# Data-Driven Insights into Hotel Booking Trends and Cancellation Prediction

**Amrita Moyade**

A comprehensive analysis of hotel booking patterns and cancellations, leveraging data-driven approaches to predict trends and optimize operations.

**28 November 2024**

## Contents

## List of Figures

# List of Tables

## 1 Abstract

The hotel industry is a low-margin industry; hence, cost-saving measures and efficiency enhancements are critical to maintaining profitability. Booking cancellations are a persistent challenge in the hospitality industry, impacting operational efficiency and revenue forecasting. For hotels, predicting cancellations is critical for resource optimization and for enhancing guest satisfaction through better planning.

This report investigates historical hotel booking data to identify key patterns and develop a predictive framework for cancellations by leveraging the Random Forest model, to address the challenges like class imbalance, noisy data and overfitting. The findings provide practical insights that can guide data-driven strategies for improving hotel operations.

## 2 Dataset

This study leverages a dataset of 36,275 hotel bookings sourced from Kaggle called Hotel Reservations Dataset.

Dataset link - https://www.kaggle.com/datasets/ahsan81/hotel-reservations-classification-dataset

The data includes bookings for a European hotel, featuring 18 features:

| **Feature Name** | **Type** | **Description** |
|---|---|---|
| Booking_ID | Categorical | Unique identifier for each booking |
| Number_of_Adults | Ordinal | Number of adults |
| Number_of_Children | Ordinal | Number of children |
| Number_of_Weekend Nights | Ordinal | Number of weekend nights (Saturday or Sunday) the guest stayed or booked to stay at the hotel |
| Number_of_Week Nights | Ordinal | Number of week nights (Monday to Friday) the guest stayed or booked to stay at the hotel |
| Type of Meal Plan | Categorical | Type of meal plan booked by the customer |
| Required_Car_ Parking_Space | Binary | Does the customer require a car parking space? (0 - No, 1 - Yes) |
| Room_Type Reserved | Categorical | Type of room reserved by the customer. The values are encoded at source; hence no description of encoding is available. |
| Lead_ Time | Continuous | Number of days between the date of booking and the arrival date |
| Arrival_Year | Temporal | Year of arrival date |

| | | |
|---|---|---|
| Arrival_Month | Temporal | Month of arrival date |
| Arrival_Date | Temporal | Date of the month |
| Market_Segment_Type | Categorical | Market segment designation |
| Repeated_Guest | Binary | Is the customer a repeated guest? (0 - No, 1 - Yes) |
| Number_of_Previous_Cancellations | Ordinal | Number of previous bookings that were canceled by the customer prior to the current booking |
| Number_of_Previous_Bookings_Not_Canceled | Ordinal | Number of previous bookings not canceled by the customer prior to the current booking |
| Average_Price_Per_Room | Continuous | Average price per day of the reservation; prices of the rooms are dynamic (in euros) |
| Number_of_Special_Requests | Ordinal | Total number of special requests made by the customer (e.g., high floor, view from the room, etc.) |
| Booking_Status | Categorical | Flag indicating if the booking was canceled or not |

Table 1: Feature Names, Types, and Descriptions

## 3 Problem Statement

Use Random Forest with cost-sensitive learning to predict `booking_status`, which indicates whether a booking is `Canceled` or `Not_Canceled`.

## 4 Objectives

- Analyze hotel booking data to identify patterns and trends influencing booking cancellations.
- Develop a Random Forest Classifier to predict cancellations accurately.
- Address key challenges - Mitigate class imbalance, overfitting, dimensionality and ensure model interpretability.
- Evaluate model performance by conducting cross-validation, bias-variance analysis, and assess metrics like precision, recall, F1 score, accuracy and AUC.
- Derive practical insights and provide actionable recommendations for improving operational efficiency and resource planning in hotels.

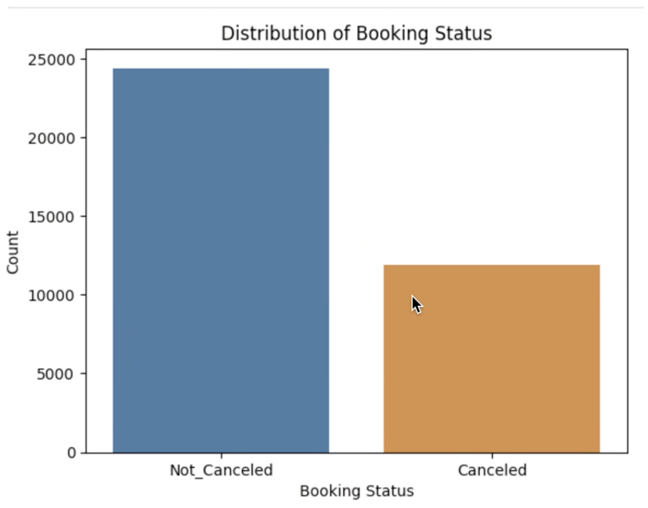

Figure 1: Class Imbalance Analysis

## 5 Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a critical step in understanding the dataset's structure, identifying patterns, and detecting anomalies or outliers. In this project, we performed the EDA to gain insights into key booking trends and assess the relationships between features that might influence booking cancellations.

We started with checking the types of features and there are no null values in the dataset. There are 1 float, 13 integer, and 5 object type features.

Then, we performed a class imbalance analysis to verify if in the target variable, one class has more examples than the other. It was found that `Not_Canceled` has more than double the instances of `Canceled` class.

Total instances of class `Not_Canceled` = 24390
Total instances of class `Canceled` = 11885

### 5.1 Feature-Target Correlation Analysis

A Feature-Target Correlation Analysis was performed to find how different features affect the target.

- **Room Price**:

  Room price doesn't seem to affect booking status since the average room price for both bookings is similar.

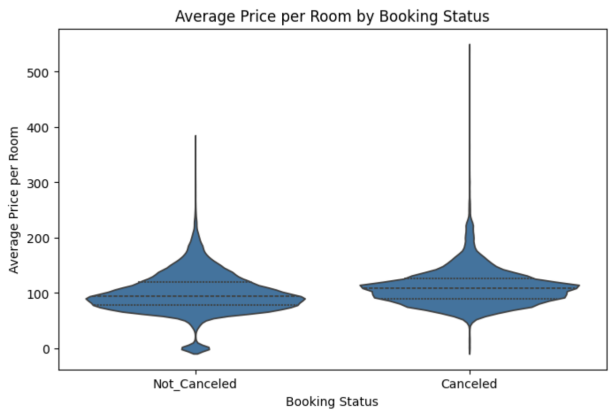

Figure 2: Room Price vs Booking Status

- **Special Requests**:

  Special request seems to impact cancellations since the bookings with special requests had lesser cancellations. Furthermore, with bookings where the number of special requests was more than 3, there were no cancellations. Encouraging customers to specify their preferences or requirements (e.g., accessible room, room with a view) can reduce cancellations.

