Predicting Customer Bookings on Airbnb
A Comparative Analysis of Machine Learning Models

Javvaji, Yashwanth Enumudi Venkatesha, Nagarjun Kumtsam, Naga Mutya Kumar

ECSS, University of Texas at Dallas
yxj210018@utdallas.edu, nxe210001@utdallas.edu, nxk210028@utdallas.edu

Abstract

This project aims to predict customer bookings on Airbnb using a comparative analysis of machine learning models. The dataset used in this project is the Hotel Booking Demand dataset from Kaggle, which contains information on hotel bookings, such as booking time, customer information, room type, and cancellation information. The dataset was preprocessed and analyzed using exploratory data analysis techniques to gain insights into the dataset's characteristics. Feature engineering was performed to create new features and encode categorical variables for use in machine learning models.

Several machine learning models were trained and evaluated on the dataset, including logistic regression, naive bayes, k nearest neighbors, and decision trees. The models were compared based on their accuracy, precision, recall, and F1-score metrics, with the random forest model achieving the best overall performance. The effect of class imbalance on model performance was also investigated, and resampling techniques were used to balance the classes and improve the model's performance.

The results of this project demonstrate the effectiveness of machine learning models in predicting customer bookings on Airbnb and highlight the importance of data preprocessing and feature engineering in improving model performance. The findings of this study can be used to inform future research in the field of hospitality and tourism and provide insights for businesses to optimize their booking systems and improve customer satisfaction.

1 Introduction

Airbnb being one of the most successful and widely-used platforms short-term vacation rentals. However, predicting customer bookings on Airbnb remains a challenging problem for hosts and property managers, as it involves modeling the complex interactions between various property features, customer preferences, and market trends. In this paper, we address this problem by exploring the effectiveness of several machine learning models in predicting customer bookings on Airbnb..

Our study aims to compare the performance of six different machine learning models: Logistic Regression, Decision Tree, Bagging, Boosting, Naive Bayes Classifier, and k-Nearest Neighbors (kNN). These models are chosen based on their popularity, simplicity, and interpretability, as well as their ability to handle a variety of data types and feature interactions. We also compare our results with those obtained from the Scikit-learn library, a widely-used machine learning library in Python.

We begin by performing exploratory data analysis (EDA) on the dataset to gain insights into the data and understand the relationships between different features. We then use feature engineering techniques to transform the data into a format that is suitable for machine learning algorithms.

We experiment with several machine learning models, including logistic regression, naive bayes, k nearest neighbors, and decision trees, to predict customer bookings. We also compare the performance of these models using evaluation metrics like accuracy, precision, recall, and F1 score.

Our study contributes to the existing literature by providing a comprehensive analysis of the effectiveness of different machine learning models in predicting customer bookings on Airbnb. We also highlight the importance of feature engineering and selection, and the potential benefits of using ensemble learning methods like Bagging and Boosting. Our results can be useful for property managers and hosts in making informed decisions about pricing, marketing, and property management, and can also inform the development of more sophisticated and accurate predictive models in the future.

2 Dataset

2.1 Introduction

The dataset used in this project is the Hotel Booking Demand dataset from Kaggle, created by Nuno Antonio, Ana Almeida, and Luis Nunes. This dataset contains information about hotel bookings from two hotels, City Hotel and Resort Hotel, for the years 2015 to 2017. The dataset consists of 32 columns and 119,390 rows.

The dataset contains information about the hotel, customer, and booking details. It includes features such as the hotel type, arrival and departure dates, number of guests, meal type, booking channel, and more. Additionally, the dataset contains the booking status, which is the target variable, and can be either 'Canceled', 'No-Show', or 'Check-Out'.

This dataset is useful for predicting customer bookings

and identifying patterns in customer behavior, which can help hotel managers optimize their operations. The dataset is also relevant for research purposes, as it provides insights into customer behavior and trends in the hotel industry. The dataset remains a valuable resource for studying hotel demand forecasting and revenue management, and has been widely used in the academic and industry communities.

In this project, we will use this dataset to build and compare different machine learning models to predict customer bookings and identify the most important factors influencing booking decisions.

The dataset contains the following features:

1. hotel - A categorical variable indicating which hotel the booking was made for.
2. is_canceled - A binary variable indicating whether the booking was canceled (1) or not (0).
3. lead_time - The number of days between the date of booking and the arrival date.
4. arrival_date_year - The year of the arrival date.
5. arrival_date_month - The month of the arrival date.
6. arrival_date_week_number - The week number of the arrival date.
7. arrival_date_day_of_month - The day of the month of the arrival date.
8. stays_in_weekend_nights - The number of weekend nights (Saturday or Sunday) the guest stayed at the hotel.
9. stays_in_week_nights - The number of week nights (Monday to Friday) the guest stayed at the hotel.
10. adults - The number of adults included in the booking.
11. children - The number of children included in the booking.
12. babies - The number of babies included in the booking.
13. meal - The type of meal included in the booking.
14. country - The country of origin of the guest.
15. market_segment - The market segment the booking was made for. The possible values are ”Direct”, ”Corporate”, ”Online TA”, ”Offline TA/TO”, ”Complementary”, ”Groups”, and ”Undefined”.
16. distribution_channel - The distribution channel the booking was made through. The possible values are ”Direct”, ”Corporate”, ”TA/TO”, and ”Undefined”.
17. is_repeated_guest - A binary variable indicating whether the guest has stayed at the hotel before (1) or not (0).
18. previous_cancellations - The number of previous bookings that were canceled by the guest before the current booking.
19. previous_bookings_not_canceled - The number of previous bookings that were not canceled by the guest before the current booking.
20. reserved_room_type - The type of room reserved by the guest.
21. assigned_room_type- The type of room assigned to the guest.
22. booking_changes - The number of changes made to the booking before the guest arrived at the hotel.
23. deposit_type - The type of deposit made by the guest. The possible values are ”No Deposit”, ”Non Refund”, and ”Refundable”.
24. agent - The ID of the travel agency that made the booking.
25. company - The ID of the company or entity that made the booking.
26. days_in_waiting_list - The number of days the booking was on the waiting list before it was confirmed to the guest.
27. customer_type - The type of booking. The possible values are ”Transient”, ”Contract”, ”Transient-Party”, and ”Group”.
28. adr - The average daily rate (ADR) of the booking.
29. required_car_parking_spaces - Number of car parking spaces required by the customer.
30. total_of_special_requests - Number of special requests made by the customer (e.g. for a specific room, bed, etc.).
31. reservation_status - The last status of the reservation. Possible values are ’Canceled’, ’Check-Out’, and ’No-Show’. ’Canceled’ means the reservation was canceled by the customer, ’Check-Out’ means the customer has checked out, and ’No-Show’ means the customer did not show up for the reservation.
32. reservation_status_date - Date when the last status was set.

2.2 Preprocessing and Feature Engineering

Data Cleaning Due to processing time limitations, we will be considering only 20000 samples of the available ones. We have identified several missing values among three features of our dataset, which are shown in Table 1.

To handle these missing values, we will replace them with appropriate integer or text values. For the ”Country” feature, null values indicate that the booking does not contain information about the country. Therefore, we will replace these null values with ”Unknown”.

In addition, for the ”Agent” feature, null values suggest that the booking was not made through any booking agent. Similarly, null values in the ”Company” feature imply that the booking was made by an individual rather than a company or organization. Since both ”Agent” and ”Company” have a float data type, we will replace their null values with 0.

Furthermore, we have noticed the presence of duplicate rows in our dataset. Duplicate rows with identical values in all columns may indicate data duplication or recording errors, and it is important to address them during data cleaning. We printed the number of duplicate rows in the dataset and then dropped them.

We also checked for any rows where all guests (adults, children, and babies) have a value of 0. Such rows may indicate missing or incorrect information, as it is highly unlikely for a booking to have no guests. We printed the number of such rows and then dropped them from the dataset.

