f1):  [0.86303091 0.87455561 0.87144654 0.86802927 0.87218045] - Scores mean:  0.8698485574076201 - Scores std (lower better):  0.003998442976618223
```

Figure 47 : Calculating F1_score for all the models

Recall score

```python
# Print the 'recall' score in each algorithm
for model,modelname in zip(algorithms,algorithm_names):
    score = cross_val_score(model, X_train, y_train, cv=kfold, scoring='recall')
    print(model," Scores(recall): ", score, "- Scores mean: ", score.mean(), "- Scores std (lower better): ", score.std())
```

Python

```
LogisticRegression()  Scores(recall):  [0.52887072 0.56488157 0.54453441 0.55504352 0.52947103] - Scores mean:  0.5445602508747123 - Scores std (lower better):  0.01411879137086383
GaussianNB()  Scores(recall):  [0.9887583  0.992276   0.98633603 0.98463902 0.98740554] - Scores mean:  0.9878829796985455 - Scores std (lower better):  0.002577174549888707
RandomForestClassifier()  Scores(recall):  [0.91262136 0.90422245 0.9048583  0.90322581 0.89974811] - Scores mean:  0.9049352054365298 - Scores std (lower better):  0.004228921579955629
SVC()  Scores(recall):  [0.09504343 0.1261586  0.10728745 0.09318996 0.11284635] - Scores mean:  0.10690515887336771 - Scores std (lower better):  0.012123909668009929
GradientBoostingClassifier()  Scores(recall):  [0.87736331 0.88671473 0.87651822 0.88069636 0.87657431] - Scores mean:  0.879573385754335 4 - Scores std (lower better):  0.003885171418190 4304
```

Figure 48 : Calculating Recall for all the models

Precision score

```python
# Print the 'precision' score in each algorithm
for model,modelname in zip(algorithms,algorithm_names):
    score = cross_val_score(model, X_train, y_train, cv=kfold, scoring='precision')
    print(model," Scores(precision): ", score, "- Scores mean: ", score.mean(), "- Scores std (lower better): ", score.std())
```

Python