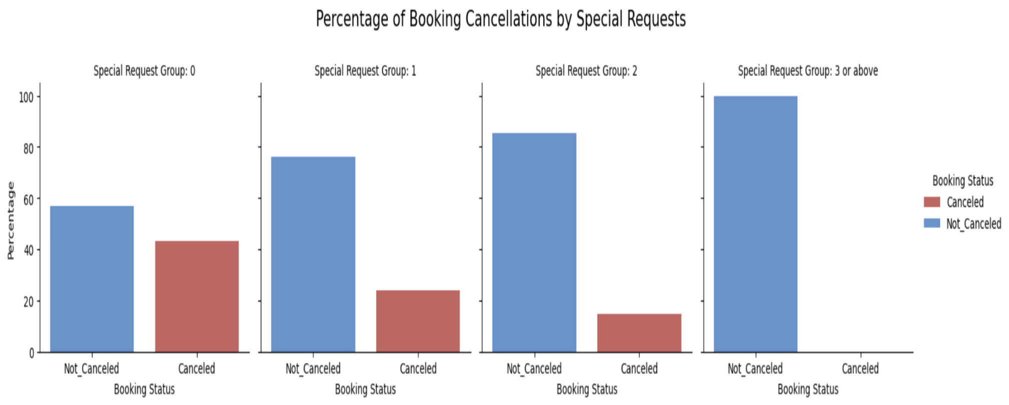

Figure 3: Special Request vs Booking Status

- **Month**:

  From this graph, cancellations seem to be at their lowest in January, which gradually increases and peaks in July, and then decline towards December. However, there is an exception in October, where there is a slight rise in cancellations compared to the surrounding months. This suggests a potential seasonal trend in cancellations. Reducing staff and resource allocation during June, July and August, while optimizing resources during January and December, can improve financial efficiency and enhance customer satisfaction.

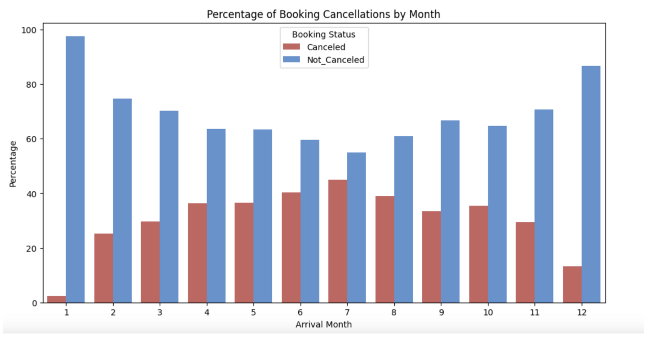

Figure 4: Month vs Booking Status

- **Day of the Week**

  Weekday or weekend booking doesn't seem to affect the booking cancellations as they had similar trend for both type of bookings.

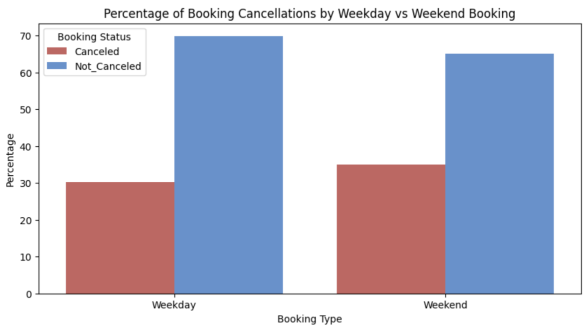

Figure 5: Day of the Week vs Booking Status

- **Presence of Children**:

  Presence of children also didn't have a noticeable impact on booking cancellations.

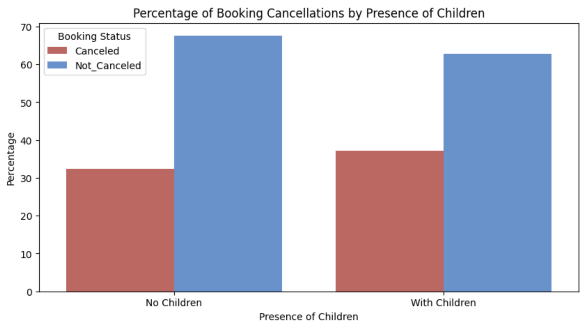

Figure 6: Presence of Children vs Booking Status

- **Market Segment**:

  There seem to be no cancellations in Complimentary bookings while Corporate segment exhibits negligible cancellation rates. In contrast, the Aviation, Offline and Online market segments show relatively higher cancellation rates. Strengthening relationships with Corporate clients and offering incentives to Complementary bookings can help maintain or reduce cancellations in these segments, while offering flexible booking options for Aviation clients can help.

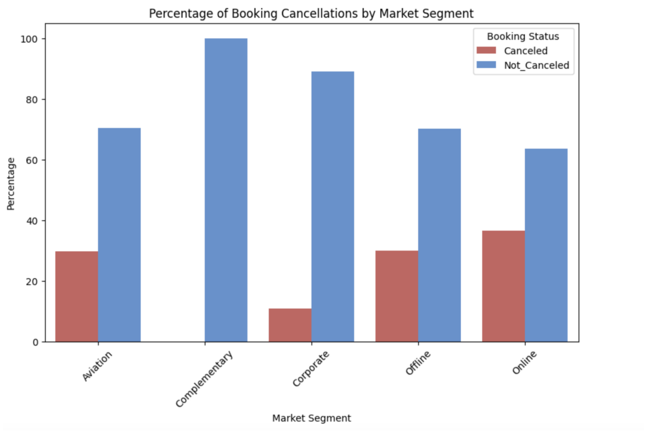

Figure 7: Market Segment vs Booking Status

- **Returning Guest**:

  Returning guests are highly unlikely to cancel the bookings compared to first time guests. Loyalty programs can be offered for returning guests to turn first time guests to returning guests as well as increasing bookings from returning guests. Providing exceptional service and personalized follow up post stay asking for feedback can also leave a positive impression leading to guest choosing to book again in future.

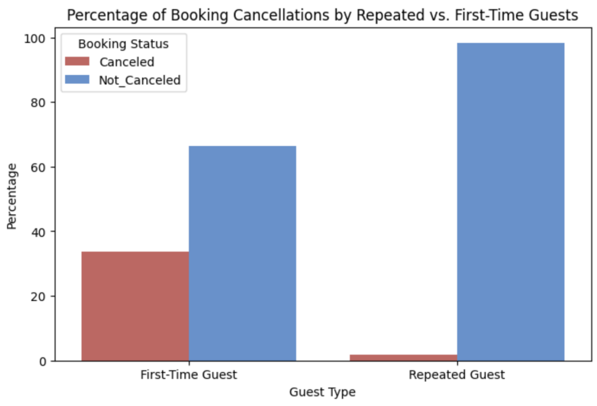

Figure 8: Returning or First Time Guest vs Booking Status

## 6 Modeling Methodology and Techniques

For this project, we are using the Random Forest Classifier, which is an Ensemble method in Supervised Learning. Ensemble methods are those where multiple machine learning models are combined to construct one final model.