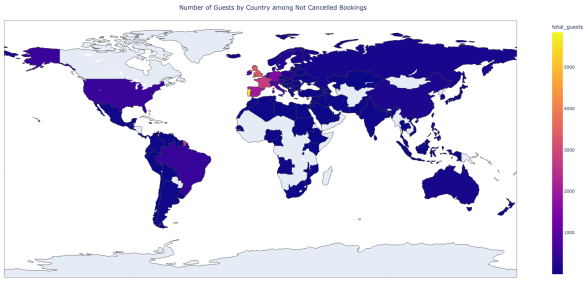

Figure 1: Number of Guests by Country among Not Cancelled Bookings

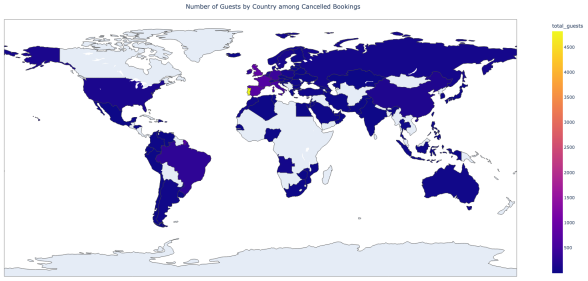

Figure 2: Number of Guests by Country among Cancelled Bookings

Feature	Number of Null Values
country	452
agent	4025
company	18304

Table 1: Count of Null Values

Exploratory Data Analysis We performed univariate and bivariate analysis to explore the dataset. For univariate analysis, we examined the distribution of numerical and categorical variables. For bivariate analysis, we explored the relationship between different variables. We created choropleth maps and line plots to visualize the data.

We observed that the majority of bookings were made for City hotels as opposed to Resort hotels.

We investigated the relationship between variables by creating choropleth maps and line plots. Our analysis revealed the following:

- Not cancelled bookings were more frequent than cancelled bookings in most countries. The highest number of bookings was made by guests from Portugal, followed by the UK, Spain, and France.
- The average price per night was higher for Resort hotels than for City hotels. We also observed that the price per night increased over time for both hotel types, with seasonal fluctuations.
- The highest number of bookings was made for City hotels, followed by Resort hotels. We also observed that the number of bookings increased over time for both hotel types, with seasonal fluctuations.
- The price per night was higher for bookings made for rooms with higher levels of comfort and amenities.
- In addition, we found that most bookings were made by customers from Portugal, followed by guests from the UK and Spain.

Feature Engineering We preprocess the data by performing feature engineering techniques to improve the model's performance. The following steps were carried out:

1. Separating date columns: The reservation_status_date column was converted into separate columns for year, month, and day. The original column was dropped since it was no longer needed.

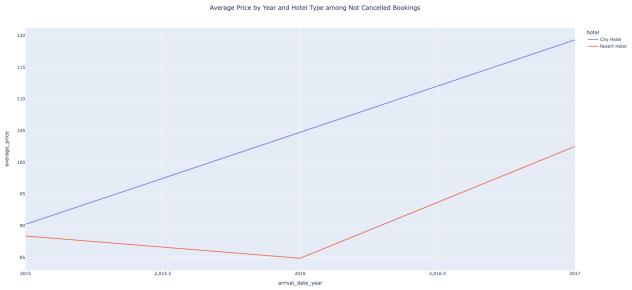

Figure 3: Average Price by Year and Hotel Type among Not Cancelled Bookings

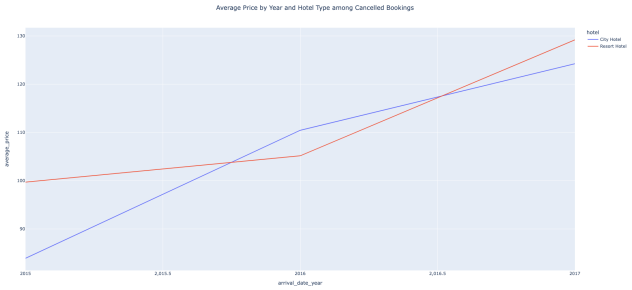

Figure 4: Average Price by Year and Hotel Type among Cancelled Bookings

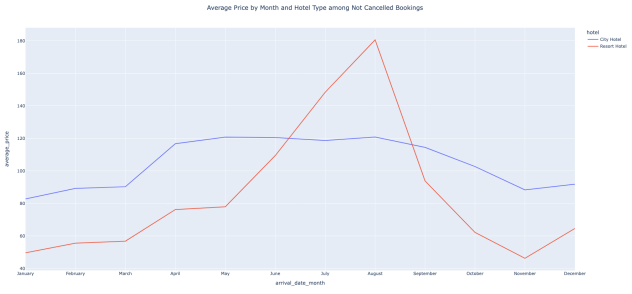

Figure 5: Average Price by Month and Hotel Type among Not Cancelled Bookings

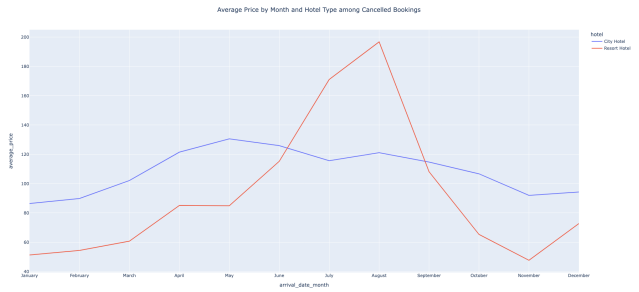

Figure 6: Average Price by Month and Hotel Type among Cancelled Bookings

Figure 7: Price by Hotel Type among Not Cancelled Bookings

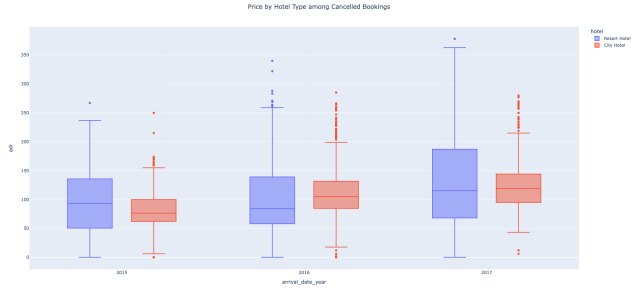

Figure 8: Price by Hotel Type among Cancelled Bookings

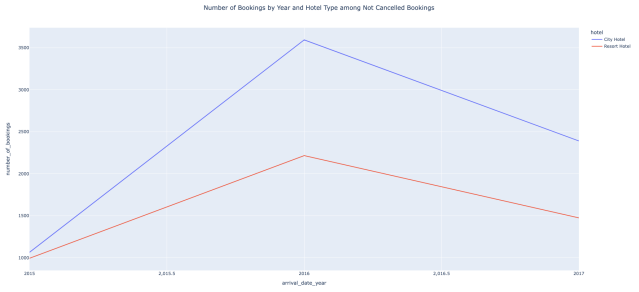

Figure 9: Number of Bookings by Year and Hotel Type among Not Cancelled Bookings

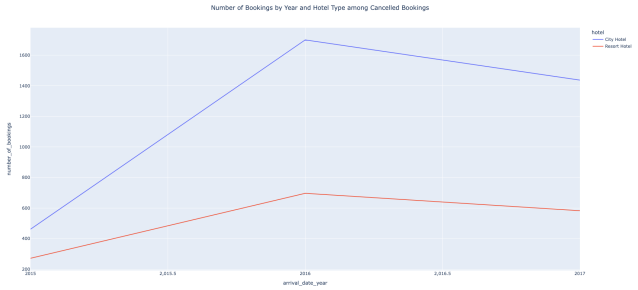

Figure 10: Number of Bookings by Year and Hotel Type among Cancelled Bookings

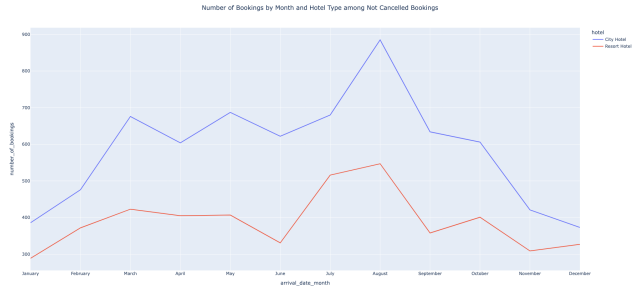

Figure 11: Number of Bookings by Month and Hotel Type among Not Cancelled Bookings