```
LogisticRegression()  Scores(precision):  [0.58773424 0.61872532 0.59678314 0.6032276  0.61390187] - Scores mean:  0.6040744352293085 - Scores std (lower better):  0.011243745382369745
GaussianNB()  Scores(precision):  [0.51435407 0.51195537 0.51807549 0.5122536  0.52003184] - Scores mean:  0.5153340720449368 - Scores std (lower better):  0.003207796293059648 8
RandomForestClassifier()  Scores(precision):  [0.86784141 0.88604418 0.87909936 0.87031558 0.87875801] - Scores mean:  0.8764117081609454 - Scores std (lower better):  0.006574585025805225
SVC()  Scores(precision):  [0.83783784 0.86267606 0.92173913 0.82727273 0.84848485] - Scores mean:  0.8596021200736448 - Scores std (lower better):  0.03320519383612322
GradientBoostingClassifier()  Scores(precision):  [0.84915925 0.86272545 0.86643322 0.85572139 0.86783042] - Scores mean:  0.8603739465508248 - Scores std (lower better):  0.007003700352961142
```

Figure 49: Calculating Pr for all the models

- GridSearch

We have performed a grid search to find the optimal hyperparameters for a RandomForestClassifier using multiple scoring metrics and provides insights into the model's performance on unseen data.

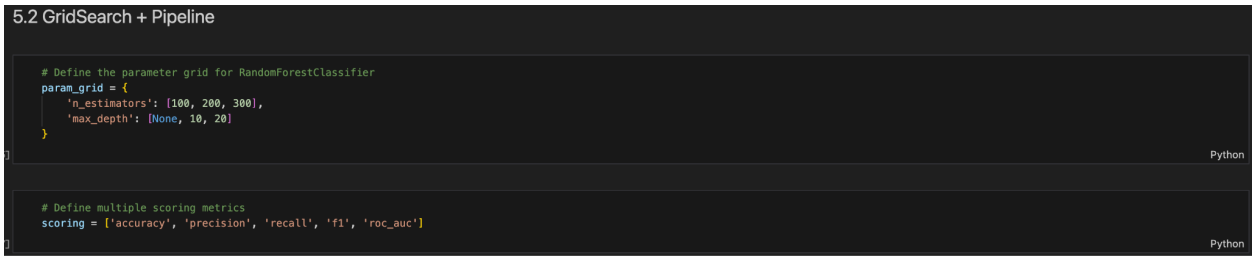


Figure 50 : Defining the parameter grid and scoring metrics

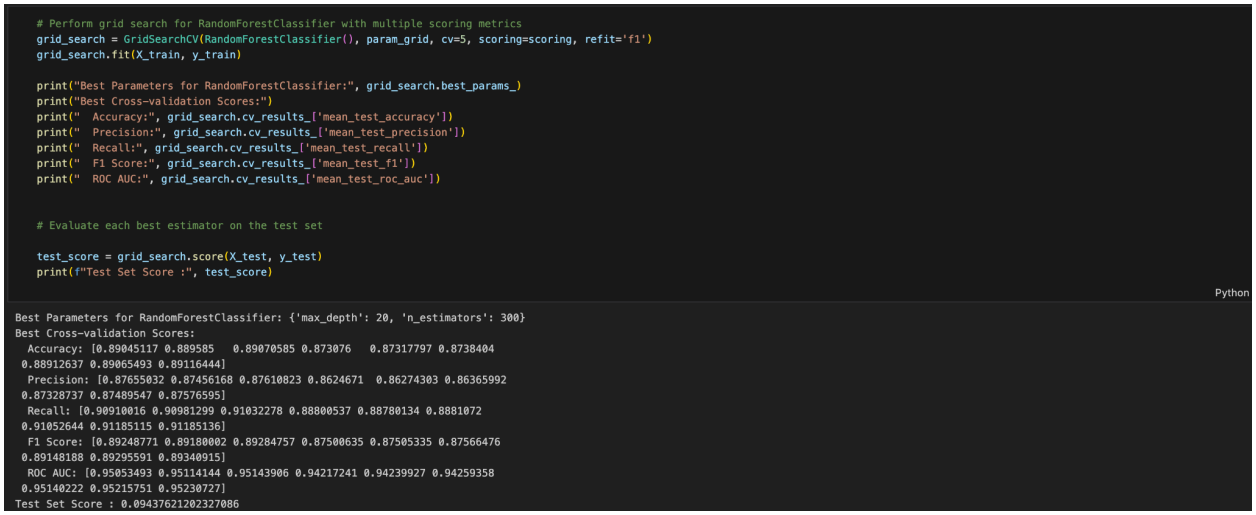


Figure 51: Finding the best parameters and cross-validation scores and evaluate the best estimator on the test set

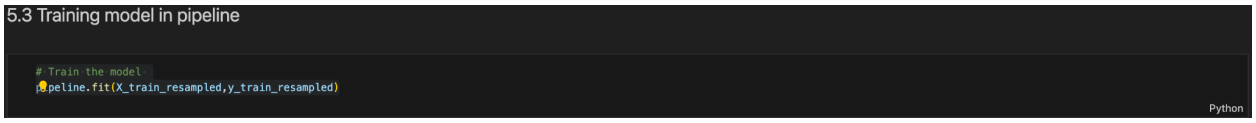
- Training Model in Pipeline

The pipeline is created, integrating both preprocessing and predicting models as RandomForestClassifiers with optimal hyperparameters. The accuracy of the model on the original test set is then assessed, providing a measure of its performance on unseen data. An example of prediction is conducted to test the usability of the pipeline, as illustrated in Figure 52. The RandomForest Classifier algorithm is chosen for testing due to its compatibility with classification problems. The pipelines appear to work well, including prediction performance, as shown in Figure 54. The model's accuracy is approximately 88%, indicating its proficiency in

prediction accuracy. However, to delve deeper into the intricacies of its predictions, the next step involves examining the confusion matrix.

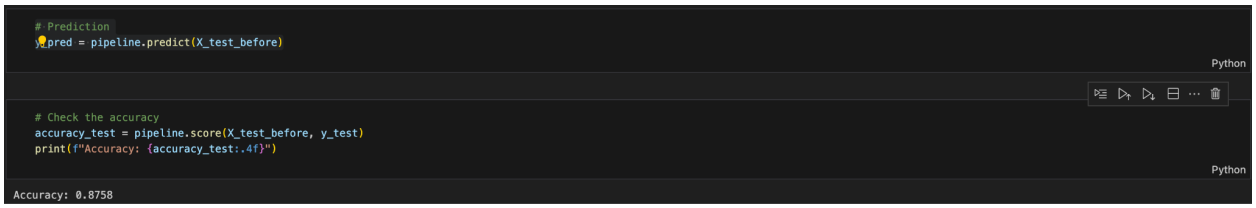

```
# Add the model and best parameter into pipeline
pipeline = Pipeline([
    ('preprocessing',Preprocessor),
    ('prediction',RandomForestClassifier(max_depth=None,n_estimators=300,n_jobs=6))
])
```

Figure 52: Creating Pipeline incorporating preprocessor and model to be used