Decision trees, which form the foundation of Random Forest, work by recursively splitting the dataset based on feature thresholds to create branches that lead to specific outcomes called leaves. Each split aims to maximize information gain (Gini Index) or minimize impurity (Entropy). While decision trees are simple to interpret and powerful for small datasets, they are prone to overfitting, especially with noisy or complex data, as they can memorize the training data rather than generalize.

Random Forest mitigates these issues by constructing multiple decision trees, each trained on a randomly sampled subset of data with a random selection of features. This process is known as bootstrap aggregation or bagging. This ensemble approach enhances predictive accuracy and reduces the risk of overfitting, as the aggregated predictions from multiple trees mitigate the biases and variances of individual trees.

Additionally, Random Forest is particularly effective in handling imbalanced datasets, such as ours, by integrating techniques like Cost Sensitive Learning to address disparities in class representation. Furthermore, it has the ability to rank features based on their importance, which helps reduce noisy data, minimizes overfitting, and enhances the model's accuracy and interpretability. [1]

The decision to employ a Random Forest classifier for this project was driven by insights gained during exploratory data analysis. The dataset exhibited a significant class imbalance and the presence of potentially noisy features. Random Forest, with its ensemble-based approach, was deemed suitable for this task due to its ability to handle imbalanced data, mitigate overfitting through bagging, and rank features based on importance.

## 6.1 Overview of Random Forest:

### 6.1.1 Underlying Principles:

Random forest follows these two underlying principles:

1. **Bootstrap Aggregating (Bagging)**: Bagging means training each tree on a random subset of the training data, ensuring that individual trees are trained on diverse samples. It reduces variance and improves accuracy.

2. **Random Feature Selection**: When building each split within a tree, only a random subset of features is considered. This reduces correlation between trees, making the forest more diverse and generalized.

### 6.1.2 Assumptions:

Random Forest has the following assumptions:

- **Independence of Observations**: Random Forest assumes that the observations in the dataset are independent of each other.

- **Minimal Preprocessing**: It assumes that minimal pre-processing is performed as it's robust to feature scaling and generally doesn't require extensive data cleaning.

- **Meaningful Features**: It assumes that the features have meaningful information.

### 6.1.3 Strengths:

Random Forest has the following strengths:

- **Handling Non-linearity and Complex Patterns**: Each tree's ability to capture different aspects of the data enables the forest to handle intricate relationships.
- **Feature Importance Measurement**: It provides feature importance scores, which can help interpret the model.
- **Low Sensitivity to Overfitting**: Since it averages multiple trees, it's less likely to overfit compared to individual decision trees.

### 6.1.4 Hyperparameters

The hyperparameters are settings used to control the model complexity, accuracy, generalization and balancing the trade offs between variance and bias. These hyperparemeters needs to be tuned according to the requirements. These are the hyperparameters of Random Forest:

- **Number of Trees** (`n_estimators`): This is the number of trees in forest. Increasing it can improve accuracy and robustness but up to a point, but it increases computational costs.
- **Max Depth** (`max_depth`): This is the depth of each tree and is used to reduce overfitting. Deeper trees improve accuracy by capturing more complexity but can cause overfitting and high computational costs.
- **Min Samples per Split** (`min_samples_split`): This is the minimum number of samples needed to split a node. Increasing it reduces model complexity and overfitting, while decreasing it allows more flexibility.
- **Number of Features** (`max_features`): This is the max number of features considered per split. Increasing it improves accuracy but may cause overfitting, while decreasing it reduces risk of overfitting at the cost of accuracy.

### 6.1.5 Training Algorithm:

Random Forest is trained as follows: [2]

1. **Bootstrapping**: A random subset of data points is sampled with replacement to create a training set for each tree.

   Let the original dataset be $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ , where $x_i$ are the features and $y_i$ are the corresponding labels.

   For each tree $T_k$ , sample m data points with replacement from D, forming the bootstrapped dataset $D_k$

   $$D_k = \{(x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), \ldots, (x_{i_m}, y_{i_m})\}, \quad i_j \in \{1, 2, \ldots, n\}$$

   Here, $m \leq n$ , and some data points in D may appear multiple times in $D_k$ , while others may not appear at all.

2. **Tree Construction**: For each tree $T_k$, the decision tree algorithm is applied on the bootstrapped dataset $D_k$. At each node, only a random subset of features is considered for splitting which reduces the correlation among the trees.

   (a) **Splitting Criterion**

   At each split in the tree:

   i. Randomly select a subset of p features $F_k \subseteq F$ , where F is the full set of features and $|F_k| \ll |F|$ .
   ii. For each feature $f \in F_k$ , evaluate potential splits s to minimize a splitting criterion $G(f, s)$

   (b) **Recursive Splitting**

   The dataset is recursively partitioned based on the best split until a stopping criterion is met, such as:

   i. Maximum depth of the tree is reached (depth $\leq d_{\max}$)
   ii. A minimum number of samples at a node (samples $\geq n_{\min}$)

3. **Ensemble Aggregation**: Once all trees $T_1, T_2, \ldots, T_k$ are trained, the forest combines the predictions of all trees to determine the final class by majority vote.

   (a) **Individual Tree Predictions**

   Each tree $T_k$ makes a prediction $\hat{y}_k(x)$ for an input x .

   (b) **Majority Voting**

   For classification problems, the final prediction $\hat{y}(x)$ is determined by majority vote:

   $\hat{y}(x) = \text{argmax} c \in C \sum k = 1^K ⊮[\hat{y}_k(x) = c]$,
   where:

   - C is the set of possible classes,
   - $⊮[\cdot]$ is the indicator function that equals 1 if the condition is true and 0 otherwise.

#### 6.1.6 Prediction on Test Data

These are the steps followed for predicting class on test data:

1. **Individual Tree Predictions:**

   Each tree produces a class prediction based on the test data. Each decision tree $T_i$, when presented with a test instance $\mathbf{x}$ produces a class prediction $h_i(\mathbf{x})$ , where:

   $h_i(\mathbf{x}) \in \{C_1, C_2, \ldots, C_k\}$

Here:

$h_i(\mathbf{x})$ is the prediction made by the i -th tree

$C_1, C_2, \ldots, C_k$ are the k possible class labels

2. **Aggregating Predictions:**

   The predictions from individual trees are aggregated using a majority voting mechanism to decide final prediction for classification.

   $$\hat{y} = \underset{C_j \in \{C_1, C_2, \ldots, C_k\}}{\operatorname{argmax}} \sum_{i=1}^{n} \mathbb{I}\left(h_i(\mathbf{x}) = C_j\right)$$

   Where:

   $\mathbb{I}(\cdot)$ is the indicator function that returns 1 if $h_i(\mathbf{x}) = C_j$ , otherwise 0.

   $\sum_{i=1}^{n} \mathbb{I}(h_i(\mathbf{x}) = C_j)$ counts the number of trees predicting class $C_j$.

