

AI Insights on Passenger Satisfaction

23/24, Artificial Intelligence

CSI_6_ARI_23-24

Student ID: 4008609

Submission date 19/04/24

VIDEO LINK: <https://youtu.be/mIZOJnKefKk>

Word count of ToC, References, Appendix, etc: 1727 words



London South Bank University
Department of Computer Science, Engineering

Table of Contents

List of Figures	6
List of Tables	7
Abstract	8
1. Introduction.....	9
1.1 Aims and objectives	9
1.2 Economic and Social Impact	9
1.3 Methodology	10
1.4 Background information + some formulas	11
2. Dataset.....	11
2.2 Exploring the dataset size	12
2.3. Exploring top five rows	12
2.4 Dataset Insight (mean, std , kurtosis, skewness, etc).....	13
2.6. Identifying missing values	18
2.7. Checking for duplicate entries	19
2.8. Identifying Outliers	19
2.8.1. Numerical outliers (Box Plot)	19
2.8.2. Categorical outliers (Histograms)	20
2.9. Class imbalances	24
2.10. Exploratory analysis (unsupervised)	26
2.10.1 Correlation matrix (Numerical Data)	26
2.10.2. Correlation matrix (Categorical Data)	27
2.10.3. PI charts (Categorical Data)	29
3. Data Pre-Processing.....	31
3.1 Dropping irrelevant column (ID)	31
3.2 Imputing missing values	31
3.3 Converting to correct datatype	32
3.4 Train/test split	34
3.4 Encoding categorical values.....	35
3.6 Data standardisation	36

4. Algorithms	37
4.1 Discriminant Analysis	37
4.2 Linear SVM	37
4.3 Logistic Regression	38
4.4 RBF SVM	39
4.5 Gradient-Boosted Trees	39
5. Implementation / EXPERIMENTATION/ Model development (AND SOME RESULTS)	40
5.1 Discriminant Analysis	40
5.1.1 Best Model	40
5.1.2 Worst Model	41
5.1.3. Experiment Outcome (Discriminant Analysis)	41
5.2 Linear SVM	42
5.2.1 Best Model	42
5.2.2 Worst Model	43
5.2.3 Experiment Outcome (Linear SVM)	43
5.3 Logistic Regression	44
5.3.1 Best Model	44
5.3.2 Worst Model	46
5.3.3 Experiment Results (Logistic Regression)	46
5.4 RBF SVM	47
5.4.1 Best Model	47
5.4.2 Worst Model	47
5.4.3 Experiment Results (RBF SVM)	48
5.5 Gradient-Boosted Trees	49
5.5.1 Best Model	49
5.5.2. Worst Model	50
5.5.3 Experiment Results (Gradient-Boosted Trees)	51
6. Results	52
6.1 Discriminant Analysis (Best)	52
6.2 Linear SVM (Best)	52
6.3 Logistic Regression (Best)	53
6.4 RBF SVM (Best)	53
6.5 Gradient booster classifier (Best)	54

6.6 Best Predictors.....	55
7. Discussion.....	56
7.1. Unsupervised learning insight	56
7.2. Supervised learning insight	56
8. Conclusion	56
Suggestions to Japan's bullet train firm include:	56
9. Self Reflection	56
References	57
10. Appendix (Proof of experimentation)	58
10.1 Discriminant Analysis.....	58
10.1.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation).....	58
10.1.1.1. Label Encoding Without SMOTE	58
10.1.1.2. Label Encoding With SMOTE	58
10.1.1.3. One-Hot Encoding Without Smote	59
10.1.1.4. One-Hot Encoding With SMOTE	59
10.1.2. WITH Hyper Tuning (SVD + LSQR + SMOTE).....	60
10.1.2.1. One-Hot Encoding + SVD + SMOTE.....	60
10.1.2.2. One-Hot Encoding + LSQR + SMOTE	60
10.2 Linear SVM	61
10.2.1. No Hyper Tuning (Label Encoding, SMOTE)	61
10.2.1.1. Label Encoding Without SMOTE	61
10.2.1.2. Label Encoding With SMOTE	61
10.2.1.3. One-Hot Encoding Without Smote.....	62
10.2.1.4. One-Hot Encoding With SMOTE	62
10.2.2. WITH Hyper Tuning (SVD + LSQR + SMOTE).....	63
10.2.2.1. One-Hot Encoding + GridSearch (c) + SMOTE.....	63
10.2.2.2. One-Hot Encoding + Random Search (tol) + SMOTE.....	64
10.3 Logistic Regression	65
10.3.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation).....	65
10.3.1.1. Label Encoding Without SMOTE	65
10.3.1.2. Label Encoding With SMOTE	65
10.3.1.3. One Hot encoding without SMOTE	66
10.3.1.4. One Hot encoding With SMOTE.....	66

10.1.2. WITH Hyper Tuning (SVD + LSQR + SMOTE)	66
10.1.2.1 One-Hot + GridSeach(c) + SMOTE	66
10.4 RBF SVM.....	68
10.4.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation).....	68
10.4.1.1. Label Encoding Without SMOTE	68
10.4.1.2. Label Encoding With SMOTE	68
10.4.1.3. One Hot encoding without SMOTE	69
10.4.1.4. One Hot encoding With SMOTE	69
10.4.2. WITH Hyper Tuning (Random Search + Grid Search + SMOTE)	70
10.4.2.1. ONEHOT + RANDOMSEARCH + PCA	70
10.4.2.2. ONEHOT + GRIDSEARCH + PCA	71
10.5 Gradient-Boosted Trees.....	72
10.5.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation).....	72
10.5.1.1. Label Encoding Without SMOTE	72
10.5.1.2. Label Encoding With SMOTE	72
10.5.1.3. OneHot Encoding Without SMOTE.....	73
10.5.1.3. OneHot Encoding With SMOTE.....	73
10.2.2. WITH Hyper Tuning (One Hot + Random Search)	74

List of Figures

Figure 1: Methodology used	10
Figure 2: Importing dataset CSV using pandas	11
Figure 3: Dataset size/dimensions	12
Figure 4: Overview of dataset structure	12
Figure 5: Basic statistics of numerical columns	13
Figure 6: Advanced statistics of numerical columns	14
Figure 7: Datatype of all columns (before preprocessing)	15
Figure 8: Identifying count and percentage of missing values in each column	18
Figure 9: Checking for duplicate values	19
Figure 10: Identifying potential numerical outliers	20
Figure 11: Identifying potential categorical outliers + further insight to data distribution	23
Figure 12: Class imbalances	15
Figure 13: Correlation matrix for numerical data	26
Figure 14: Correlation matrix for categorical data	27
Figure 15: PI charts	30
Figure 16: Dropping irrelevant column	31
Figure 17: Imputing missing values using mode	31
Figure 18: Converting datatypes	33
Figure 19: Splitting train/test data	34
Figure 20: Initial (Label Encoding)	35
Figure 21 Final Decision (One-Hot encoding)	36
Figure 22: Data Scaling	36
Figure 23: Modelling plan	40
Figure 24: Best LDA Model	41
Figure 25: Worst LDA Model	41
Figure 25: Best Linear SVM Model	42
Figure 26: Worst Linear SVM Model	43

Figure 27: Best Logistic Regression Model	45
Figure 28: Best Logistic Regression Model	46
Figure 29: Best RBF SVM Model	47
Figure 30: Worst RBF SVM Model	47
Figure 31: Best Gradient-Boosted Trees Model	49
Figure 32: Worst Gradient-Boosted Trees Model	50
Figure 33: LDA Results Final	52
Figure 34: Linear SVM Results Final	52
Figure 35: Logistic Regression Results Final	53
Figure 36: RBF SVM Results Final	53
Figure 37: Gradient Booster Results Final	54
Figure 38: GBC Importance predictors of Overall_Exprience	55

List of Tables

Table 1: Data Type insight and correction	17
Table 2: Correlation matrix insight	28
Table 3: LDA experiment results	41
Table 4: Linear SVM experiment results	44
Table 5: Logistic Regression SVM experiment results	46
Table 6: RBF SVM experiment results	48

Abstract

This report conducts unsupervised analysis and gains insight; Travel delays significantly decrease passenger experience, a comfortable seat leads to a higher perception of catering, most customers are loyal business travellers. Supervised learning trains five models, best being Gradient booster trees. Trained model suggests the onboard experience is the best predictor of a customer's overall experience.