```
# Train the model
pipeline.fit(X_train_resampled,y_train_resampled)
```

Figure 53: Fitting the training data



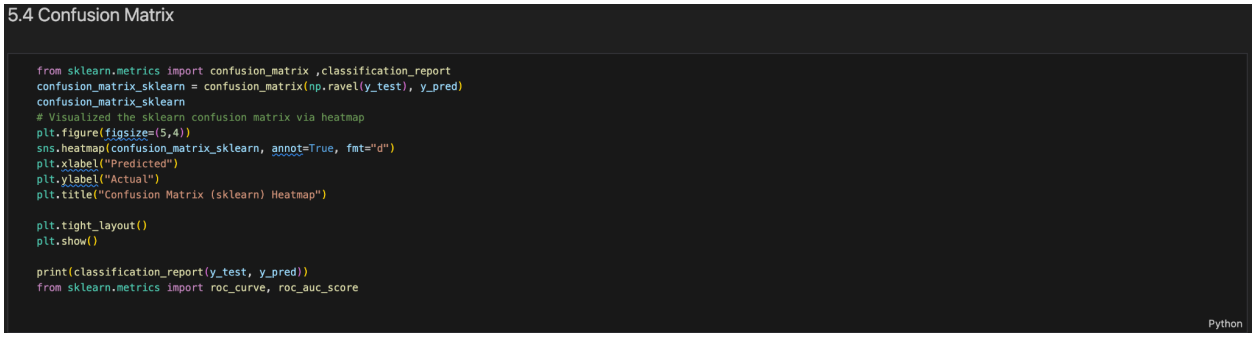

```
# Prediction
y_pred = pipeline.predict(X_test_before)
```

```
# Check the accuracy
accuracy_test = pipeline.score(X_test_before, y_test)
print(f"Accuracy: {accuracy_test:.4f}")
```

Figure 54: Predicting the test set from trained model and calculated the accuracy

- Confusion Matrix

In this section, we have calculated and visualized a confusion matrix using scikit-learn's confusion matrix function, and provides insights to differentiate between classes and the trade-off between precision and recall. The precision, recall, and F1-score are computed for both the training and test sets providing insights on model's performance.




```
from sklearn.metrics import confusion_matrix ,classification_report
confusion_matrix_sklearn = confusion_matrix(np.ravel(y_test), y_pred)
confusion_matrix_sklearn
# Visualized the sklearn confusion matrix via heatmap
plt.figure(figsize=(5,4))
sns.heatmap(confusion_matrix_sklearn, annot=True, fmt="d")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix (sklearn) Heatmap")

plt.tight_layout()
plt.show()

print(classification_report(y_test, y_pred))
from sklearn.metrics import roc_curve, roc_auc_score
```

Figure 55: Using Confusion Matrix function to generate classification report

```python
# Predictions on training and test sets
y_train_pred = pipeline.predict(X_train_resampled)
y_test_pred = pipeline.predict(X_test_before)

# Calculate precision, recall, and F1-score on both sets
precision_train = precision_score(y_train_resampled, y_train_pred)
recall_train = recall_score(y_train_resampled, y_train_pred)
f1_train = f1_score(y_train_resampled, y_train_pred)

precision_test = precision_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred)
f1_test = f1_score(y_test, y_test_pred)

print("Training Set:")
print(f"Precision: {precision_train:.4f}, Recall: {recall_train:.4f}, F1-score: {f1_train:.4f}")
print("Test Set:")
print(f"Precision: {precision_test:.4f}, Recall: {recall_test:.4f}, F1-score: {f1_test:.4f}")
```

Python

```
Training Set:
Precision: 0.9928, Recall: 0.9918, F1-score: 0.9923
Test Set:
Precision: 0.0503, Recall: 0.9086, F1-score: 0.0953
```

Figure 56: Model Predictions on training and test set and calculating evaluation metrics

- ROC Curve

The Receiver Operating Characteristic (ROC) curve is generated for both the training and test sets and calculates the corresponding Area Under the Curve (AUC) values. ROC curve helps in finding the model's ability to discriminate between the positive and negative classes across different probability thresholds.

5.5 ROC curve

```python
# Call the model from pipeline
pipeline['prediction']
```

Python

```
RandomForestClassifier(n_estimators=300)
```

```python
# Get predicted class probabilities for the test set
y_pred_pro_test = pipeline.predict_proba(X_test_before)[:, 1]
y_pred_prob_train = pipeline.predict_proba(X_train_resampled)[:, 1]
```

Python

Figure 57: Predicting Class Probabilities