3. **Probability Estimates (Optional)**

   This is optional and is done when there is a tie and a tiebreaker is required to predict one class as final prediction. If probability estimates are needed, it is computed by taking the probability of each class $C_j$ by averaging the predictions of all trees:

   $$P(C_j \mid \mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}\left(h_i(\mathbf{x}) = C_j\right)$$

   The predicted class $\hat{y}$ is then chosen as the one with the highest probability:

   $$\hat{y} = \underset{C_j \in \{C_1, C_2, \ldots, C_k\}}{\operatorname{argmax}} P(C_j \mid \mathbf{x})$$

**Example for Binary Classification:**

For a binary classification problem with class $C_1$ represented by 0 and class $C_2$ represented by 1:

1. Suppose there are n = 5 trees in the forest.
2. The individual tree predictions for a test instance $\mathbf{x}$ are:
   $h_1(\mathbf{x}) = 1$, $h_2(\mathbf{x}) = 0$, $h_3(\mathbf{x}) = 1$, $h_4(\mathbf{x}) = 1$, $h_5(\mathbf{x}) = 0$
3. Aggregation:

   Votes for $C_1 = 0 : \mathbb{I}(h_i(\mathbf{x}) = 0) = 2$

   Votes for $C_2 = 1 : \mathbb{I}(h_i(\mathbf{x}) = 1) = 3$

4. Final Prediction:
   $\hat{y} = 1 \quad \text{(class 2 has more votes)}$
5. Probability Estimates:
   $P(C_1 \mid \mathbf{x}) = \frac{2}{5} = 0.4, \quad P(C_2 \mid \mathbf{x}) = \frac{3}{5} = 0.6$

Predicted class will be the class with $\hat{y} = 1$ i.e., class 2 (since $P(C_2 \mid \mathbf{x}) > P(C_1 \mid \mathbf{x})$)

### 6.1.7 Appropriate Problem Types

Random Forest is very versatile and is suitable for this category of problems:

- **Classification**: It is very effective in scenarios with imbalanced datasets, such as fraud detection.
- **Regression**: It works well in cases where there's non-linearity and complex relationships, such as housing prices prediction.
- **Mixed data types**: It works well with both categorical and continuous features.
- **Missing and Noisy Data**: It can handle missing and noisy data.
- **High-Dimensional Data**: It performs well in datasets with many irrelevant features.
- **Explainability**: It provides a feature importance score which helps with interpretation of model, where it's important such as medical datasets.

Some of the real-world problems where Random Forest excels are - Fraud detection, credit risk assessment, customer churn prediction, medical diagnosis, image classification, recommendation systems, sales forecasting, spam detection, loan default prediction, disease outbreak prediction, and sentiment analysis.

## 6.2 Evaluation Metrics

The efficacy of a machine learning model hinges on its ability to accurately make predictions. To comprehensively assess the performance of the model trained in this study, a suite of evaluation metrics has been employed.

To understand the evaluation metrics used in this research, it is important to first define the confusion matrix, a table that evaluates a classification model by comparing predicted and actual outcomes. It consists of four components: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). These elements form the foundation for calculating key metrics—Precision, Recall, F1-score and Accuracy—which assess the model's ability to correctly classify instances and identify errors.

- **True Positives (TP)**: Instances where the model correctly predicts the positive class i.e., identify cancellations as cancellations.
- **True Negatives (TN)**: Instances where the model correctly predicts the negative class, i.e., identifying non-cancellations as non-cancellations.
- **False Positives (FP)**: Instances where the model incorrectly predicts the positive class i.e., mistakenly classifying a non-cancellation as a cancellation.

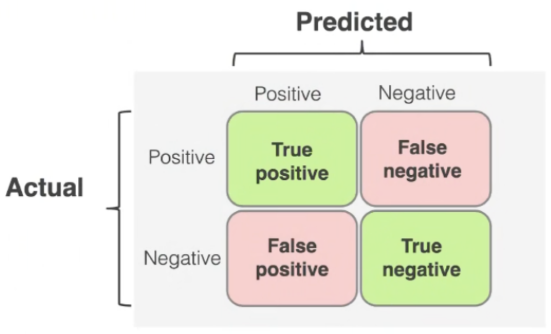

Figure 9: Confusion Matrix

- **False Negatives (FN)**: Instances where the model incorrectly predicts the negative class i.e., identify cancellation as a non-cancellation.

Now based on these components, the crucial evaluation metrics can be defined as:

- **Precision:** The proportion of correctly predicted positive instances out of all instances predicted as positive.

  $\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$

- **Recall:** The proportion of correctly predicted positive instances out of all actual positive instances.

  $\text{Recall} = \frac{\text{TP}}{\text{TP+FN}}$

- **F1-Score:** The harmonic mean of Precision and Recall, providing a balance between the two metrics.

  $\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision+Recall}}$

- **Accuracy:** The proportion of correctly predicted instances (both positive and negative) out of all instances.

  $\text{Accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$

The focus is on assessing recall and precision, as these metrics play a crucial role in optimizing revenue and maintaining customer satisfaction. High recall ensures that actual cancellations are correctly predicted, allowing the hotel to reallocate rooms to other potential customers and reduce revenue loss. Additionally, correct predictions enable effective cost management by deallocating resources on days with high predicted cancellations, ensuring financial efficiency.

Meanwhile, high precision prevents overestimating cancellations, ensuring smooth operations with adequate staffing and minimal disruptions, ultimately enhancing customer satisfaction.

Furthermore, to evaluate the overall performance and robustness of the model, we also considered metrics such as the F1-score to balance precision and recall, ensuring neither is disproportionately prioritized.

However, we did not calculate Accuracy because it can be misleading in the presence of class imbalance, as it may overestimate model performance by focusing on the majority class while ignoring the minority class, which is critical in our context.

## 6.3 Implementation

### 6.3.1 Feature Encoding

Machine Learning models can only work with numerical variables. To prepare the data for modeling, categorical features were transformed into numerical features:

- **Label Encoding**: The target variable `booking_status`, was converted into boolean labels.
  - `Not_Canceled` was encoded as 0.
  - `Canceled` was encoded as 1.
- **One-Hot Encoding**: Categorical features were converted to binary columns using one-hot encoding. This ensured that the model could handle these features.

### 6.3.2 Dataset Splitting

To prepare the dataset for model training and evaluation, the data is divided into three distinct subsets: training, validation, and test sets. This step helps prevent overfitting, tune hyperparameters and evaluate model effectively.

There is no fixed rule to the size of data portions, but some splits which are commonly used are 60:20:20, 70:20:10 and 80:10:10. The first value is the size of training data, second value is size of validation data and third is the size of training data.

```
Training:Validation:Testing
```

To prepare the dataset for model training and evaluation, the data was divided into three distinct subsets:

- **Training Set**: It is used to train the model and is the largest portion of the dataset to ensure the model has enough examples to learn effectively.
- **Validation Set**: It is used during hyperparameter tuning to assess the model's performance on unseen data and mitigate the risk of overfitting. It is generally a very small portion relative to the training set.
- **Test Set**: This is the data held out until the final evaluation phase to assess the model's generalization performance on completely unseen data. It is generally similar sized as the validation set.