Gradient booster classifier performance scores include ROC and Precision-Recall of 0.99, accuracy of 0.94, precision, recall and F1 of 0.95.

1. Introduction

The reliability and efficiency of Japan's renowned bullet train services has come under scrutiny due to a significant drop in customer satisfaction. This decline is primarily attributed to travel delays caused by strike actions and suboptimal performance. A comprehensive dataset collected from passenger surveys and travel data forms the basis of this analysis.

This project will utilise unsupervised learning, and supervised learning algorithms **Discriminant Analysis, Linear SVM, Logistic Regression, RBF SVM, and Gradient-Boosted Trees** to gain an insight into the factors contributing to a decline in customer satisfaction/experience.

The goal is to provide feedback to Japan's bullet train network on how to enhance passenger's experience.

1.1 Aims and objectives

The **aim** of this project is to develop five machine learning models that can accurately predict customer satisfaction levels for Arasaka Electric Railway based on the provided data. The models will help identify key factors influencing passenger experience and guide improvements in service quality, which would in turn increase the company's revenue.

The **objectives** of this project are:

- Perform exploratory data analysis to identify key variables and understand their distributions.
- Preprocess data through handling missing values, encoding categorical variables, and normalizing/scaling .
- Split the dataset into training and testing.
- Tune hyperparameters for each model to optimise their performance.
- Implement cross-validation techniques as a mean to prevent overfitting.
- Outline the implications of the model findings and how they can be used to improve customer satisfaction and the company's efficiency.

1.2 Economic and Social Impact

Economically, higher customer satisfaction boosts ridership and revenue, which in turn attracts more regular users and investors. It also supports fare stability and growth through positive customer perceptions. **Socially**, it promotes the use of public transport, which in turn reduces the usage on private vehicle, resulting in a decrease in emissions.

1.3 Methodology

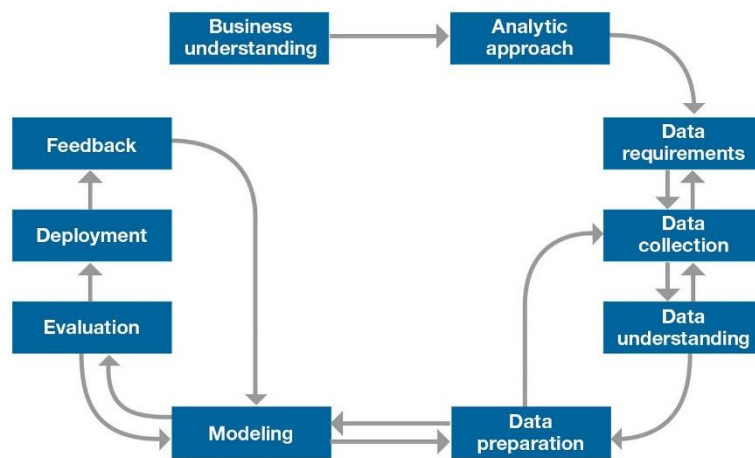


Figure 1: Methodology used

The image (see figure 1) outlines the CRISP-DM methodology. The methodology is used in this report to complete the business analysis.

1. Business Understanding: Define the project objectives and requirements from a business perspective.
2. Analytic Approach: Decide on the suitable data analytics strategies to tackle the business problem.
3. Data Requirements: Identify the data sets necessary for the analysis.
4. Data Collection: Gather the required data.
5. Data Understanding: Analyse the data to gain insight
6. Data Preparation: Clean data for modelling.
7. Modelling: Build and evaluate models utilising various algorithms.
8. Evaluation: Evaluate the model to aids business objectives.

1.4 Background information + some formulas

SMOTE -> creating synthetic data to balance classes (only train set) .

$$\text{Accuracy} = \frac{\text{True Positives (TP)} + \text{True Negatives (TN)}}{\text{Total Population}}$$

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

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

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

2. Dataset

2.1 Importing the dataset

```
[3] df = pd.read_csv ('CSI_6_ARI_CW_23_24_25.csv')  
✓ 0.1s
```

Figure 2: Importing dataset CSV using pandas

This code (see figure 1) reads a CSV file into a pandas DataFrame object. The `pd.read_csv()` function is imported from the pandas library.

2.2 Exploring the dataset size

```
print(f"Dataset Shape: {df.shape}")
print(f"Number Of Rows: {df.shape[0]}")
print(f"Number Of Columns: {df.shape[1]}")
```

[6] ✓ 0.0s

... Dataset Shape: (30000, 25)
Number Of Rows: 30000
Number Of Columns: 25

Figure 3: Dataset size/dimensions

The code (see figure 2) outputs the rows and columns of the dataset. `df.shape[0]` accesses the first element of the shape tuple, which represents the number of rows, while `df.shape[1]` accesses the second element, representing the number of columns. The dataset has 30000 rows and 25 columns.

2.3. Exploring top five rows

```
df.head()
```

[7] ✓ 0.0s

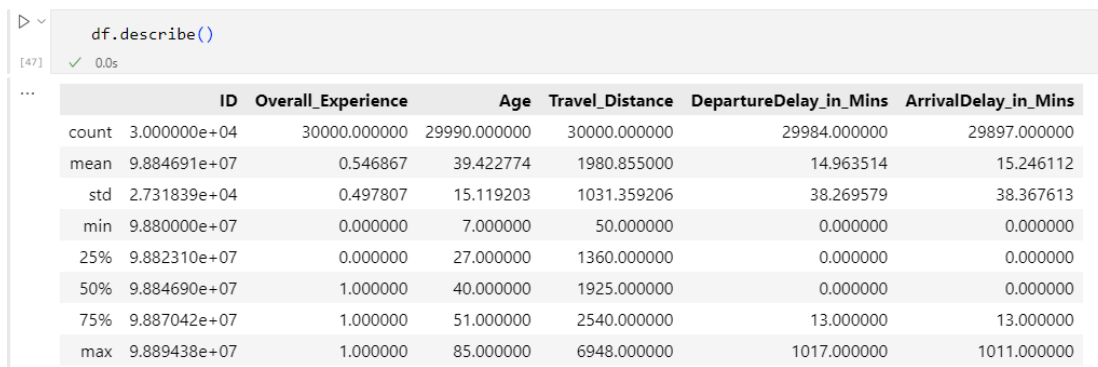
	ID	Overall_Experience	Seat_comfort	Seat_Class	Arrival_time_convenient	Catering	Platform_location	Onboardwifi_service	Onboard_entertainment	Online_support
0	98813062	1	good	Green Car	good	good	Convinient	excellent	excellent	excellent
1	98871729	0	acceptable	Ordinary	acceptable	acceptable	Convinient	excellent	acceptable	excellent
2	98893437	1	acceptable	Green Car	acceptable	NaN	manageable	good	good	excellent
3	98860750	0	poor	Ordinary	poor	poor	need improvement	excellent	poor	excellent
4	98803703	0	acceptable	Ordinary	acceptable	acceptable	manageable	acceptable	acceptable	acceptable

5 rows × 25 columns

Figure 4: Overview of dataset structure

As expected (see figure 2), the dataset has 25 columns. Only the top five rows have been outputted for clearer presentation. The dataset includes an ID column which is to be dropped (see section 3.1) as it is likely irrelevant for supervised predictive modelling.

2.4 Dataset Insight (mean, std , kurtosis, skewness, etc)



```
df.describe()
```

	ID	Overall_Experience	Age	Travel_Distance	DepartureDelay_in_Mins	ArrivalDelay_in_Mins
count	3.000000e+04	30000.000000	29990.000000	30000.000000	29984.000000	29897.000000
mean	9.884691e+07	0.546867	39.422774	1980.855000	14.963514	15.246112
std	2.731839e+04	0.497807	15.119203	1031.359206	38.269579	38.367613
min	9.880000e+07	0.000000	7.000000	50.000000	0.000000	0.000000
25%	9.882310e+07	0.000000	27.000000	1360.000000	0.000000	0.000000
50%	9.884690e+07	1.000000	40.000000	1925.000000	0.000000	0.000000
75%	9.887042e+07	1.000000	51.000000	2540.000000	13.000000	13.000000
max	9.889438e+07	1.000000	85.000000	6948.000000	1017.000000	1011.000000

Figure 5: Basic statistics of numerical columns

Notes:

1. **Overall_Experience:** mean is about 0.55 indicating that more than half of the experiences are positive.
2. **Travel_Distance:** The range of distance travelled is very large (50 to 6948), and of mean 1980.85 that is greater than the median of 1925, **therefore indicating a right skew**. This may suggest suggests that there are outliers or long-distance customers that skew the average upwards.
3. **DepartureDelay_in_Mins:** The mean and standard deviation are relatively low compared to the maximum value, which is extremely high (1017 minutes). The 75th percentile is only 13 minutes, suggesting that **extreme delays are rare but have a significant impact on the average delay time**.
4. **ArrivalDelay_in_Mins:** The arrival delays show a similar pattern to departure delays, with most being non-existent (median of 0) but with **some extreme values that are skewing the mean upwards**.