```python
# Compute ROC curve and its corresponding values for both training and test sets
fpr_train, tpr_train, thresholds_train = roc_curve(y_train_resampled, y_pred_prob_train)
fpr_test, tpr_test, thresholds_test = roc_curve(y_test, y_pred_pro_test)
# Calculate the Area Under the ROC Curve (AUC) for both training and test sets
auc_train = roc_auc_score(y_train_resampled, y_pred_prob_train)
auc_test = roc_auc_score(y_test, y_pred_pro_test)
# Plot ROC curves for both training and test sets on the same plot
plt.figure(figsize=(8, 6))
plt.plot(fpr_train, tpr_train, label=f'Training ROC Curve (AUC = {auc_train:.2f})', color='blue')
plt.plot(fpr_test, tpr_test, label=f'Test ROC Curve (AUC = {auc_test:.2f})', color='orange')
plt.plot([0, 1], [0, 1], 'k--')  # Plotting the diagonal line for reference
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```

Python

Figure 58: Computing ROC Curve and Threshold

6. Feature Importance

We have used feature importance illustrated in the below figure to show the ranking of the features based on the importance, with the most important feature listed first. We have used the

best parameters and the best estimator from a grid search and then prints the feature importances of the selected model, ranked them in descending order.

Figure 59: Extracting the best parameters and best estimators

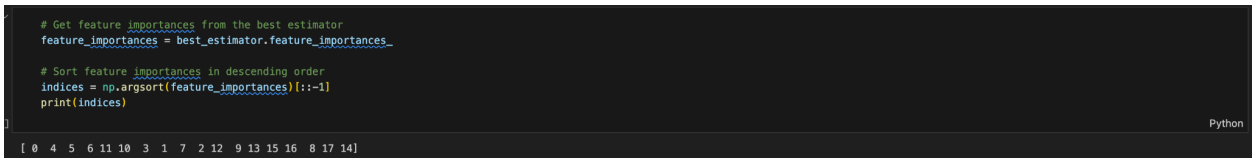
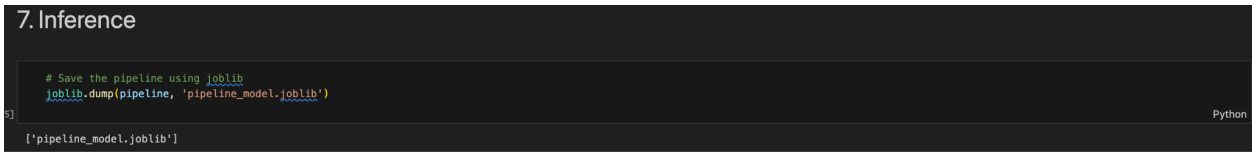


Figure 60: Sorting the features based on descending order

7. Inference

The following figure illustrates the process of saving a machine learning pipeline using joblib, loading the saved pipeline, and making predictions, both on the entire test set and a single sample.



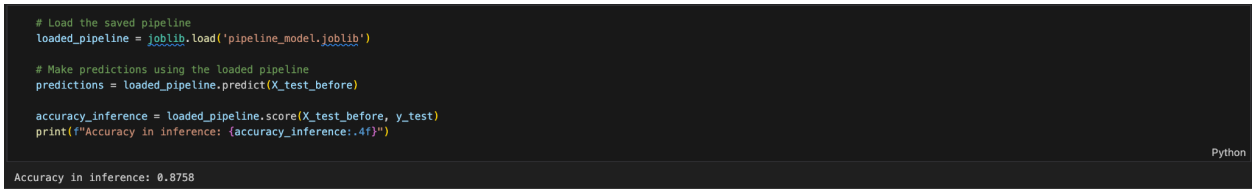


Figure 61: Inference on the Entire Test Set

```python
# Extract a single sample and convert it to a DataFrame
single_sample = X_test_before.iloc[100]  # Assuming this is a single row of your test data
single_sample_df = pd.DataFrame([single_sample], columns=X_test_before.columns)
```

Python

```python
single_sample
```

Python

```
national_inv         262.0
lead_time             52.0
in_transit_qty         0.0
sales_1_month          3.0
sales_3_month          9.0
sales_6_month         40.0
sales_9_month         59.0
min_bank               2.0
potential_issue         No
pieces_past_due        0.0
perf_6_month_avg       0.0
perf_12_month_avg      0.0
local_bo_qty           0.0
deck_risk              Yes
oe_constraint           No
ppap_risk               No
stop_auto_buy          Yes
rev_stop                No
Name: 1766297, dtype: object
```