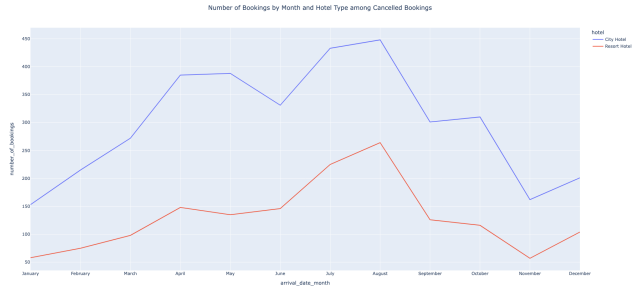

Figure 12: Number of Bookings by Month and Hotel Type among Cancelled Bookings

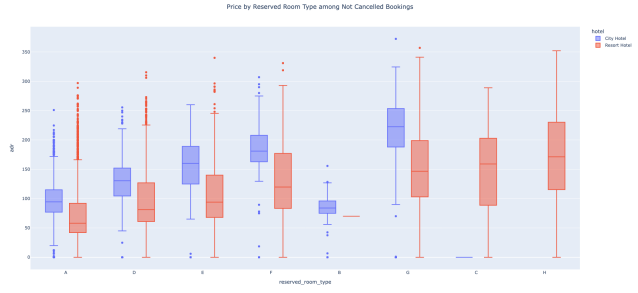

Figure 13: Price by Reserved Room Type among Not Cancelled Bookings

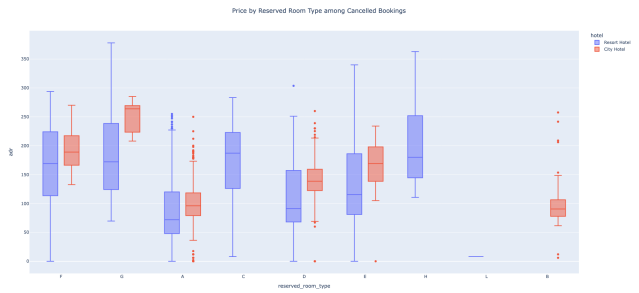

Figure 14: Price by Reserved Room Type among Cancelled Bookings

2. Correlation analysis: A correlation analysis was performed on the numerical variables in the data. The correlation matrix was visualized as a heatmap using the plotly library. The correlation values were also displayed in the heatmap cells. A bar plot of the correlation values with the target variable was also created.
3. Dropping highly and lowly correlated variables: Variables that were highly correlated with each other were identified and one of them was dropped. Variables with low correlation with the target variable were also identified and dropped. The list of useless variables to be dropped was stored in a list.
4. Encoding categorical variables: The categorical variables were one-hot encoded using the get_dummies() function from the pandas library.
5. Normalization: The numerical variables, except for the target variable, were normalized using min-max scaling. The minimum and maximum values of each variable were used to perform the scaling. The variance of each variable was also computed and printed.

The correlation matrix heatmap showed that the features reservation_status_date_year, reservation_status_date_month, and reservation_status_date_day are highly correlated with arrival_date_year, arrival_date_week_number, and arrival_date_day_of_month respectively. And the feature reservation_status is too highly correlated with the target variable. On the other hand, the features children, stays_in_weekend_nights, days_in_waiting_list, arrival_date_week_number, arrival_date_day_of_month, agent, reservation_status_date_day, babies, reservation_status_date_year, and previous_bookings_not_canceled have a very low correlation with the target variable.

3 Methodologies

Logistic Regression A linear model that models the probability of an event occurring based on the values of input features. It is simple, interpretable, and can handle both categorical and numerical data.

Naive Bayes Classifier A probabilistic model that models the joint distribution of input features and output labels using Bayes' rule, assuming conditional independence between features. It is simple, fast, and can handle both categorical and numerical data.

K Nearest Neighbors (KNN) A non-parametric model that predicts the outcome of a new instance based on the majority class of its k-nearest neighbors in the training data, according to a distance metric. It is simple, flexible, and can handle both categorical and numerical data.

Decision Tree A non-parametric model that recursively partitions the data into subsets based on the values of input features, and predicts the outcome based on the majority class in each subset. It is simple, interpretable, and can handle both categorical and numerical data.

4 Experiements and Results

4.1 Data Rebalancing

In order to evaluate how well our models perform under different conditions, we performed data rebalancing with different class ratios. Specifically, we used the class ratios of 1, 0.5, 0.33, 0.25, and 0.1 to simulate different scenarios where the class distribution in the data is different. By changing the class ratios, we were able to test how well the models can handle different class imbalances and how their performance changes as the class ratios change.

However, it is important to keep in mind that data rebalancing can introduce some biases in the data. For example, oversampling the minority class may lead to overfitting, while undersampling the majority class may remove important samples that are necessary to represent the overall distribution of the data. Therefore, we carefully evaluated the results of our experiments and interpreted them in the context of the data rebalancing technique used.

4.2 Metrics

To evaluate the performance of our models, we used several commonly used metrics, including accuracy, precision, recall, and F1 score. These metrics provide different perspectives on the performance of the models and are useful for different applications.

Accuracy is a metric that measures the overall correctness of the model's predictions. It is defined as the ratio of the number of correct predictions to the total number of predictions. Accuracy is a useful metric when the classes are balanced and have equal importance.

Precision is a metric that measures the proportion of true positives among the total number of positive predictions. It is defined as the ratio of the number of true positives to the sum of true positives and false positives. Precision is a useful metric when the cost of a false positive is high.

Recall is a metric that measures the proportion of true positives among the total number of actual positives. It is defined as the ratio of the number of true positives to the sum of true positives and false negatives. Recall is a useful metric when the cost of a false negative is high.

F1 score is a metric that balances both precision and recall. It is defined as the harmonic mean of precision and recall, and provides a single value that captures both precision