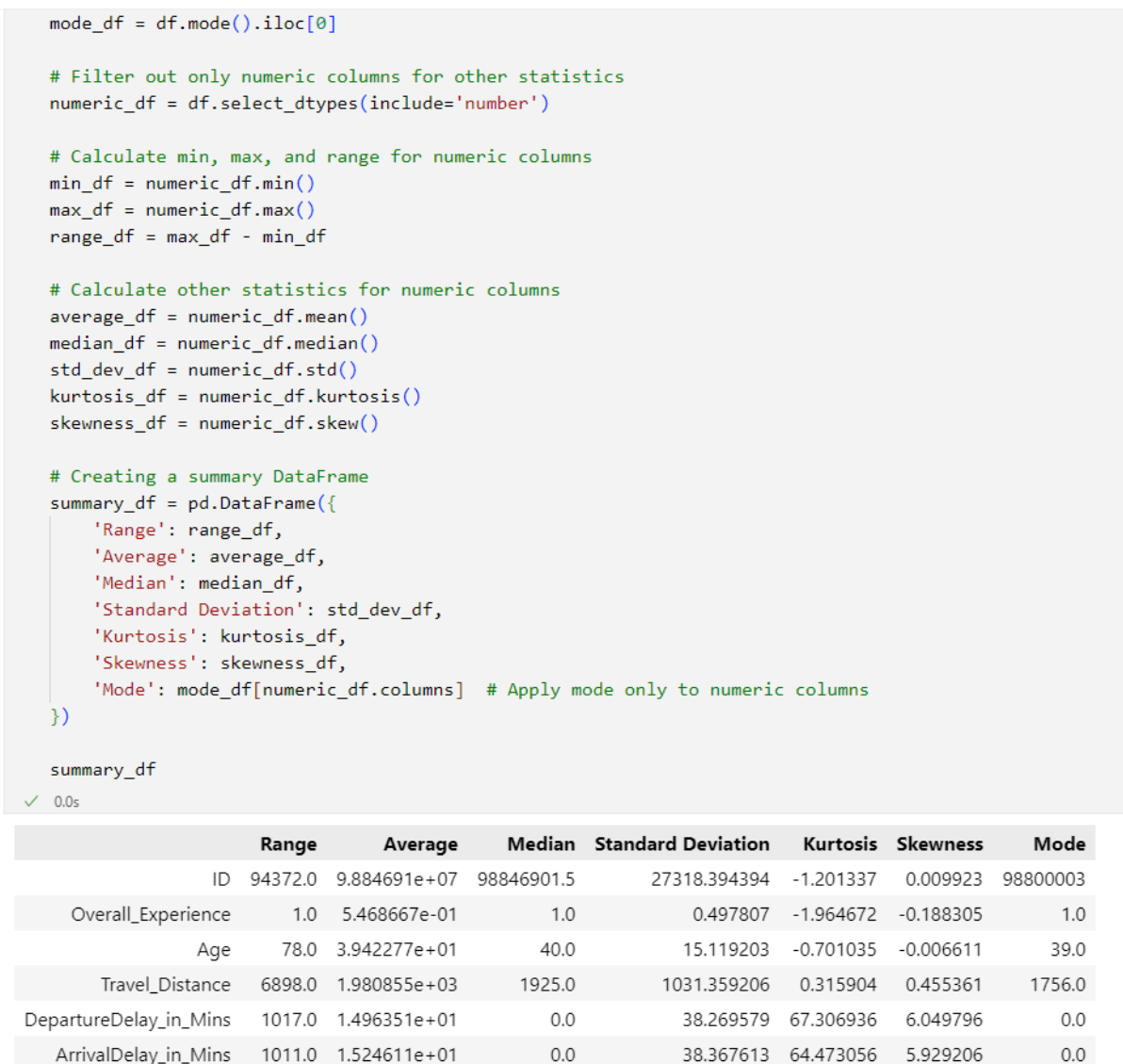


Figure 6: Advanced statistics of numerical columns

Notes:

1. **Travel_Distance:** There's a **significant range** (6898), and the median (1925) being less than the mean (about 1980) indicates a right-skewed distribution. The mode being a very high value suggests that a particular travel distance is very common, or it may be a default or outlier value.
2. **DepartureDelay_in_Mins:** The range is **very large** (1017), indicating **significant variability in departure delays**. The mean is around 1.49, which is not very informative without knowing the unit of measure. **A median of 0 and mode of 0 suggest that most flights were not delayed.**

3. **ArrivalDelay_in_Mins**: Similar to departure delays, **there's a large range (1011)**. The median and **mode of 0** again suggest that most arrivals were on time, with some outliers causing the average to be around 1.52.
4. **ID**: The range of ID numbers is also high, further **suggesting they are not ordinal** or contain any time-series information

2.5. Exploring datatypes

```

df.info()
[26] ✓ 0.0s

... <class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
#   Column                                Non-Null Count  Dtype
---  -
0   ID                                    30000 non-null  int64
1   Overall_Experience                   30000 non-null  int64
2   Seat_comfort                         29985 non-null  object
3   Seat_Class                           30000 non-null  object
4   Arrival_time_convenient              27198 non-null  object
5   Catering                             27240 non-null  object
6   Platform_location                    29991 non-null  object
7   Onboardwifi_service                  29991 non-null  object
8   Onboard_entertainment                29994 non-null  object
9   Online_support                       29982 non-null  object
10  Onlinebooking_Ease                   29988 non-null  object
11  Onboard_service                       27621 non-null  object
12  Leg_room                             29975 non-null  object
13  Baggage_handling                     29955 non-null  object
14  Checkin_service                      29978 non-null  object
15  Cleanliness                          29998 non-null  object
16  Online_boarding                      29998 non-null  object
17  Gender                               29977 non-null  object
18  CustomerType                         27139 non-null  object
19  Age                                  29990 non-null  float64
20  TypeTravel                           27064 non-null  object
21  Travel_Class                         30000 non-null  object
22  Travel_Distance                      30000 non-null  int64
23  DepartureDelay_in_Mins                29984 non-null  float64
24  ArrivalDelay_in_Mins                  29897 non-null  float64
dtypes: float64(3), int64(3), object(19)
memory usage: 5.7+ MB

```

Figure 7: Datatype of all columns (before preprocessing)

The original datatypes are shown above. As discussed previously, the ID column needs to be dropped as it is irrelevant to the mode. The table (see figure 6) outlines the required conversions in red. It also shows all possible values for each column, for example 2 possible values for seat_class (Green Car and ordinary).

Column	Possible Values	Current Datatype	Correct datatype
ID	(∞) Numbers	Int64	Int
Overall_Experience	(2) 0, 1	Int64	Binary
Seat_comfort	(6) ['good', 'acceptable', 'poor', 'excellent', 'need improvement', 'extremely poor']	object	Categorical
Seat_Class	(2) ['Green Car', 'Ordinary']	Object	Binary
Arrival_time_convenient	6) ['good', 'acceptable', 'poor', 'extremely poor', 'need improvement', 'excellent']	Object	Categorical
Catering	6) ['good', 'acceptable', 'poor', 'extremely poor', 'need improvement', 'excellent']	Object	Categorical
Platform_location	(5) ['Convinient', 'manageable', 'need improvement', 'Inconvinient', 'very convinient']	Object	Categorical
Onboardwifi_service	(6) ['excellent', 'good', 'acceptable', 'poor', 'need improvement', 'extremely poor']	Object	Categorical
Onboard_entertainment	(6) ['excellent', 'acceptable', 'good', 'poor', 'need improvement', 'extremely poor']	Object	Categorical
Online_support	(6) ['excellent', 'acceptable', 'poor', 'good', 'need improvement', 'extremely poor']	Object	Categorical
Onlinebooking_Ease	(6) ['excellent', 'acceptable', 'poor', 'need improvement', 'good', 'extremely poor']	Object	Categorical
Onboard_service	(6) ['good', 'need improvement', 'excellent', 'acceptable', 'poor', 'extremely poor']	Object	Categorical
Leg_room	(6) ['good', 'need improvement',	Object	Categorical