Figure 62 : Inference on a Single Sample

```python
# # Make predictions using the loaded pipeline on the single sample
pred_test = loaded_pipeline.predict(single_sample_df)

# Mapping the predicted values (0 or 1)
pred_test_mapped = ['No' if pred == 0 else 'Yes' for pred in pred_test]
print(f'This product can potentially be a backorder?: {pred_test_mapped[0]}')
```

Python

```
This product can potentially be a backorder?: No
```

Figure 63: Printing the result whether the sample is in backorder state or not

7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

In the exploration and application of various machine learning models for backorder prediction, including Logistic Regression, Gaussian Naive Bayes, Random Forest, SVM, and Gradient Boosting, it was found that the Random Forest model exhibited the highest accuracy on the undersampled dataset. The choice of Random Forest suggests its effectiveness in capturing the complexities of the data and making accurate predictions. However, it was observed that the model's strength lies in accurately predicting the target class, while its performance on the other class could be further improved. This implies that the model has a tendency to focus more on detecting instances of backorders (the target class) and might neglect instances that do not fall into this category. In real-world inventory management scenarios, where the consequences of backorders can be significant, this model could prove valuable in identifying potential issues.

The successful creation and deployment of the model's pipeline can facilitate real-time predictions and integration into inventory management systems, contributing to more informed decision-making. To enhance the model's applicability, it is recommended to further optimize it to achieve a balance between precision and recall. This means finding a suitable threshold or adjusting model parameters to reduce the trade-off between correctly identifying backorders (precision) and capturing as many backorders as possible (recall). Achieving a better balance in performance metrics would make the model more robust and reliable in handling a variety of inventory management scenarios. Overall, the Random Forest model shows promise for backorder prediction, and with continued refinement, it can become a valuable tool in optimizing inventory and supply chain management processes.

7.2 Future Work

The identified future work for backorder prediction involves addressing class imbalance, performing tree-based feature selection, implementing deep learning modules, and exploring feature engineering techniques like Principal Component Analysis (PCA). Here's an explanation of each aspect:

1. **Addressing Class Imbalance with Oversampling**

The recommendation to try oversampling techniques implies addressing the class imbalance in the dataset. This involves increasing the number of instances in the minority class (backorders) to achieve a more balanced distribution. Techniques such as Synthetic Minority Over-sampling Technique (SMOTE) could be applied to generate synthetic instances and enhance the representation of the minority class.

2. **Tree-Based Feature Selection**

Performing tree-based feature selection techniques, such as those based on Random Forests or Gradient Boosting, can help identify the most relevant features for the prediction task. By focusing on the most informative features, the model can potentially improve its performance and reduce overfitting.

3. **Implementing Deep Learning Modules**

Exploring deep learning modules offers the potential for capturing intricate patterns and dependencies in the data. Neural networks, especially deep learning architectures like deep neural networks or recurrent neural networks, can uncover complex relationships that may not be apparent with traditional machine learning models. This approach could enhance the model's ability to generalize and make accurate predictions.

4. **Feature Engineering with PCA**

Feature engineering, specifically employing Principal Component Analysis (PCA), is suggested as a method to achieve a balance between precision and recall. PCA transforms the original features into a set of linearly uncorrelated components, potentially reducing dimensionality and highlighting the most important information. This can be beneficial for improving model performance while maintaining a balance between the positive and negative classes.

In summary, these future work recommendations aim to enhance the backorder prediction model in different aspects. Addressing class imbalance, selecting relevant features, exploring deep learning capabilities, and leveraging feature engineering techniques can collectively contribute to a more robust and accurate predictive model. Implementing these strategies would likely lead to improved performance and reliability in real-world scenarios of inventory and supply chain management.

7. REFERENCES

[1] Santis, Rodrigo & Pestana de Aguiar, Eduardo & Goliatt, Leonardo. (2017). Predicting Material Backorders in Inventory Management using Machine Learning. https://doi.org/10.1109/LA-CCI.2017.8285684

[2] Islam, S., Amin, S.H. Prediction of probable backorder scenarios in the supply chain using Distributed Random Forest and Gradient Boosting Machine learning techniques. J Big Data 7, 65 (2020). https://doi.org/10.1186/s40537-020-00345-2

[3] Gao, H., Ren, Q., & Lv, C. (2022). Supply Chain Management and Backorder Products Prediction Utilizing Neural Network and Naive Bayes Machine Learning Techniques in Big Data Area: A Real-life Case Study. https://doi.org/10.21203/rs.3.rs-2020401/v1

[4] S. Zinjad, "Backorder Dataset," Kaggle, [Online]. Available: https://www.kaggle.com/datasets/ztrimus/backorder-dataset?select=Kaggle_Training_Dataset_v2.csv.