### Stratified Splitting

Since the target variable (`booking_status`) exhibited class imbalance, stratified sampling was employed during splitting. This ensured that the proportion of `Canceled` and `Not_Canceled` classes was preserved across the training, validation, and test sets. This approach prevents bias during training and provides a more reliable evaluation.

### Split Ratios

We tested multiple split ratios to ensure optimal model performance:

- **60:20:20**: 60% for training, 20% for validation, and 20% for testing.
- **70:15:15:** 70% for training, 15% for validation, and 15% for testing.
- **80:10:10**: 80% for training, 10% for validation, and 10% for testing.

The splitting was implemented using `train_test_split` function from `scikit-learn` as below:

```python

# Function to split data into train, test and validate

def split_data(df, target_column, test_size=0.2, val_size=0.25, random_state=42):

    train_val_df, test_df = train_test_split(df, test_size=test_size, random_state=random_state, stratify=df[target_column])

    train_df, val_df = train_test_split(train_val_df, test_size=val_size, random_state=random_state, stratify=train_val_df[target_column])

    return train_df, val_df, test_df
```

This resulted in:

- Training set size: 21765
- Validation set size: 7255
- Test set size: 7255

## 6.3.3 Model Building

We iteratively refined the models to improve performance while balancing training and validation metrics, effectively mitigating overfitting and enhancing overall effectiveness. The model building was done using this function:

```python
# Function to train and evaluate models on training data

def train_and_evaluate_model(model, X_train, y_train, X_val,
    y_val):

    model.fit(X_train, y_train)
    y_train_pred = model.predict(X_train)
    y_val_pred = model.predict(X_val)

    y_train_prob = model.predict_proba(X_train)[:, 1]
    y_val_prob = model.predict_proba(X_val)[:, 1]

    metrics = {
        "Accuracy": (accuracy_score(y_train, y_train_pred),
           accuracy_score(y_val, y_val_pred)),

        "Precision": (precision_score(y_train, y_train_pred,
                        average='weighted'), precision_score(
                           y_val, y_val_pred, average='weighted')
                           ),

        "Recall": (recall_score(y_train, y_train_pred,
                    average='weighted'), recall_score(y_val,
                       y_val_pred, average='weighted')),

        "F1-Score": (f1_score(y_train, y_train_pred,
                        average='weighted'), f1_score(y_val,
                           y_val_pred, average='weighted')),

        "AUC": (roc_auc_score(y_train, y_train_prob),
           roc_auc_score(y_val, y_val_prob)),
    }

    return model, metrics
```

Listing 1: Function to Train and Evaluate Models on Training Data

#### 6.3.3.1 Model with Cost-Sensitive Learning

This is the baseline model configured with default settings. To address the class imbalance in the target feature, we set the class weights to “balanced.” This approach ensures that misclassifications of the minority class are penalized more heavily than those of the majority class.

This model serves as the baseline for comparison, allowing us to evaluate whether the performance improves in subsequent iterations.

```
rf_classifier= RandomForestClassifier(
                class_weight='balanced',
                random_state=42)
```

Listing 2: Model with Cost Sensitive Learning

**Model Performance on Training Data**

| Metric | Train | Validation |
|---|---|---|
| Precision | 0.994766 | 0.892021 |
| Recall | 0.994762 | 0.893039 |
| F1-Score | 0.994764 | 0.892013 |
| AUC | 0.999519 | 0.947523 |
| Accuracy | 0.994762 | 0.893039 |

Table 2: Training Metrics for Model with Cost-Sensitive Learning

The training and validation metrics reveal a noticeable disparity, indicating that the model is overfitting. While it performs exceptionally well on the training data, it struggles to generalize to unseen validation data. To address this, we introduced custom hyperparameters in the next iteration to mitigate overfitting.

**Model Performance on Test Data**

| Metric | Score |
|---|---|
| Precision | 0.899629 |
| Recall | 0.900482 |
| F1-Score | 0.899451 |
| AUC | 0.952792 |
| Accuracy | 0.900482 |

Table 3: Test Metrics for Model with Cost-Sensitive Learning

Overfitting caused the model to perform poorly on unseen test data compared to its training performance, highlighting the need for overfitting mitigation in further iterations.

#### 6.3.3.2 Model with Hyperparameter Tuning

In this model, we defined the hyperparameters of the model based on common practice. We set `max_depth` to 10, `min_samples_leaf` to 2 and `min_samples_split` to 5.

```
rf_classifier = RandomForestClassifier(
    n_estimators=100,
    max_depth=10,
    min_samples_split=5,
    min_samples_leaf=2,
    class_weight='balanced',
    random_state=42)
```

Listing 3: Model with Hyperparameter Tuning

| Metric | Train | Validation |
|---|---|---|
| Precision | 0.874330 | 0.863498 |
| Recall | 0.872869 | 0.861159 |
| F1-Score | 0.873456 | 0.862076 |
| AUC | 0.940680 | 0.922775 |
| Accuracy | 0.872869 | 0.861199 |

Table 4: Training Metrics for Model with Hyperparameter Tuning

The Hyperparameter tuned model's performance performance dipped compared to the previous iteration, however it greatly improves the generalization of the model with training and validation metrics being closely aligned. To improve the prediction performance of the model, we then performed a Grid Search to find Best Hyperparameters.

**Model Performance on Test Data**

| Metric | Score |
|---|---|
| Precision | 0.871221 |
| Recall | 0.869745 |
| F1-Score | 0.870342 |
| AUC | 0.930892 |
| Accuracy | 0.869745 |

Table 5: Test Metrics for Model with Hyperparameter Tuning

It is evident by the test data metrics that the overfitting is now resolved, since the test data metrics closesly matches with the training data metrics. However, to improve the prediction performance, further improvements are required.

#### 6.3.3.3 Model with Optimal Hyperparameters

In this model we performed Grid Search with Cross Validation to find out the best hyperparameters.

Grid Search is a hyperparameter tuning technique that systematically tests all possible combinations of hyperparameters to find the optimal configuration for the model.

Cross-Validation is a method used to assess model performance by splitting the dataset into multiple folds, training on some folds, and validating on the others to ensure the model generalizes well to unseen data.