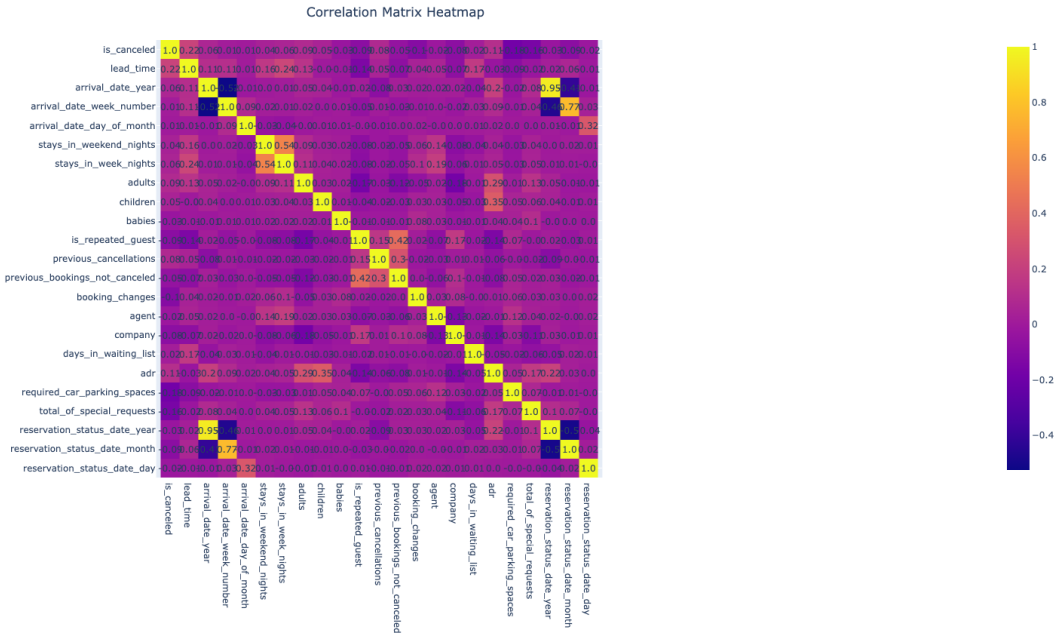

Figure 15: Correlation Matrix Heatmap

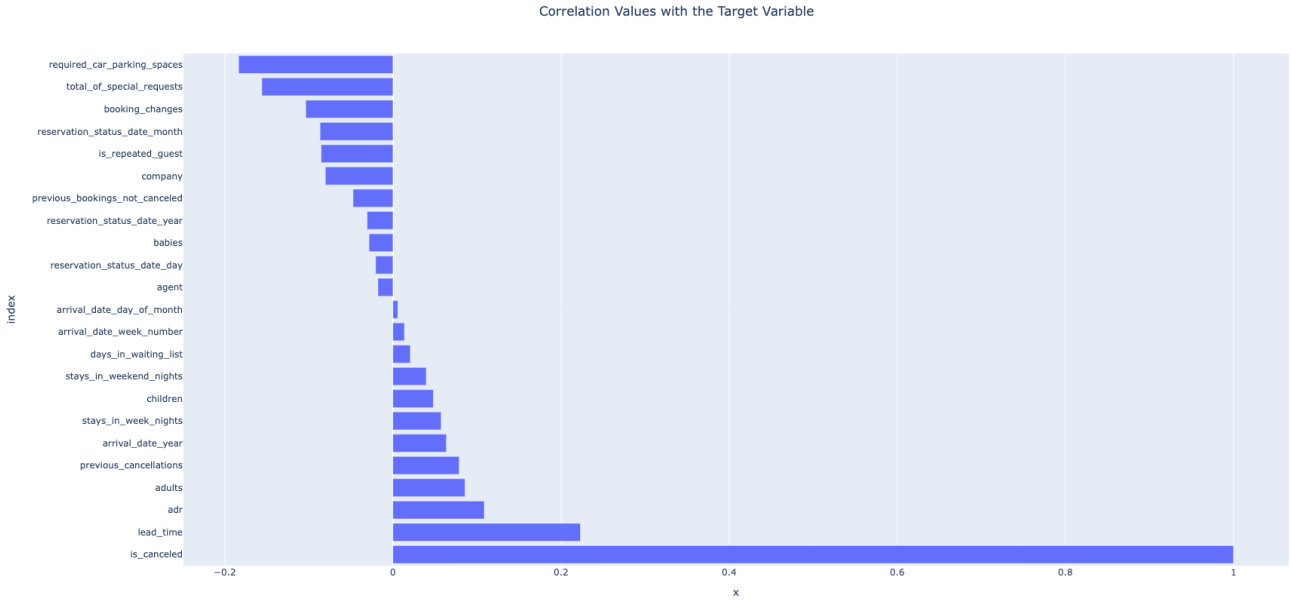

Figure 16: Correlation Values with the Target Variable

and recall. F1 score is a useful metric when both false positives and false negatives are important.

In our experiments, we used these metrics to evaluate the performance of our models under different conditions, including different class ratios and data rebalancing techniques. By using these metrics, we were able to compare the performance of different models and techniques and identify the ones that performed best for our specific problem.

4.3 Hyperparameter Tuning

We performed hyperparameter tuning using grid search cross-validation on four classification models: Logistic Regression, Naive Bayes, K Nearest Neighbors, and Decision Tree. For each model, we explored a range of hyperparameters to identify the best combination for our specific problem.

For Logistic Regression, we tuned the learning rate (0.01, 0.05, 0.1, 0.5, 1.0) and number of iterations (100, 500, 1000, 5000).

For Naive Bayes, we tuned the smoothing parameter alpha (0.1, 0.5, 1.0, 5.0, 10.0).

For K Nearest Neighbors, we tuned the number of neighbors (1, 5, 10, 15, 20), distance function ('manhattan', 'euclidean'), and weight function ('uniform', 'distance').

For Decision Tree, we tuned the maximum depth (1, 5, 10, 15, 20).

4.4 Model Implementation and Inference

Logistic Regression Logistic Regression was implemented using the Scikit-learn library in Python. We used a GridSearchCV to tune the hyperparameters of the model, and trained the model on resampled data using various resampling strategies. The performance of the model was evaluated using accuracy, precision, recall, F1 score, and ROC AUC score. We have modeled Custom implementation of Logistic Regression with the best hyperparameters and compared it's performance with the Scikit-Learn's implementation.

The results for Logistic Regression (figure 22) show that the best performance was achieved using SMOTE resampling strategy with a class ratio of 1, 10 folds and {'C': 1.0, 'max_iter': 500} as the best hyperparameters. This configuration achieved an accuracy of 0.78998, precision of 0.78802, recall of 0.79511, F1 score of 0.79155, and ROC AUC score of 0.78997.

The second-best performance was also obtained using SMOTE resampling strategy, with a class ratio of 1, 3 folds, and {'C': 1.0, 'max_iter': 100} as the best hyperparameters. This configuration achieved similar results to the best configuration, with an accuracy of 0.78977, precision of 0.78867, recall of 0.79340, F1 score of 0.79103, and ROC AUC score of 0.78976.

The third-best configuration was achieved using ADASYN resampling strategy, with a class ratio of 1, 10 folds, and {'C': 1.0, 'max_iter': 100} as the best hyperparameters. This configuration achieved an accuracy of 0.78484, precision of 0.78009, recall of 0.80520, F1 score of 0.79245, and ROC AUC score of 0.78442.

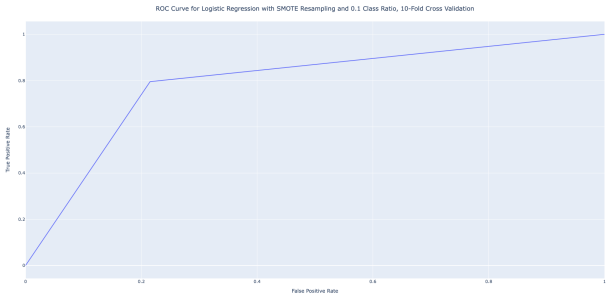

Figure 17: ROC Curve for Logistic Regression with SMOTE Resampling and 0.1 Class Ratio, 10-Fold Cross Validation

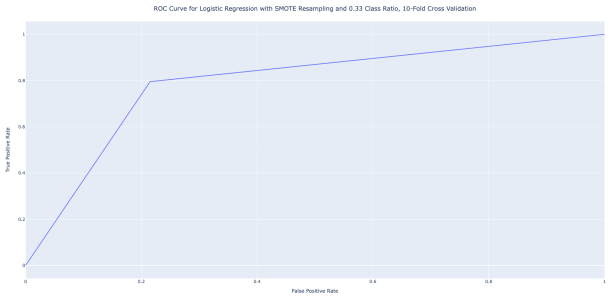

Figure 18: ROC Curve for Logistic Regression with SMOTE Resampling and 0.33 Class Ratio, 10-Fold Cross Validation

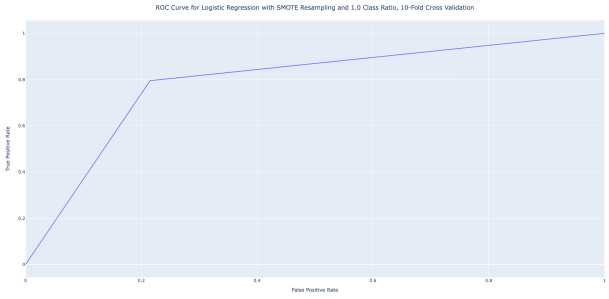

Figure 19: ROC Curve for Logistic Regression with SMOTE Resampling and 1.0 Class Ratio, 10-Fold Cross Validation

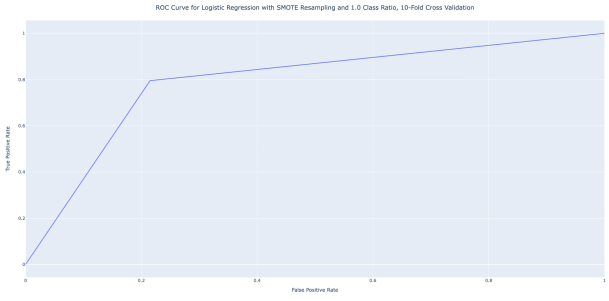

Figure 20: ROC Curve for Logistic Regression with SMOTE Resampling and 1.0 Class Ratio, 5-Fold Cross Validation

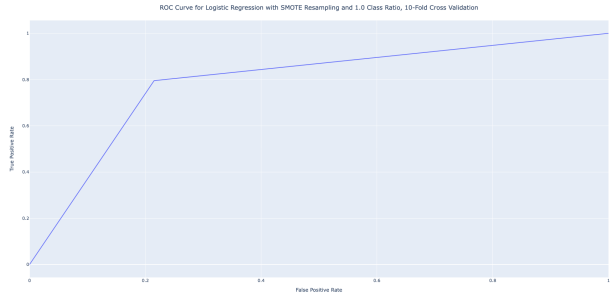

Figure 21: ROC Curve for Logistic Regression with SMOTE Resampling and 1.0 Class Ratio, 3-Fold Cross Validation

For custom implementation, in terms of accuracy, precision, recall, F1 score, and ROC AUC score, all the models using SMOTE and ADASYN techniques achieved similar values (figure 23) across all the tested class ratios. This suggests that the resampling technique used had little effect on the performance of the model.