	'excellent', 'acceptable', 'poor', 'extremely poor']		
Baggage_handling	(5) ['good', 'excellent', 'need improvement', 'acceptable', 'poor']	Object	Categorical
Checkin_service	(6) ['excellent', 'good', 'acceptable', 'poor', 'need improvement', 'extremely poor']	Object	Categorical
Cleanliness	(6) ['good', 'excellent', 'acceptable', 'need improvement', 'poor', 'extremely poor']	Object	Categorical
Online_boarding	(6) ['excellent', 'good', 'acceptable', 'poor', 'need improvement', 'extremely poor']	Object	Categorical
Gender	(2) ['Female', 'Male']	Object	Binary
CustomerType	(2) ['Loyal Customer', 'disloyal Customer']	Object	Binary
Age	(75 unique values)	Float	Int64
TypeTravel	(2) ['Personal Travel', 'Business travel']	Object	Binary
Travel_Class	(2) ['Eco', 'Business']	Object	Binary
Travel_Distance	(4547 unique values)	Int64	Int64
DepartureDelay_in_Mins	(342 unique values)	Float64	Int64
ArrivalDelay_in_Mins	(334 unique values)	Float64	Int64

Table 1: Data Type insight and correction

2.6. Identifying missing values

```
missing_values_count = df.isnull().sum()

# Calculate the percentage of missing values per column
total_rows = len(df)
missing_values_percentage = (missing_values_count / total_rows) * 100

# Format the percentage of missing values
missing_values_percentage_formatted = missing_values_percentage.apply(lambda x: "{:.2f}%".format(x))

# Display the count and percentage of missing values for each column
missing_values_info = pd.DataFrame({
    'Count': missing_values_count,
    'Percentage': missing_values_percentage_formatted
})

print(missing_values_info)

# Print total missing values
total_missing_values = missing_values_count.sum()
print(f"\nTotal Missing values: {total_missing_values}")

# Calculate and print the overall percentage of missing values in the DataFrame
total_values = total_rows * len(df.columns)
overall_missing_percentage = (total_missing_values / total_values) * 100
print(f"Overall Percentage of Missing Values: {overall_missing_percentage:.2f}%")
```

[27] ✓ 0.0s

```
...
      Count Percentage
ID          0      0.00%
Overall_Experience  0      0.00%
Seat_comfort      15      0.05%
Seat_Class        0      0.00%
Arrival_time_convenient  2802    9.34%
Catering          2760    9.20%
Platform_location    9      0.03%
Onboardwifi_service   9      0.03%
Onboard_entertainment  6      0.02%
Online_support       18      0.06%
Onlinebooking_Ease    12      0.04%
Onboard_service      2379    7.93%
Leg_room            25      0.08%
Baggage_handling     45      0.15%
Checkin_service      22      0.07%
Cleanliness          2      0.01%
Online_boarding       2      0.01%
Gender              23      0.08%
CustomerType        2861    9.54%
Age                 10      0.03%
TypeTravel          2936    9.79%
Travel_Class         0      0.00%
Travel_Distance      0      0.00%
DepartureDelay_in_Mins  16      0.05%
ArrivalDelay_in_Mins  103      0.34%

Total Missing values: 14055
Overall Percentage of Missing Values: 1.87%
```

Figure 8: Identifying count and percentage of missing values in each column

The code and output (see figure 7) outputs missing values in all columns. It also outputs the percentage of missing values relative to all values. This has been done as imputing any

columns where there is a percentage of missing values is high (i.e > 20%) would lead to inaccurate imputation when utilising the mode. The entire dataset has 1.87% of its values missing. No columns exceed a 10% or higher proportion of missing values. Imputing all can be imputed without significant loss of data integrity.

2.7. Checking for duplicate entries

```
print(df.duplicated().sum())
```

[28] ✓ 0.0s

... 0

Figure 9: Checking for duplicate values

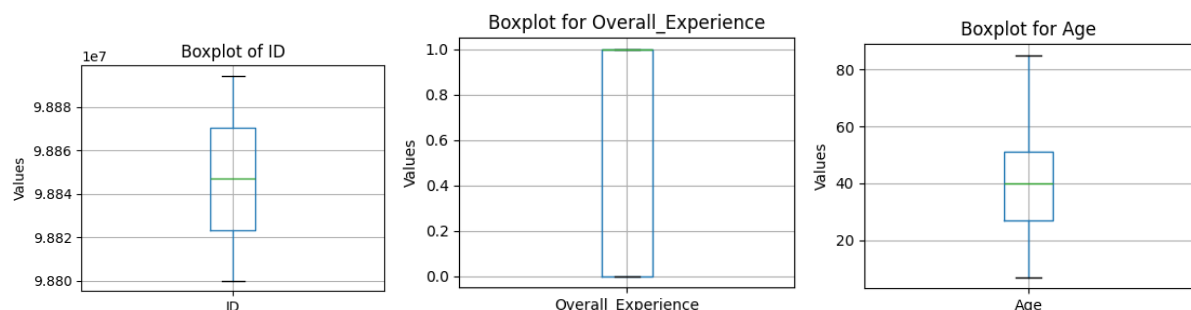
2.8. Identifying Outliers

2.8.1. Numerical outliers (Box Plot)

```
# Filter to include only numeric data for boxplots
numeric_cols = df.select_dtypes(include=['number']).columns

# Create a figure for each numeric column
for col in numeric_cols:
    plt.figure(figsize=(4, 3))
    df.boxplot(column=col)
    plt.title(f'Boxplot for {col}')
    plt.ylabel('Values')
    plt.show()
```

[40] ✓ 0.5s



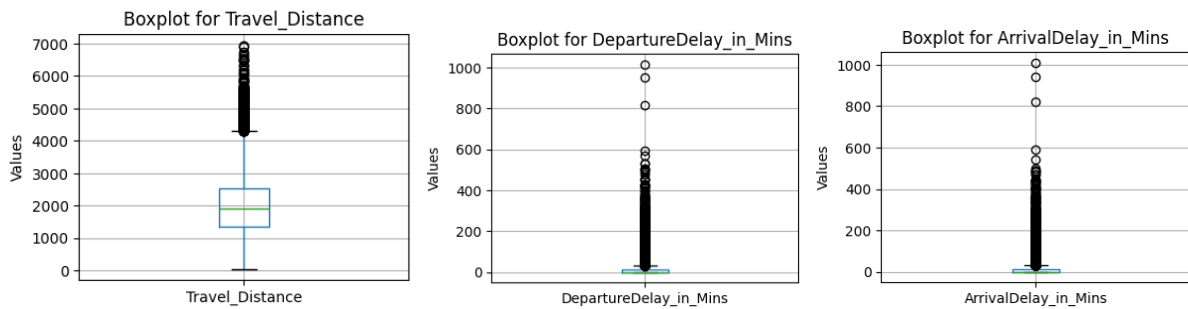


Figure 10: Identifying potential numerical outliers

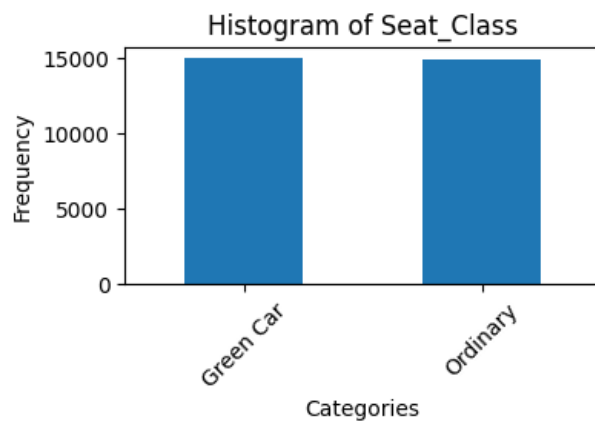
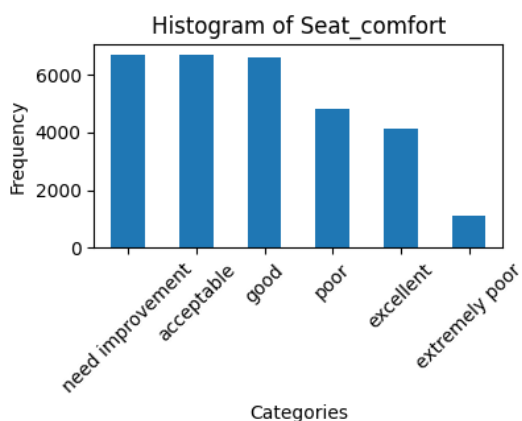
The code and output (see figure 9) show boxplots for all numerical columns (disregard ID as it will be dropped). Though technically outliers, all outliers shown above are to be expected, as delays are usually abnormal. It would therefore not make sense to drop any values regarding delays. It could though be argued that very extreme outliers (i.e. 1000 minute departure delay) should be dropped.