Combining these, Grid Search with Cross-Validation evaluates each hyperparameter combination across multiple folds, ensuring robust hyperparameter tuning while preventing overfitting. This approach was used to fine-tune our Random Forest Classifier for improved accuracy and generalizability.

```
    param_grid = {
    'n_estimators': [50, 75, 100, 125],
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 7],
    'min_samples_leaf': [1, 2, 4]
}

# Performing Grid Search to Find Best Parameters
grid_search = GridSearchCV(
    RandomForestClassifier(class_weight='balanced', random_state
       =42),
    param_grid,
    cv=5,
    scoring='f1'
)

grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
print("Best␣Parameters:", best_params)

results = pd.DataFrame(grid_search.cv_results_)
```

Listing 4: Grid Search For Finding Best Parameters

The grid search returned these hyperparameters as most optimal:

| **Hyperparameter** | **Value** |
|---|---:|
| n_estimators | 125 |
| max_depth | 15 |
| min_samples_split | 2 |
| min_samples_leaf | 1 |
| F1-Score | 0.915831 |

Table 6: Grid Search Results

```
rf_classifier = RandomForestClassifier(**best_params,
    class_weight='balanced', random_state=42)
```

Listing 5: Model with Optimal Hyperparameters

| Metric | Train | Validation |
|---|---|---|
| Precision | 0.918335 | 0.882277 |
| Recall | 0.918217 | 0.882564 |
| F1-Score | 0.918273 | 0.882410 |
| AUC | 0.979180 | 0.943675 |
| Accuracy | 0.918217 | 0.882564 |

Table 7: Training Metrics for Model with Optimal Hyperparameters

The model with optimal hyperparameters demonstrates further improvements in performance compared to previous models. The training and validation metrics are closely aligned and model maintains it's robustness with strong precision, recall, and F1-scores. However, in order to further optimize the performance, we decided to reduce dimensionality of dataset by using Random Forest's Feature Ranking capability.

**Model Performance on Test Data**

| Metric | Score |
|---|---|
| Precision | 0.891807 |
| Recall | 0.892350 |
| F1-Score | 0.892022 |
| AUC | 0.949457 |
| Accuracy | 0.892350 |

Table 8: Test Metrics for Model with Optimal Hyperparameters

Using optimal hyperparameters shows further performance improvements. However, to improve further improvement, we will use Random Forest's Feature Ranking capability to get rid of less important features.

#### 6.3.3.4 Model with Most Important Features (Final Model)

To build this model, we used the model ranking ability of Random Forests and dropped all the features from the dataset which had importance score of less than 0.01.

The top ranking features with importance more than 0.1 were:

| Rank | Feature | Importance |
|---|---|---|
| 1 | lead_time | 0.319113 |
| 2 | no_of_special_requests | 0.150718 |
| 3 | avg_price_per_room | 0.121123 |
| 4 | arrival_month | 0.075094 |
| 5 | arrival_date | 0.055412 |
| 6 | arrival_year | 0.040900 |
| 7 | no_of_week_nights | 0.037613 |
| 8 | market_segment_type_Online | 0.037604 |
| 9 | no_of_weekend_nights | 0.030047 |
| 10 | market_segment_type_Offline | 0.027045 |
| 11 | no_of_adults | 0.021680 |
| 12 | type_of_meal_plan_Meal Plan 2 | 0.012234 |
| 13 | required_car_parking_space | 0.012135 |

Table 9: Feature Importance Ranking Based on Model Output

| Metric | Train | Validation |
|---|---|---|
| Precision | 0.927212 | 0.884819 |
| Recall | 0.926579 | 0.884769 |
| F1-Score | 0.926820 | 0.884794 |
| AUC | 0.984000 | 0.947492 |
| Accuracy | 0.926579 | 0.884769 |

Table 10: Training Metrics of Model with Most Important Features

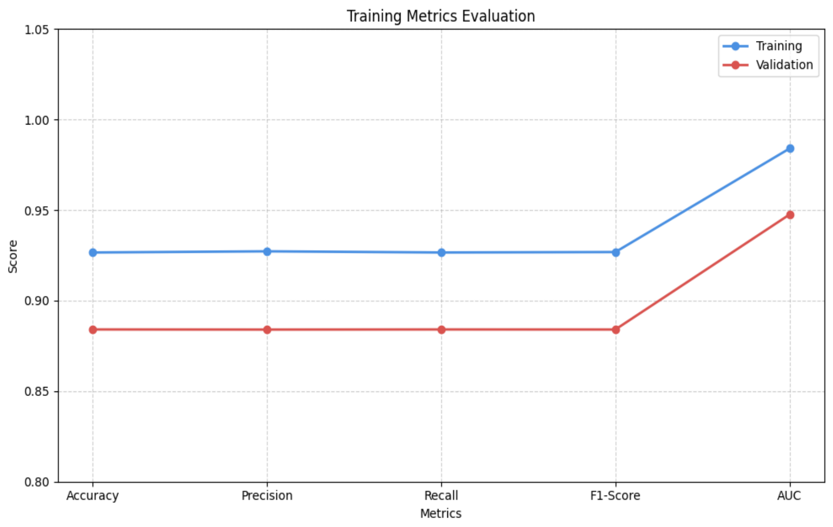

Figure 10: Training Metrics of Model with Most Important Features

This model shows by far the best performance in terms of Recall, Precision and F1-Score on training data with a good balance of prediction performance and generalization.

### Bias Variance Analysis

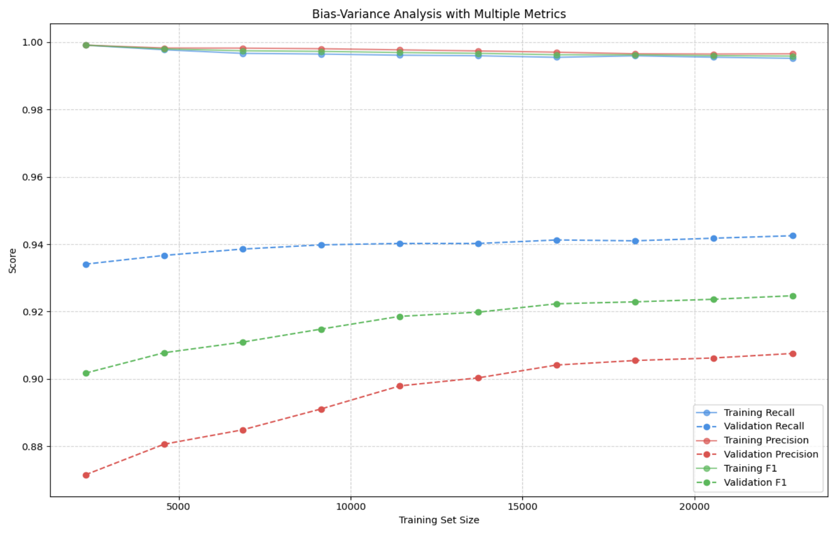

Figure 11: Bias Variance Analysis of Model with Most Important Features

Bias-variance analysis helps assess a model's performance by understanding the trade-off between bias (error due to overly simplistic assumptions) and variance (error due to model sensitivity to small data fluctuations). It reveals whether the model is underfitting, overfitting, or achieving a good balance.

The bias-variance analysis plot for this model shows that the variance is reducing between training and validation as the dataset size is increasing which indicates model is generalizing well.