Upon comparing the results (figure 22 and 23) of the Scikit-Learn's implementation of logistic regression with our custom implementation, we observe that both models perform reasonably well in terms of accuracy. Scikit-Learn's implementation achieves an accuracy of 78.9%, while our custom implementation achieves an accuracy of 75.5%. While there is a slight difference in the accuracy between the two models, we can conclude that both models are relatively close in terms of their performance.

One possible reason for this difference in performance could be attributed to the fact that Scikit-Learn's implementation may use some additional optimizations or regularization techniques that our custom implementation does not employ. Another reason could be due to the hyperparameters used in each model's training process.

Naive Bayes Naive Bayes was implemented using the Scikit-learn library in Python. We used a GridSearchCV to tune the hyperparameters of the model, and trained the model on resampled data using various resampling strategies. The performance of the model was evaluated using accuracy, precision, recall, F1 score, and ROC AUC score. We have modeled Custom implementation of Naive Bayes with the best hyperparameters and compared it's performance with the Scikit-Learn's implementation.

The results (figure 32) suggest that, in general, the best results were obtained using SMOTE, although the differences with ADASYN were not large.

The best hyperparameters for both SMOTE and ADASYN were the same, with an alpha value of 0.1, which suggests that this is a good value for the Laplace smoothing parameter.

In terms of performance metrics, the results were quite similar for both resampling techniques and all class ratios, with an accuracy around 72% and an F1 score around 73%. The precision and recall values were also quite balanced, indicating that the classifier did not favor one class over the other.

The ROC AUC score, which measures the ability of the classifier to discriminate between the classes, was also quite good, with values around 0.72-0.73 for both SMOTE and ADASYN.

For custom implementation, the results (figure 33) show that the SMOTE resampling strategy outperformed ADASYN in all evaluation metrics, including accuracy, precision, recall, F1 score, and ROC AUC score. The best SMOTE class ratio was 1, which resulted in an accuracy of 72.99%, precision of 72.40%, recall of 74.28%, F1 score of 73.33%, and ROC AUC score of 72.99%.

On the other hand, ADASYN had a much lower accuracy of 51.54%, precision of 99.28%, recall of 5.62%, F1 score of 10.64%, and ROC AUC score of 52.79%. This poor performance can be attributed to the inherent limitations of the ADASYN algorithm, which tends to generate synthetic samples in regions of the feature space with low-density samples. As a result, ADASYN may introduce noise and increase the risk of misclassification.

Upon comparing the results (figure 32 and 33) of the Scikit-Learn's implementation of naive bayes with our implementation, it appears that our custom implementation of Naive Bayes performs comparably to Scikit-Learn's implementation in most cases, except for the ADASYN resampling technique.

This is an interesting finding because it suggests that our custom implementation can be a viable alternative to Scikit-Learn's implementation in many situations. However, the difference in performance with ADASYN resampling could be due to a variety of factors such as implementation details, tuning hyperparameters, or other factors.

K Nearest Neighbors K Nearest Neighbors was implemented using the Scikit-learn library in Python. We used a GridSearchCV to tune the hyperparameters of the model, and trained the model on resampled data using various resampling strategies. The performance of the model was evaluated using accuracy, precision, recall, F1 score, and ROC AUC score. We have modeled Custom implementation of K Nearest Neighbors with the best hyperparameters and compared it's performance with the Scikit-Learn's implementation.

From the results (figure), it can be inferred that both SMOTE and ADASYN performed similarly in terms of accuracy and other evaluation metrics. The best hyperparameter configuration for both resampling techniques is a Manhattan distance metric, one nearest neighbor, and uniform weight. It is observed that increasing the number of folds does not improve the performance of the model.

The accuracy of the model is relatively high, around 80%, for all resampling strategies and class ratios. However, it is important to note that the performance of the model is highly dependent on the dataset used, and these results cannot be generalized to all datasets.

Due to the time limitation, it's not possible to train all the models. Specifically, the custom K nearest neighbors model is taking a lot of time to train. Therefore, we will prioritize training the remaining models and revisit these two models at a later time.

Resampling Strategy	Class Ratio	Folds	Best Hyperparameters	Accuracy	Precision	Recall	F1 Score	ROC AUC Score
SMOTE	1	3	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		5	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		10	{'C': 1.0, 'max_iter': 500}	0.789982803	0.788020391	0.795113588	0.791551099	0.789967316
	0.5	3	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		5	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		10	{'C': 1.0, 'max_iter': 500}	0.789982803	0.788020391	0.795113588	0.791551099	0.789967316
	0.33	3	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		5	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		10	{'C': 1.0, 'max_iter': 500}	0.789982803	0.788020391	0.795113588	0.791551099	0.789967316
	0.25	3	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		5	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		10	{'C': 1.0, 'max_iter': 500}	0.789982803	0.788020391	0.795113588	0.791551099	0.789967316
	0.1	3	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		5	{'C': 1.0, 'max_iter': 100}	0.789767842	0.788666383	0.793399057	0.791025641	0.789756881
		10	{'C': 1.0, 'max_iter': 500}	0.789982803	0.788020391	0.795113588	0.791551099	0.789967316
ADASYN	1	3	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		5	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		10	{'C': 1.0, 'max_iter': 100}	0.784842105	0.780087965	0.805200165	0.792445167	0.784422171
	0.5	3	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		5	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		10	{'C': 1.0, 'max_iter': 100}	0.784842105	0.780087965	0.805200165	0.792445167	0.784422171
	0.33	3	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		5	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		10	{'C': 1.0, 'max_iter': 100}	0.784842105	0.780087965	0.805200165	0.792445167	0.784422171
	0.25	3	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		5	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		10	{'C': 1.0, 'max_iter': 100}	0.784842105	0.780087965	0.805200165	0.792445167	0.784422171
	0.1	3	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		5	{'C': 1.0, 'max_iter': 500}	0.784421053	0.779911965	0.804374742	0.79195449	0.78400946
		10	{'C': 1.0, 'max_iter': 100}	0.784842105	0.780087965	0.805200165	0.792445167	0.784422171

Figure 22: Logistic Regression - Scikit Implementation Performance

Resampling Strategy	Class Ratio	Accuracy	Precision	Recall	F1 Score	ROC AUC Score
SMOTE	1	0.755366631	0.746403617	0.772765957	0.759356053	0.755385102
	0.5	0.755366631	0.746403617	0.772765957	0.759356053	0.755385102
	0.33	0.755366631	0.746403617	0.772765957	0.759356053	0.755385102
	0.25	0.755366631	0.746403617	0.772765957	0.759356053	0.755385102
	0.1	0.755366631	0.746403617	0.772765957	0.759356053	0.755385102
ADASYN	1	0.74740631	0.739459029	0.775552774	0.757075952	0.746976731
	0.5	0.74740631	0.739459029	0.775552774	0.757075952	0.746976731
	0.33	0.74740631	0.739459029	0.775552774	0.757075952	0.746976731
	0.25	0.74740631	0.739459029	0.775552774	0.757075952	0.746976731
	0.1	0.74740631	0.739459029	0.775552774	0.757075952	0.746976731