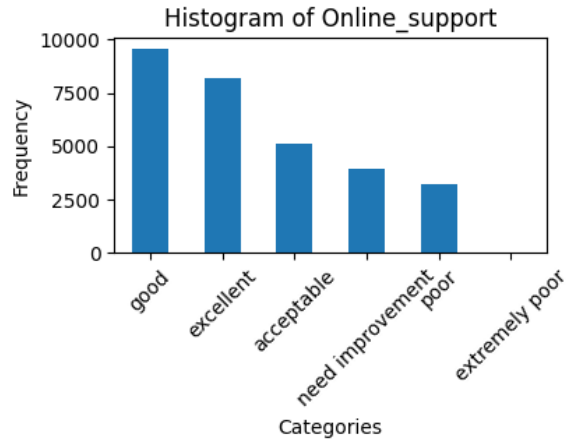
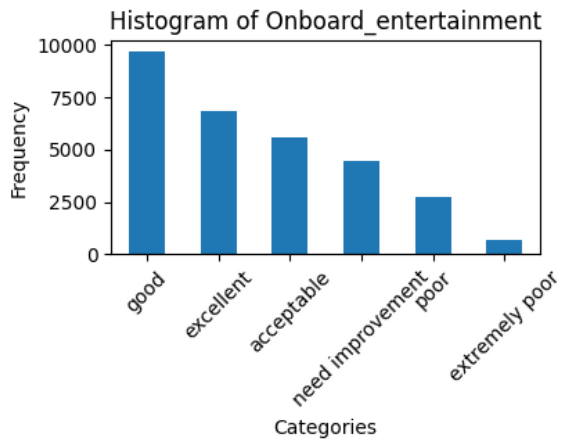
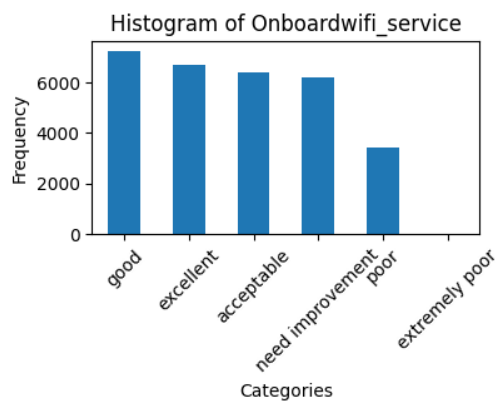
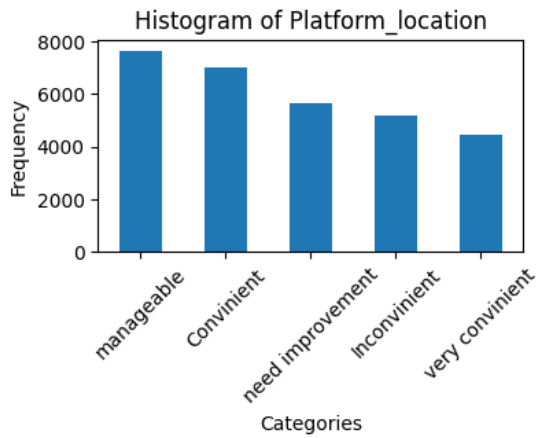
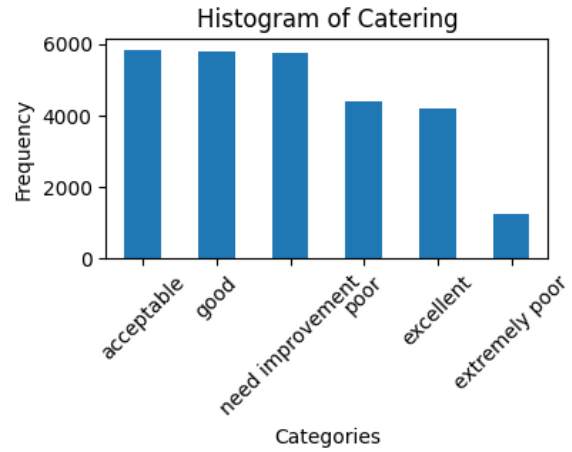
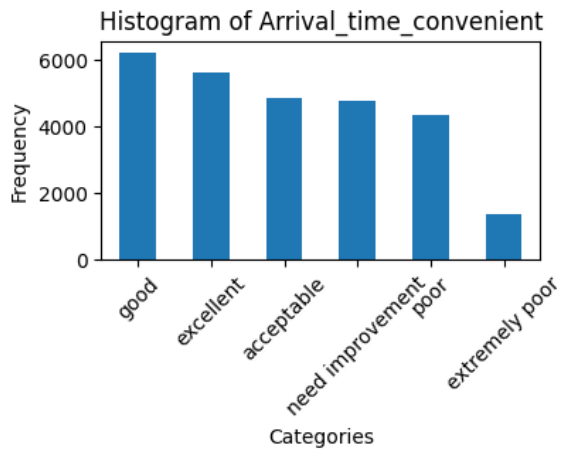
2.8.2. Categorical outliers (Histograms)

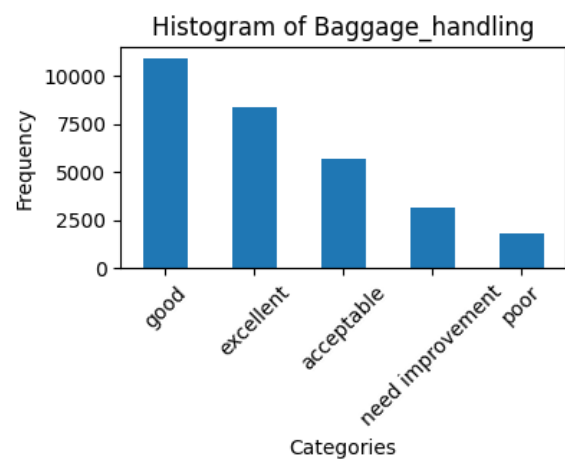
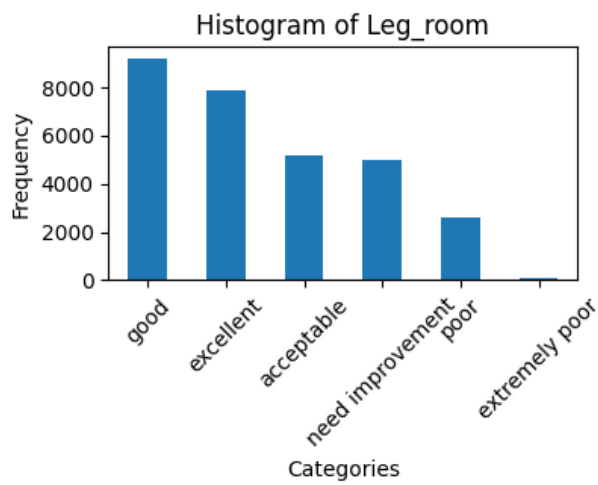
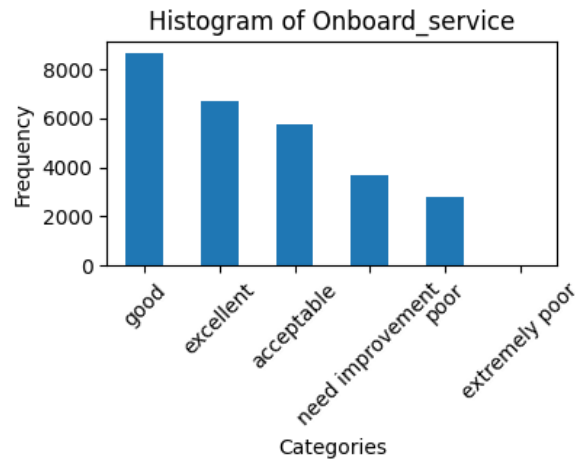
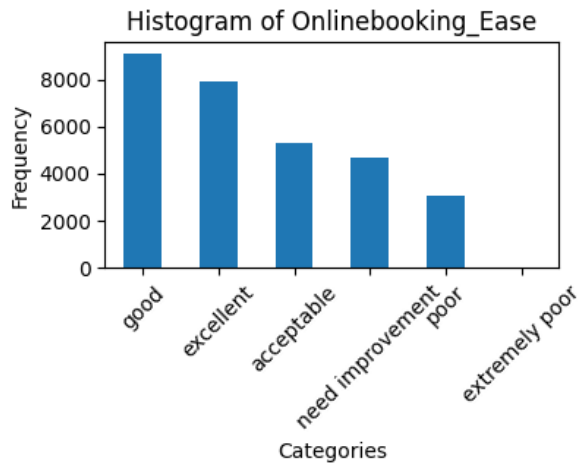
```
# Filter to include only categorical data for histograms
categorical_cols = df.select_dtypes(include=['object']).columns

# Create a histogram for each categorical column
for col in categorical_cols:
    plt.figure(figsize=(4, 2))
    df[col].value_counts().plot(kind='bar')
    plt.title(f'Histogram of {col}')
    plt.xlabel('Categories')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.show()
```