### Model Performance on Test Data

| **Metric** | **Score** |
|---|---|
| Precision | 0.894105 |
| Recall | 0.894418 |
| F1-Score | 0.894244 |
| AUC | 0.953298 |
| Accuracy | 0.894418 |

Table 11: Test Data Metrics

Choosing most important features to train the model improved the performance even further and this is the best performing model till now.

#### 6.3.3.5 Models with 70:15:15 and 80:10:10 splits

We further trained two more models with varying dataset splits to analyze if this impacts the model performance. Our original model had 60:20:20 split, and the new models were trained on 70:15:15 and 80:10:10 splits for comparison.

**Models' performance on Training Data**

| **Metric** | **Train** | | | **Validate** | | |
|---|---|---|---|---|---|---|
| | **60:20:20** | **70:15:15** | **80:10:10** | **60:20:20** | **70:15:15** | **80:10:10** |
| Precision | 0.927212 | 0.917017 | 0.91636 | 0.884819 | 0.879911 | 0.889316 |
| Recall | 0.926579 | 0.916818 | 0.916194 | 0.884769 | 0.880029 | 0.889733 |
| F1-Score | 0.92682 | 0.916909 | 0.916271 | 0.884794 | 0.879969 | 0.889496 |
| AUC | 0.984 | 0.978359 | 0.977059 | 0.947492 | 0.946677 | 0.949751 |
| Accuracy | 0.926579 | 0.916818 | 0.916194 | 0.884769 | 0.880029 | 0.889733 |

Table 12: Comparison of Training and Validation Metrics Across Splits

**Models' performance on Test Data**

| **Metric** | **60:20:20** | **70:15:15** | **80:10:10** |
|---|---|---|---|
| Precision | 0.894105 | 0.892795 | 0.893516 |
| Recall | 0.894418 | 0.893054 | 0.893605 |
| F1-Score | 0.894244 | 0.892914 | 0.893559 |
| AUC | 0.953298 | 0.952638 | 0.950373 |
| Accuracy | 0.894418 | 0.893054 | 0.893605 |

Table 13: Comparison of Test Metrics Across Splits

These results indicate that the choice of dataset split had a negligible impact on the model's performance.

#### 6.3.3.6 Cross Validation Model

Cross-validation is a technique to evaluate a model's performance and ensure it generalizes well to unseen data. The dataset is divided into multiple folds (subsets), and the model is trained on some folds and tested on the remaining one. This process is repeated across all folds, and the results are averaged to provide a robust estimate of the model's performance. By testing the model on different data splits, cross-validation reduces the risk of overfitting and helps in selecting the best model configuration.

We performed a 5-fold cross-validation to evaluate the performance of the Model with Most Important Features when using Cross Validation Sampling.

| **Metric** | **Cross-Validation (Mean)** | **Holdout (Test Set)** |
|---|---|---|
| Precision | $0.9182 \pm 0.0033$ | 0.8941 |
| Recall | $0.9222 \pm 0.0043$ | 0.8944 |
| F1-Score | $0.9202 \pm 0.0026$ | 0.8942 |
| AUC | $0.9521 \pm 0.0013$ | 0.9533 |
| Accuracy | $0.8924 \pm 0.0035$ | 0.8944 |

Table 14: Comparison of Metrics Between Cross-Validation and Holdout Models

In this comparison, the Cross-Validation metrics represent the mean and standard deviation of the model's performance across 5 folds, while the Holdout Test Set metrics reflect the model's performance on unseen data. The Cross-Validation results exhibit slightly higher scores due to averaging across multiple folds. However, the consistent AUC value validates that the model is robust and performs consistently across both sampling techniques.

## 7 Results and Discussion

The final model's metrics on test data were as follows:

| **Metric** | **Score** |
|---|---|
| Precision | 0.894105 |
| Recall | 0.894418 |
| F1-Score | 0.894244 |
| AUC | 0.953298 |
| Accuracy | 0.894418 |

Table 15: Test Metrics of Final Model

The test metrics indicate the model performed really well with high precision, recall and F1-score.

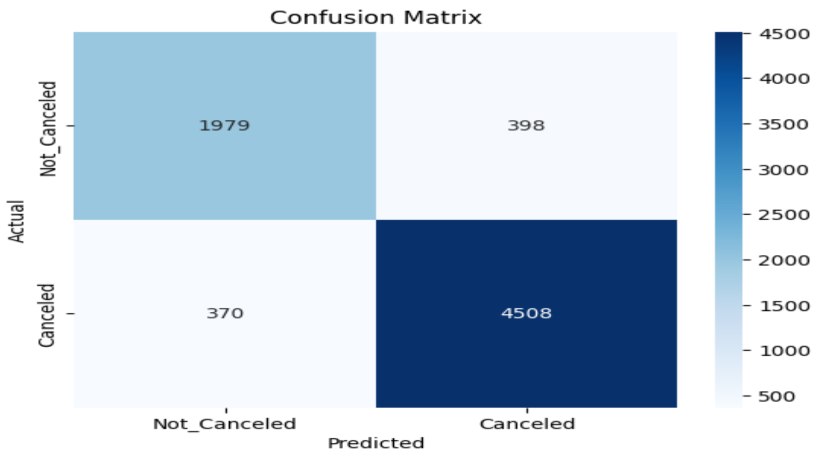

Figure 12: Confusion Matrix

The confusion matrix further confirms that the model performs well in predicting cancellations and non-cancellations. It correctly identifies 4508 cancellations (True Positives) and 1979 non-cancellations (True Negatives). There are 398 false positives and 370 false negatives. This indicates a good balance between minimizing missed cancellations and avoiding unnecessary alerts.

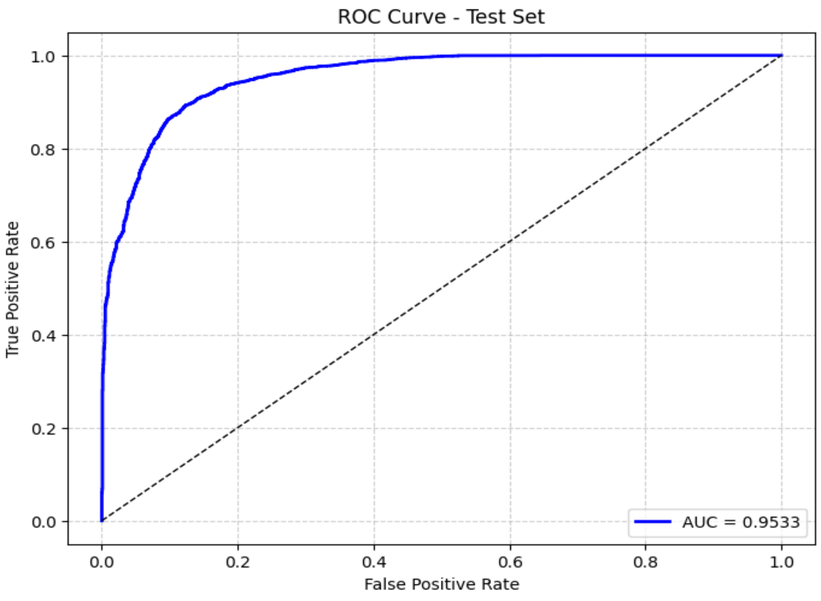

Figure 13: ROC Curve

The ROC curve illustrates the model's ability to distinguish between cancellations and non-cancellations across thresholds, with the True Positive Rate (proportion of actual positives correctly classified as positives) plotted against the False Positive Rate (proportion of actual negatives incorrectly classified as positives). The model's AUC of 0.9533 indicates excellent performance, showing strong discrimination between the two classes and significantly outperforming random guessing (diagonal line).