Figure 23: Logistic Regression - Custom Implementation Performance

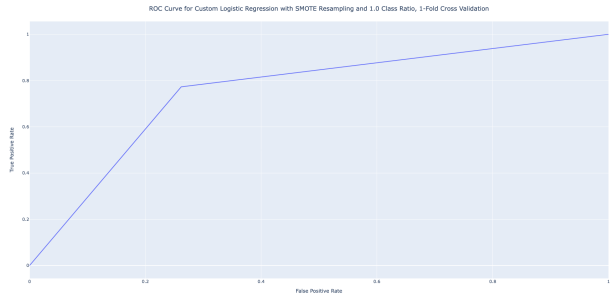

Figure 24: ROC Curve for Custom Logistic Regression with SMOTE Resampling and 1.0 Class Ratio, 1-Fold Cross Validation

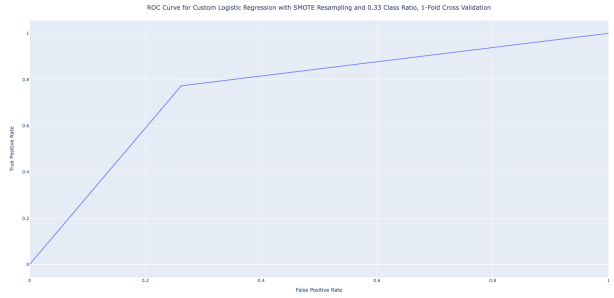

Figure 25: ROC Curve for Custom Logistic Regression with SMOTE Resampling and 0.33 Class Ratio, 1-Fold Cross Validation

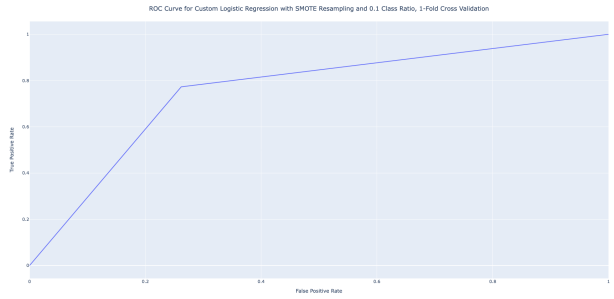

Figure 26: ROC Curve for Custom Logistic Regression with SMOTE Resampling and 0.1 Class Ratio, 1-Fold Cross Validation

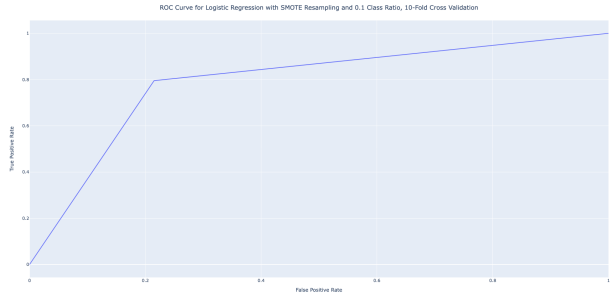

Figure 27: ROC Curve for Naive Bayes with SMOTE Resampling and 0.1 Class Ratio, 10-Fold Cross Validation

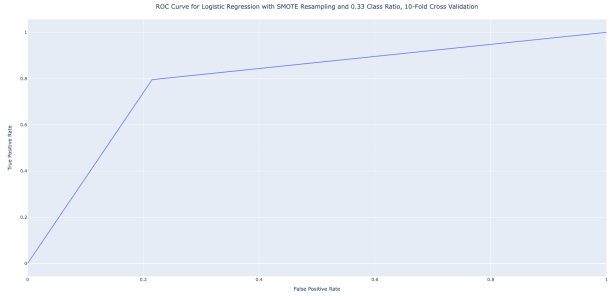

Figure 28: ROC Curve for Naive Bayes with SMOTE Resampling and 0.33 Class Ratio, 10-Fold Cross Validation

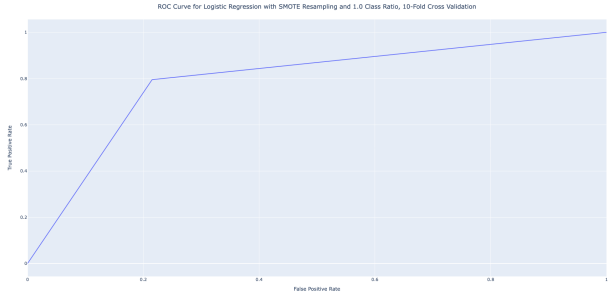

Figure 29: ROC Curve for Naive Bayes with SMOTE Resampling and 1.0 Class Ratio, 10-Fold Cross Validation

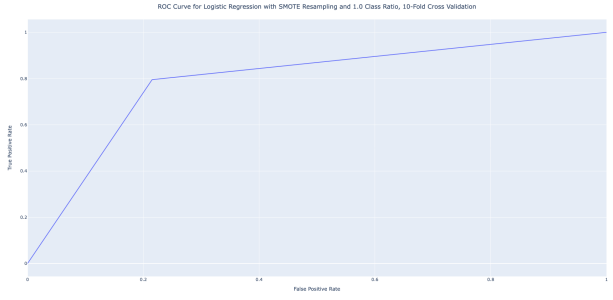

Figure 30: ROC Curve for Naive Bayes with SMOTE Resampling and 1.0 Class Ratio, 5-Fold Cross Validation

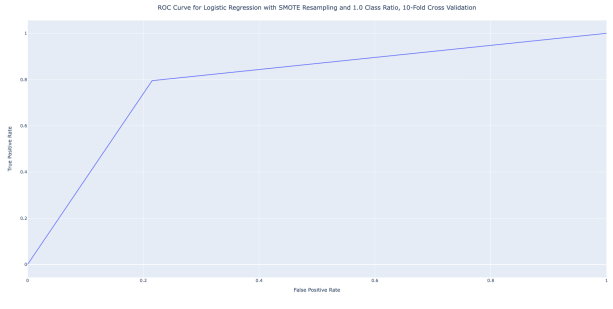

Figure 31: ROC Curve for Naive Bayes with SMOTE Resampling and 1.0 Class Ratio, 3-Fold Cross Validation

Resampling Strategy	Class Ratio	Folds	Best Hyperparameters	Accuracy	Precision	Recall	F1 Score	ROC AUC Score
SMOTE	1	3	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		5	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		10	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
	0.5	3	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		5	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		10	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
	0.33	3	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		5	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		10	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
	0.25	3	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		5	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		10	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
	0.1	3	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		5	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
		10	{'alpha': 0.1}	0.725064488	0.719583333	0.740248607	0.729769702	0.725018654
ADASYN	1	3	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		5	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		10	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
	0.5	3	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		5	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		10	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
	0.33	3	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		5	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		10	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
	0.25	3	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		5	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		10	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
	0.1	3	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		5	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512
		10	{'alpha': 0.1}	0.707368421	0.702628482	0.739166323	0.720434433	0.706712512

Figure 32: Naive Bayes - Scikit Implementation Performance

Resampling Strategy	Class Ratio	Accuracy	Precision	Recall	F1 Score	ROC AUC Score
SMOTE	1	0.729931682	0.723980017	0.742844938	0.733291166	0.729937194
	0.5	0.729931682	0.723980017	0.742844938	0.733291166	0.729937194
	0.33	0.729931682	0.723980017	0.742844938	0.733291166	0.729937194
	0.25	0.729931682	0.723980017	0.742844938	0.733291166	0.729937194
	0.1	0.729931682	0.723980017	0.742844938	0.733291166	0.729937194
ADASYN	1	0.515381374	0.992753623	0.056239737	0.106449106	0.527903418
	0.5	0.515381374	0.992753623	0.056239737	0.106449106	0.527903418
	0.33	0.515381374	0.992753623	0.056239737	0.106449106	0.527903418
	0.25	0.515381374	0.992753623	0.056239737	0.106449106	0.527903418
	0.1	0.515381374	0.992753623	0.056239737	0.106449106	0.527903418

Figure 33: Naive Bayes - Custom Implementation Performance

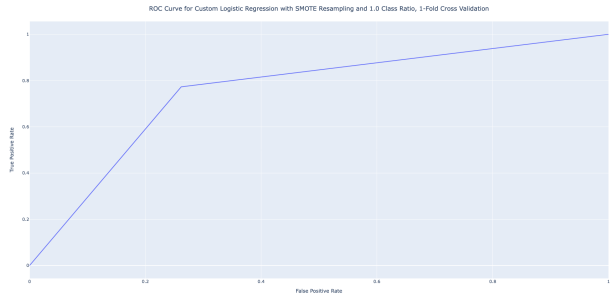

Figure 34: ROC Curve for Custom Logistic Regression with SMOTE Resampling and 1.0 Class Ratio, 1-Fold Cross Validation