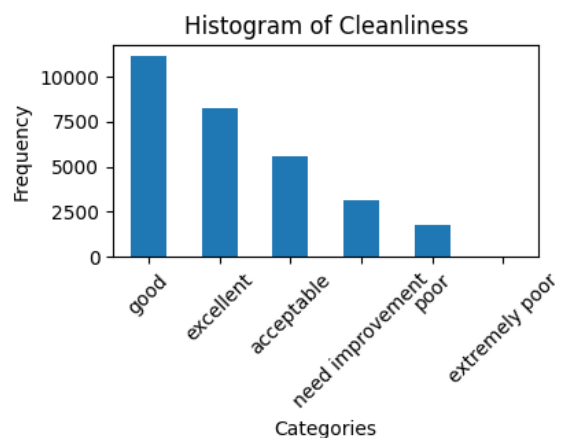
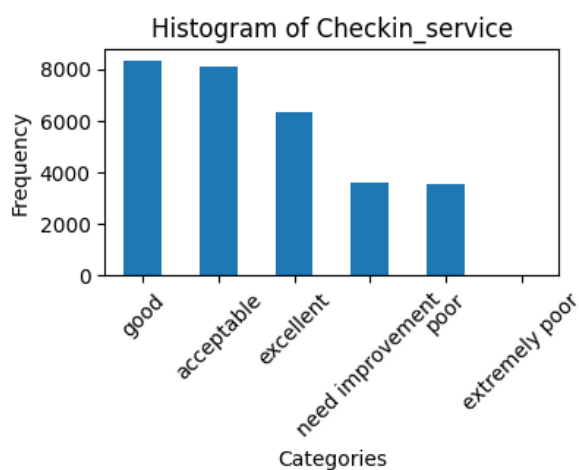
[43] ✓ 1.6s







[View next page ->](#)



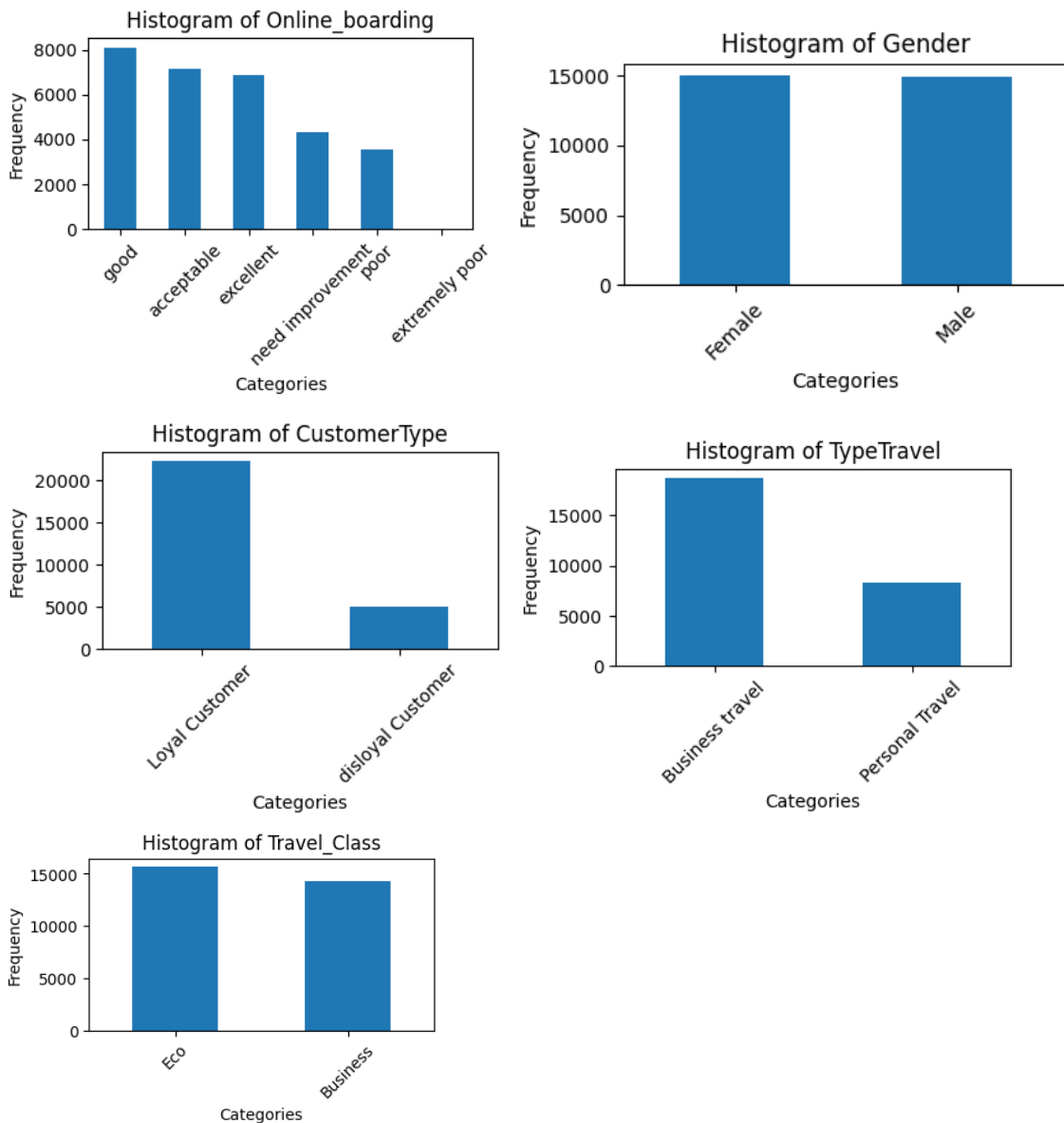


Figure 11: Identifying potential categorical outliers + further insight to data distribution [View next page ->](#)

Histogram Notes:

- Service Quality Feedback:** The majority of responses cluster in the middle categories. Only a few responses in extreme categories.
- Customer Demographics and Preferences:**
 - Gender: The distribution between 'Female' and 'Male' categories is relatively even.
 - Customer Type: There is a clear distinction with a larger number of 'Loyal Customer' responses compared to 'Disloyal Customer'.

- Type of Travel: 'Personal Travel' seems to have more responses than 'Business Travel'.
- Class of Travel: The 'Eco' class has significantly more responses compared to 'Business', indicating a higher frequency of customers travelling in the economy class.

3. Potential Areas of Concern:

- Certain service attributes like 'Seat Comfort', 'Platform Location', and 'Catering' show a notable number of 'poor' or 'extremely poor' ratings, which could be areas requiring attention and improvement.
- Other attributes like 'Onboard Wi-Fi Service' and 'Online Boarding' demonstrate a good number of 'excellent' ratings, suggesting higher customer satisfaction in these areas.