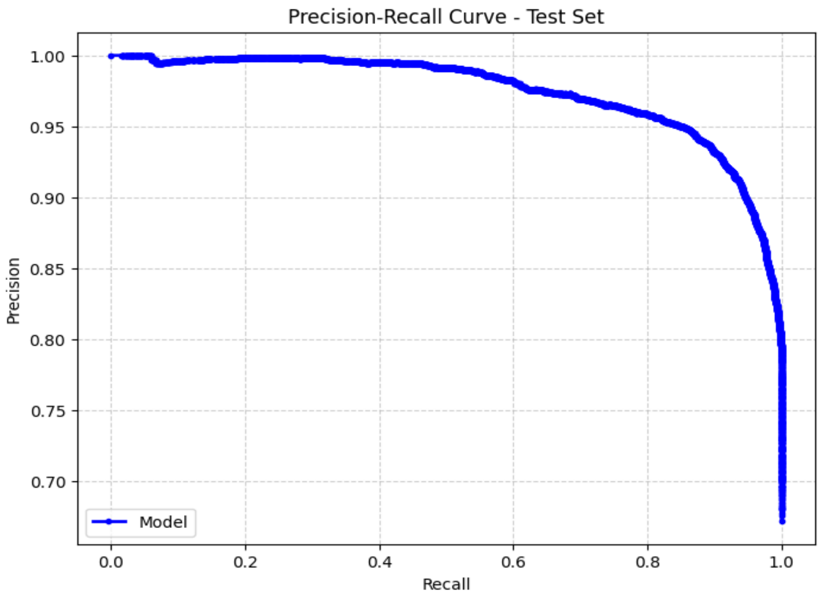

Figure 14: Precision Recall Curve

The Precision-Recall curve shows the trade-off between Precision and Recall for different thresholds. The model maintains high Precision and Recall across most thresholds, with a gradual decline as Recall approaches 1. This indicates the model's strong ability to identify cancellations (high Recall) while minimizing false positives (high Precision).

**Model Performance Comparison**

The table below compares the performance metrics of all the iterations of the model. The improvements in recall and precision are evident, demonstrating the model's effectiveness in handling imbalanced data and predicting cancellations accurately.

| **Model** | **Precision** | **Recall** | **F1-Score** | **AUC** | **Accuracy** |
|---|---|---|---|---|---|
| Model with Cost-Sensitive Learning (Baseline) | 0.8929 | 0.8939 | 0.8928 | 0.9474 | 0.8939 |
| Model with Hyperparameter Tuning | 0.8742 | 0.8727 | 0.8733 | 0.9405 | 0.8727 |
| Model with Optimal Hyperparameters | 0.9183 | 0.9181 | 0.9182 | 0.9793 | 0.9181 |
| Model with Most Important Features (Final Model) | 0.9272 | 0.9266 | 0.9268 | 0.9840 | 0.9266 |
| Model Trained on 70:15:15 Split | 0.9243 | 0.9240 | 0.9241 | 0.9825 | 0.9240 |
| Model Trained on 80:10:10 Split | 0.9254 | 0.9251 | 0.9252 | 0.9831 | 0.9251 |

Table 16: Comparison of Metrics Across Models

The evaluation of models for the classification problem demonstrates that the Model with Most Important Features (Final Model) achieves the best performance, with a Precision of `92.72%` and Recall of `92.66%`, ensuring accurate identification of cancellations without over-prioritizing one metric over the other. The F1-Score of `92.68%` confirms a balanced trade-off between Precision and Recall.

Models trained on different splits (70:15:15 and 80:10:10) showed similar results, indicating the robustness of the approach in handling class imbalance.

## 8 Discussion Topic

**Question :** How does class imbalance affect the performance of the Random Forest model, and does using class balancing improve robustness?

To investigate the impact of addressing class imbalance on model performance, we trained a Random Forest model using the best hyperparameters on the 60:20:20 split without incorporating class balancing (`class_weight`='balanced') and compared it with our Final Model. We compared the metrics for training, validation, and test datasets to see if the metrics most sensitive to class imbalance - Recall and F1-Score show any difference.

| **Metric** | **Class-Weighted Model** | | **No Class Weights** | |
|---|---|---|---|---|
| | **Train** | **Validate** | **Train** | **Validate** |
| Precision | 0.927212 | 0.884819 | 0.924777 | 0.886520 |
| Recall | 0.926579 | 0.884769 | 0.925201 | 0.887664 |
| F1-Score | 0.926820 | 0.884794 | 0.924595 | 0.886492 |
| AUC | 0.984000 | 0.947492 | 0.984793 | 0.947452 |
| Accuracy | 0.926579 | 0.884769 | 0.925201 | 0.887664 |

Table 17: Training Metrics Comparison Between Class-Weighted and No Class Weights Models

| **Metric** | **Class-Weighted (Test)** | **No Weights (Test)** |
|---|---|---|
| Precision | 0.8941 | 0.8941 |
| Recall | 0.8944 | 0.8944 |
| F1-Score | 0.8942 | 0.8942 |
| AUC | 0.9533 | 0.9533 |
| Accuracy | 0.8944 | 0.8944 |

Table 18: Test Metrics Comparison Between Class-Weighted and No Weights Models

The analysis reveals that the differences in metrics between the class-weighted and non-weighted models are minimal during training and validation, suggesting that Random Forest inherently handles class imbalance due to its ensemble nature. On the test set, both models performed similarly, with identical Precision, Recall, and F1-Score. While the class-weighted model slightly improves metrics like Recall and F1-Score during training and validation, it did not drastically outperform the non-weighted model overall.

Random Forest models often handle class imbalance inherently due to their ensemble structure and the way they combine predictions from multiple decision trees. Bootstrapping ensures minority class samples may appear multiple times in the training of individual trees, while measures like Gini impurity or entropy prioritize splits that reduce misclassification for all classes. The aggregation of predictions through majority voting further reduces bias toward the majority class, and randomness in feature selection at each split helps prevent overfitting. While these characteristics make Random Forests robust to imbalance, explicit class weighting can still improve Recall for the minority class, ensuring fairness and sensitivity in critical applications or highly imbalanced datasets.

## References

[1] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[2] G. Biau and E. Scornet, “A random forest guided tour,” *TEST*, vol. 25, no. 2, pp. 197–227, Apr. 2016.