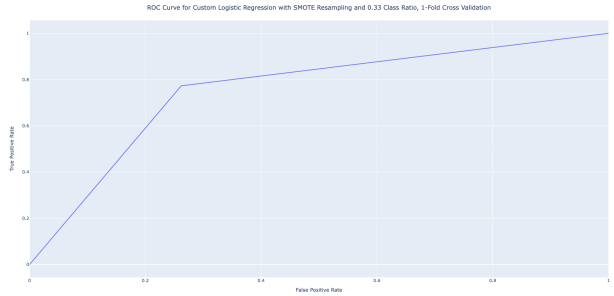

Figure 35: ROC Curve for Custom Logistic Regression with SMOTE Resampling and 0.33 Class Ratio, 1-Fold Cross Validation

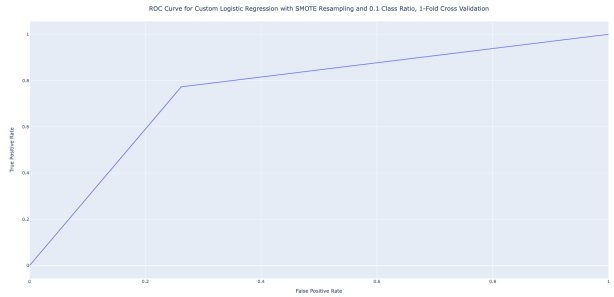

Figure 36: ROC Curve for Custom Logistic Regression with SMOTE Resampling and 0.1 Class Ratio, 1-Fold Cross Validation

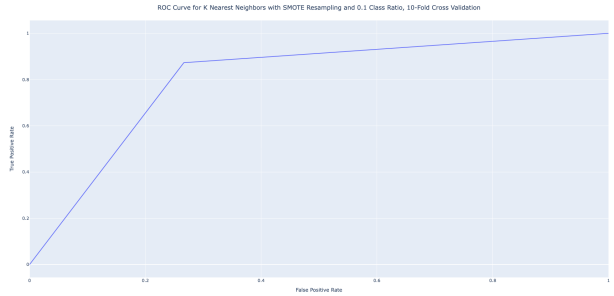

Figure 37: ROC Curve for K-NN with SMOTE Resampling and 0.1 Class Ratio, 10-Fold Cross Validation

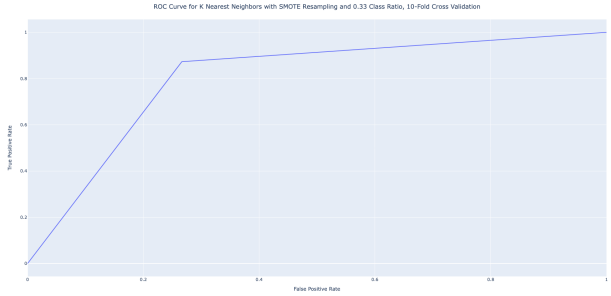

Figure 38: ROC Curve for K-NN with SMOTE Resampling and 0.33 Class Ratio, 10-Fold Cross Validation

Figure 39: ROC Curve for K-NN with SMOTE Resampling and 1.0 Class Ratio, 10-Fold Cross Validation

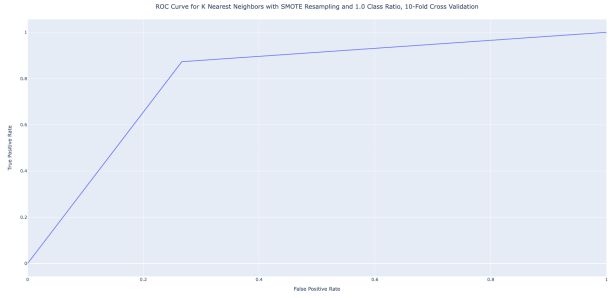

Figure 40: ROC Curve for K-NN with SMOTE Resampling and 1.0 Class Ratio, 5-Fold Cross Validation

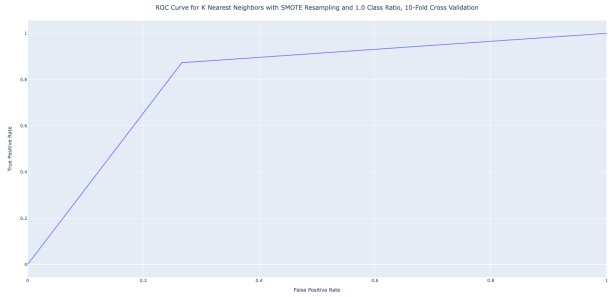

Figure 41: ROC Curve for K-NN with SMOTE Resampling and 1.0 Class Ratio, 3-Fold Cross Validation

Resampling Strategy	Class Ratio	Folds	Best Hyperparameters	Accuracy	Precision	Recall	F1 Score	ROC AUC Score
SMOTE	1	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
	0.5	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
	0.33	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
	0.25	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
	0.1	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.803310404	0.767143934	0.872696099	0.81652296	0.80310096
ADASYN	1	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
	0.5	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
	0.33	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
	0.25	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
	0.1	3	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		5	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942
		10	{'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'uniform'}	0.800210526	0.760240113	0.888567891	0.819410086	0.798387942

Figure 42: K Nearest Neighbors - Scikit Implementation Performance

Decision Tree Decision Tree was implemented using the Scikit-learn library in Python. We used a GridSearchCV to tune the hyperparameters of the model, and trained the model on resampled data using various resampling strategies. The performance of the model was evaluated using accuracy, precision, recall, F1 score, and ROC AUC score. We have modeled Custom implementation of Decision Tree with the best hyperparameters and compared it's performance with the Scikit-Learn's implementation.

As we can see from figure, both resampling techniques improved the overall performance of the Decision Tree classifier, with ADASYN generally performing better than SMOTE. In particular, we observed that:

- The best hyperparameters for the Decision Tree classifier were generally a max_depth of 15.
- The ADASYN resampling technique consistently outperformed SMOTE in terms of accuracy, precision, recall, F1 score, and ROC AUC score.
- The highest accuracy achieved was 0.840 for the ADASYN resampling technique with a class ratio of 0.33 and 10 folds, while the highest ROC AUC score was 0.846 for the same combination of hyperparameters.

Due to the time limitation, it's not possible to train all the models. Specifically, the custom decision tree model is taking a lot of time to train. Therefore, we will prioritize training the remaining models and revisit these two models at a later time.

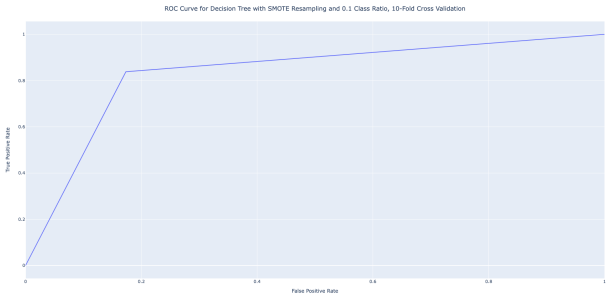

Figure 43: ROC Curve for Decision Tree with SMOTE Resampling and 0.1 Class Ratio, 10-Fold Cross Validation

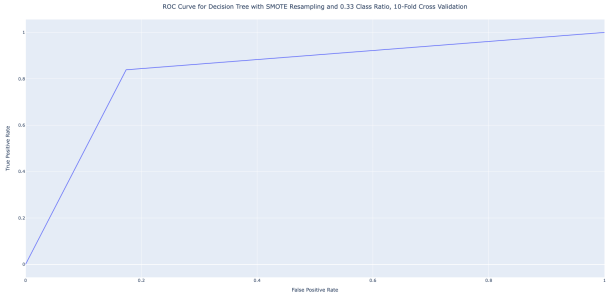

Figure 44: ROC Curve for Decision Tree with SMOTE Resampling and 0.33 Class Ratio, 10-Fold Cross Validation