2.9. Class imbalances

```
def identify_imbalanced_classes_for_all(df, threshold_percentage=10.0):

    # Identify categorical columns
    categorical_columns = df.select_dtypes(include=['object', 'category']).columns
    RED = '\033[91m'
    RESET = '\033[0m'
    for column in categorical_columns:
        # Calculate the frequency of each class
        class_counts = df[column].value_counts(normalize=True) * 100

        # Identify classes where the percentage is below the threshold
        imbalanced_classes = class_counts[class_counts < threshold_percentage]

        # Output the imbalanced classes
        if not imbalanced_classes.empty:
            print(f"Imbalanced classes in {RED}{column}{RESET} that are below {threshold_percentage}%:")
            # Formatting the output to exclude 'Name' and 'dtype'
            for class_name, proportion in imbalanced_classes.items():
                print(f"{class_name}: {proportion:.2f}%")
            print()

    identify_imbalanced_classes_for_all(df, 10) # Adjust the threshold as necessary
```

[131] ✓ 0.0s

Imbalanced classes in 'Seat_comfort' below 10%: extremely poor: 3.67%	Imbalanced classes in 'Onboard_service' below 10%: extremely poor: 0.01%
Imbalanced classes in 'Arrival_time_convenient' below 10%: extremely poor: 4.98%	Imbalanced classes in 'Leg_room' below 10%: poor: 8.75% extremely poor: 0.31%
Imbalanced classes in 'Catering' below 10%: extremely poor: 4.59%	Imbalanced classes in 'Baggage_handling' below 10%: poor: 6.00%
Imbalanced classes in 'Onboardwifi_service' below 10%: extremely poor: 0.07%	Imbalanced classes in 'Checkin_service' below 10%: extremely poor: 0.00%
Imbalanced classes in 'Onboard_entertainment' below 10%: poor: 9.12% extremely poor: 2.33%	Imbalanced classes in 'Cleanliness' below 10%: poor: 6.03% extremely poor: 0.01%
Imbalanced classes in 'Online_support' below 10%: extremely poor: 0.00%	
Imbalanced classes in 'Onlinebooking_Ease' below 10%: extremely poor: 0.02%	Imbalanced classes in 'Online_boarding' below 10%: extremely poor: 0.01%

Figure 12: Class imbalances

The code and output (see figure x) outline imbalanced classes within each categorical column of a dataset. it outputs any categories that constitute 10% or less of the total entries in a column and labels them as imbalanced. It was initially considered (and tested on modelling) to drop categories that represented less than 10% (proportional) of a given column. However, after further consideration, it was realised that it leads to a significant loss of information, therefore under sampling. Combining two imbalanced columns was also considered (i.e. combining poor and extremely poor). Though somewhat better, it would lead to a loss of granular information. Furthermore, it would be ethically unfair to disregard extremely poor reviews and replace/combine them with 'poor' reviews. A better solution is to leave values as is and to introduce SMOTE (generate synthetic data for imbalanced classes) within each of the developed models. It is worth noting that **smote should only be applied to the training data AND NOT THE TEST DATA** (discussed further in section 5).

2.10. Exploratory analysis (unsupervised)

2.10.1 Correlation matrix (Numerical Data)

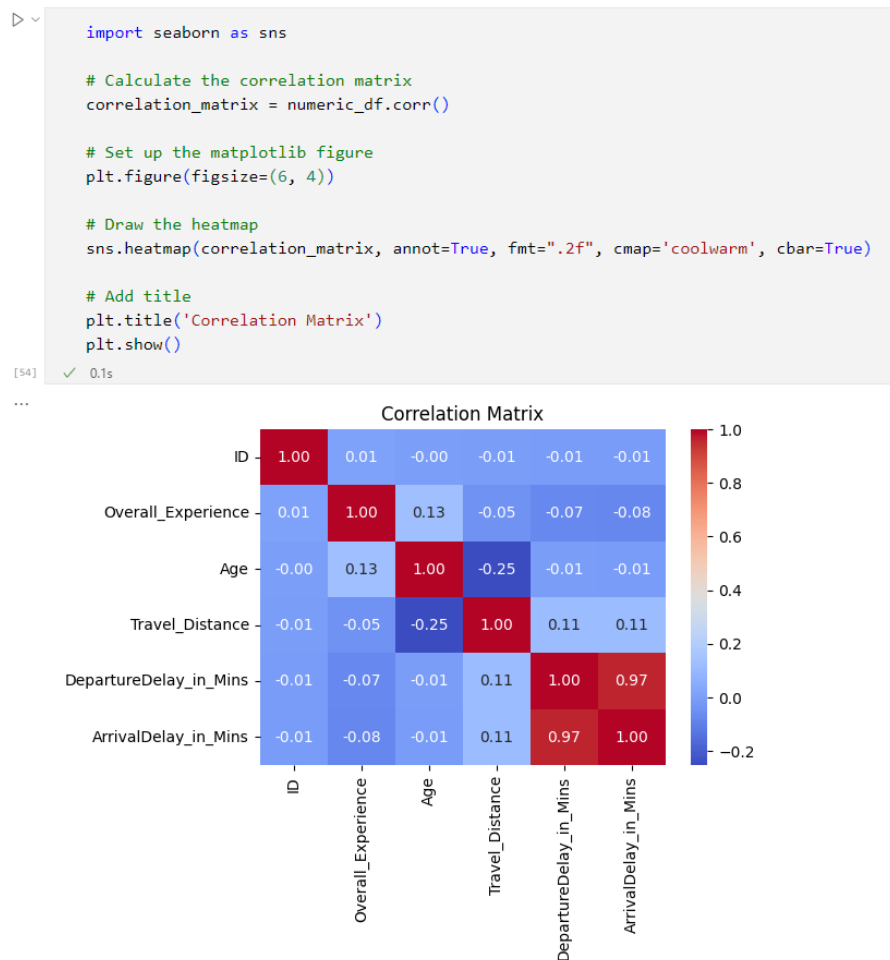


Figure 13: Correlation matrix for numerical data

The numerical correlation matrix (see figure 11) shows a slight negative correlation (**-0.25** coefficient) between the distance travelled and the age, suggesting that the older the person is, the shorter their travel distance. It is worth noting that this is a relatively weak correlation, and may not be linear. As expected, there is a strong positive correlation (**0.97**) between the Departure and Arrival delay.

2.10.2. Correlation matrix (Categorical Data)

```
#use of Cramér's V as it provides a straightforward interpretation similar to the Pearson correlation.
# function to compute Cramér's V based on the chi-squared statistic from the contingency table.

def cramers_v(x, y):
    confusion_matrix = pd.crosstab(x, y)
    chi2 = chi2_contingency(confusion_matrix)[0]
    n = confusion_matrix.sum().sum()
    phi2 = chi2 / n
    r, k = confusion_matrix.shape
    phi2_corr = max(0, phi2 - ((k-1)*(r-1))/(n-1))
    r_corr = r - ((r-1)**2)/(n-1)
    k_corr = k - ((k-1)**2)/(n-1)
    return np.sqrt(phi2_corr / min((k_corr-1), (r_corr-1)))

# Apply Cramér's V for each pair of categorical variables
categorical_df = df.select_dtypes(include=['object'])
correlations = pd.DataFrame(index=categorical_df.columns, columns=categorical_df.columns)

for col1 in categorical_df.columns:
    for col2 in categorical_df.columns:
        correlations.loc[col1, col2] = cramers_v(categorical_df[col1], categorical_df[col2])

correlations = correlations.astype(float)

# Plotting the heatmap for Cramér's V correlation matrix
plt.figure(figsize=(12, 8))
sns.heatmap(correlations, annot=True, fmt=".2f", cmap='coolwarm', cbar=True)
plt.title('Cramér's V Correlation Matrix for Categorical Data')
plt.show()
```