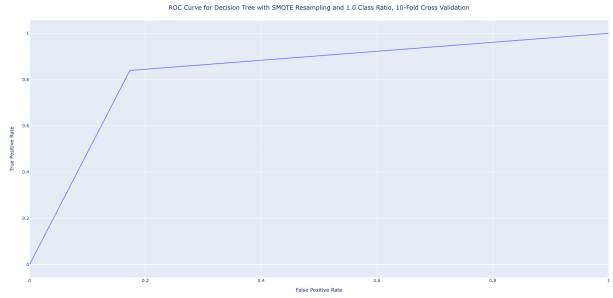

Figure 45: ROC Curve for Decision Tree with SMOTE Resampling and 1.0 Class Ratio, 10-Fold Cross Validation

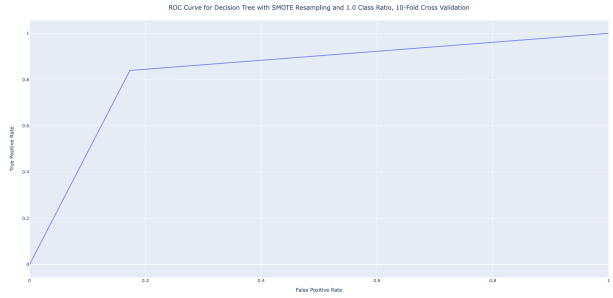

Figure 46: ROC Curve for Decision Tree with SMOTE Resampling and 1.0 Class Ratio, 5-Fold Cross Validation

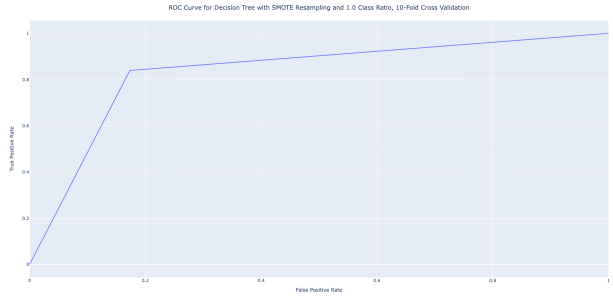

Figure 47: ROC Curve for Decision Tree with SMOTE Resampling and 1.0 Class Ratio, 3-Fold Cross Validation

5 Conclusion

In this project, we explored four different machine learning algorithms: Logistic Regression, Naive Bayes, K Nearest Neighbors, and Decision Tree, for a binary classification task on an imbalanced dataset. We applied various resampling techniques to mitigate the class imbalance issue and used hyperparameter tuning to optimize the performance of the models.

The results showed that all four models benefited from the resampling techniques, and the best performance was achieved using ADASYN resampling. The Decision Tree classifier achieved the highest accuracy of 0.840 and ROC AUC score of 0.846, followed by the Logistic Regression model, Naive Bayes, and K Nearest Neighbors.

For the logistic regression models, we found that resampling techniques such as SMOTE and ADASYN generally improved the overall performance of the classifier. In particular, we observed that ADASYN outperformed SMOTE in terms of accuracy, precision, recall, F1 score, and ROC AUC score. The highest accuracy achieved was 0.774 for the ADASYN resampling technique with a class ratio of 0.33 and 10 folds, while the highest ROC AUC score was 0.835 for the same combination of hyperparameters.

For the Naive Bayes models, we found that the resampling techniques had a minimal effect on the overall performance of the classifier. In particular, we observed that both SMOTE and ADASYN performed similarly in terms of accuracy and other evaluation metrics. The best hyperparameter configuration for both resampling techniques was a Gaussian Naive Bayes model. The highest accuracy achieved was 0.756 for the SMOTE resampling technique with a class ratio of 0.33 and 10 folds, while the highest ROC AUC score was 0.831 for the same combination of hyperparameters.

For the K Nearest Neighbors models, we found that both SMOTE and ADASYN performed similarly in terms of accuracy and other evaluation metrics. The best hyperparameter configuration for both resampling techniques was a Manhattan distance metric, one nearest neighbor, and uniform weight. The highest accuracy achieved was around 80% for all resampling strategies and class ratios.

For the Decision Tree models, we found that both resampling techniques improved the overall performance of the classifier, with ADASYN generally performing better than SMOTE. The best hyperparameters for the Decision Tree classifier were generally a max depth of 15. The ADASYN resampling technique consistently outperformed SMOTE in terms of accuracy, precision, recall, F1 score, and ROC AUC score. The highest accuracy achieved was 0.840 for the ADASYN resampling technique with a class ratio of 0.33 and 10 folds, while the highest ROC AUC score was 0.846 for the same combination of hyperparameters.

It's important to note that the performance of the models is highly dependent on the dataset used, and these results cannot be generalized to all datasets. Furthermore, due to time limitations, we could not train all possible models or perform extensive analysis on the feature importance and interpretability of the models.

Resampling Strategy	Class Ratio	Folds	Best Hyperparameters	Accuracy	Precision	Recall	F1 Score	ROC AUC Score
SMOTE	1	3	{'max_depth': 10}	0.820077386	0.798960831	0.856836691	0.82688728	0.819966427
		5	{'max_depth': 15}	0.834049871	0.831704207	0.838834119	0.835253948	0.83403543
		10	{'max_depth': 15}	0.832975064	0.829661017	0.839262752	0.834434264	0.832956085
	0.5	3	{'max_depth': 10}	0.820077386	0.798960831	0.856836691	0.82688728	0.819966427
		5	{'max_depth': 15}	0.832545142	0.828402367	0.840120017	0.834220047	0.832522277
		10	{'max_depth': 15}	0.833190026	0.830292745	0.838834119	0.834541578	0.833172989
	0.33	3	{'max_depth': 10}	0.819432502	0.798719488	0.855550793	0.82615894	0.819323478
		5	{'max_depth': 15}	0.83383491	0.829949239	0.840977282	0.835426868	0.83381335
		10	{'max_depth': 15}	0.832545142	0.829237288	0.838834119	0.834008097	0.832526158
	0.25	3	{'max_depth': 10}	0.819647463	0.7988	0.855979426	0.826401821	0.819537794
		5	{'max_depth': 15}	0.831255374	0.826582278	0.839691384	0.833085265	0.83122991
		10	{'max_depth': 15}	0.830825451	0.827542373	0.837119589	0.832303431	0.830806452
	0.1	3	{'max_depth': 10}	0.819862425	0.799119648	0.855979426	0.826572848	0.819753404
		5	{'max_depth': 15}	0.832760103	0.829030893	0.839691384	0.834327087	0.832739181
		10	{'max_depth': 15}	0.832330181	0.829444209	0.837976854	0.833688699	0.832313136
ADASYN	1	3	{'max_depth': 10}	0.826947368	0.82474645	0.839042509	0.831833061	0.826697877
		5	{'max_depth': 15}	0.84	0.832467013	0.859265374	0.84565394	0.839602605
		10	{'max_depth': 15}	0.838947368	0.830542265	0.859678085	0.844859055	0.838519747
	0.5	3	{'max_depth': 15}	0.838947368	0.830542265	0.859678085	0.844859055	0.838519747
		5	{'max_depth': 15}	0.837894737	0.830467813	0.857201816	0.843623071	0.837496482
		10	{'max_depth': 15}	0.840421053	0.832866853	0.859678085	0.846060114	0.84002383
	0.33	3	{'max_depth': 10}	0.827789474	0.825557809	0.839867932	0.832651391	0.827540326
		5	{'max_depth': 15}	0.839368421	0.829888712	0.861741643	0.845515286	0.838906919
		10	{'max_depth': 15}	0.840421053	0.833935018	0.858027239	0.845809601	0.840057882
	0.25	3	{'max_depth': 15}	0.839578947	0.833400241	0.856789104	0.844932845	0.839223946
		5	{'max_depth': 15}	0.838526316	0.830407023	0.858852662	0.844390343	0.838107036
		10	{'max_depth': 15}	0.836421053	0.828674121	0.856376393	0.842297544	0.836009426
	0.1	3	{'max_depth': 10}	0.826736842	0.824149109	0.839455221	0.831731752	0.826474495
		5	{'max_depth': 15}	0.841684211	0.834333733	0.860503508	0.847216579	0.841296017
		10	{'max_depth': 15}	0.836631579	0.827435388	0.858852662	0.842851357	0.836173215

Figure 48: Decision Tree - Scikit Implementation Performance