[56] ✓ 3.2s

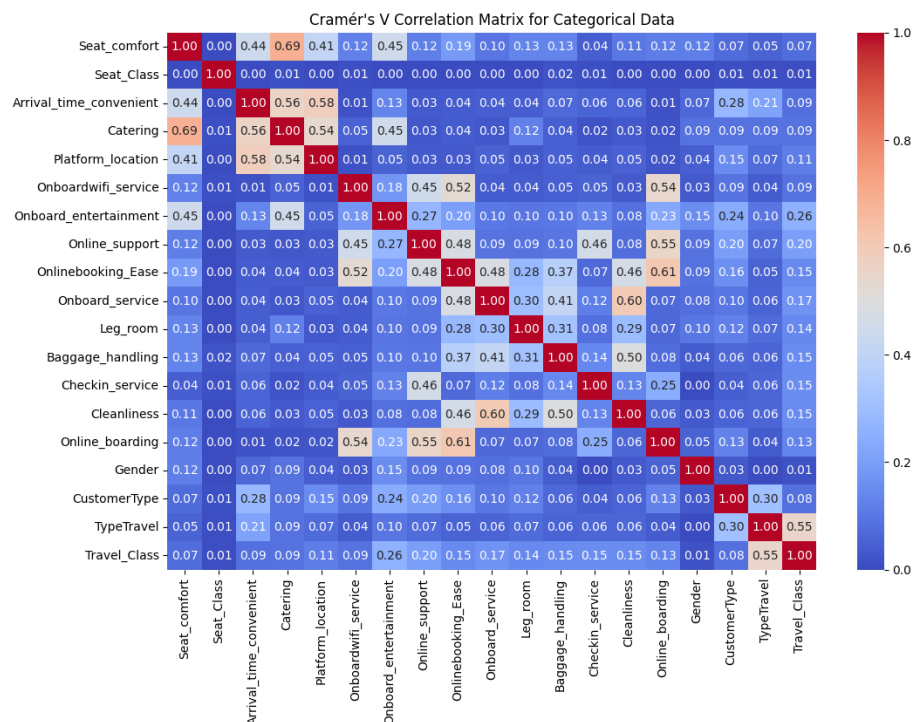


Figure 14: Correlation matrix for categorical data

The categorical correlation matrix created using cramers V suggests the following correlations (threshold of 0.31):

Column One	Column Two	Correlation Coefficient	Interpretation
Seat Comfort	Catering	0.69	Passengers satisfied with seat comfort have high satisfaction with catering services.
Seat Comfort	Online Boarding	0.54	higher seat comfort = higher satisfaction with online boarding
Catering	Platform Location	0.56	happiness catering services correlate with satisfaction with platform location
Catering	Onlinebooking_Ease	0.54	those who are satisfied with the catering services also find the online booking process easier.
Online Support	Onlinebooking_Ease	0.52	those who find the online support satisfactory also tend to find the online booking process easy.
Online Boarding	CustomerType	0.61	strong association between customer type and satisfaction with online boarding, suggesting different experiences or preferences between, for example, frequent customers and occasional travelers.
Online Boarding	& TypeTravel	0.55	moderately strong correlation suggests that the type of travel (perhaps business vs. leisure) is associated with satisfaction with online boarding.

Table 2: Correlation matrix insight

2.10.3. PI charts (Categorical Data)

```
def plot_pie_charts(df, charts_per_row=3):
    categorical_columns = df.select_dtypes(include=['object', 'category']).columns
    num_cols = len(categorical_columns)
    num_rows = -(num_cols // charts_per_row) # Ceiling division to get number of rows needed

    # Adjust subplot spacing
    fig, axes = plt.subplots(num_rows, charts_per_row, figsize=(charts_per_row * 4, num_rows * 4), gridspec_kw={'hspace': 0.3, 'wspace': 0.3})

    # If there's only one row, axes is a 1D array
    if num_cols <= charts_per_row:
        axes = [axes]

    for i, col in enumerate(categorical_columns):
        ax = axes[i // charts_per_row, i % charts_per_row]
        category_counts = df[col].value_counts()
        ax.pie(category_counts, labels=category_counts.index, autopct='%1.1f%%', startangle=90)
        ax.set_title(col)

    # If there are empty subplots, hide them
    for j in range(num_cols, num_rows * charts_per_row):
        if num_rows > 1:
            axes[j // charts_per_row, j % charts_per_row].axis('off')
        else:
            axes[j].axis('off')

    plt.tight_layout()
    plt.show()

plot_pie_charts(df, charts_per_row=3)
```

[View next page ->](#)

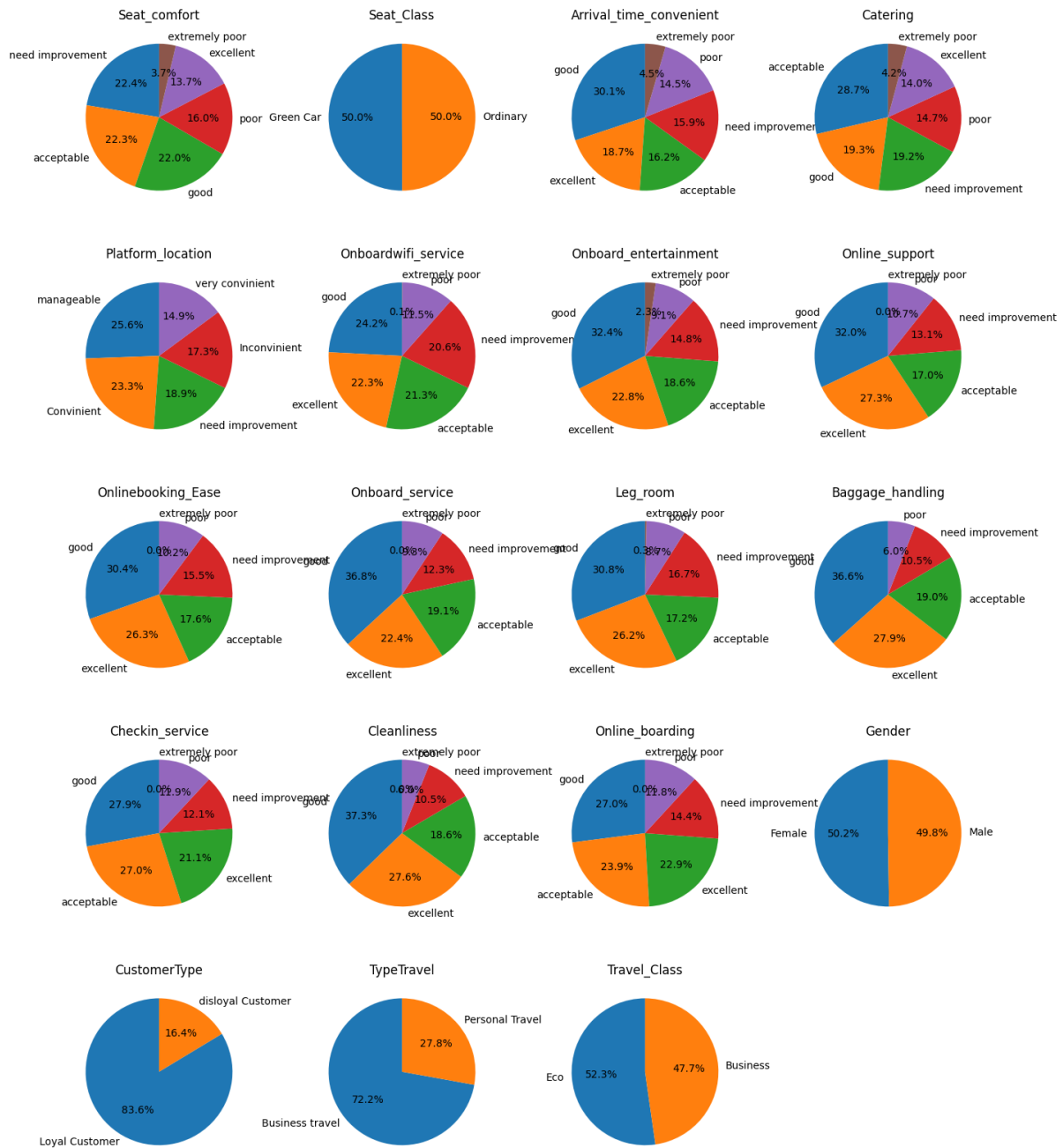


Figure 15: PI charts

Inference:

- Customer feedback varies widely, with notable portions indicating the need for improvement in seat comfort and online support.
- High satisfaction levels are seen in onboard entertainment and Wi-Fi services, with over 22% rating these as excellent.
- Baggage handling and online support are identified as critical areas needing improvement.
- The majority of respondents are loyal customers, predominantly traveling for business, with a near even gender split.

3. Data Pre-Processing

3.1 Dropping irrelevant column (ID)

```
[32] df.drop('ID', axis=1, inplace=True)
```

Figure 16: Dropping irrelevant column

The 'ID' column is dropped from the dataset before training models because it contains unique identifiers for each row of data which are useful for searching and indexing, but do not (generally) carry any meaningful information for supervised prediction. They have no predictive power, and overfitting. The model may also (wrongfully) use/infer ordered ID's as time-related patterns which should not be done/is not intended.

3.2 Imputing missing values

```
[10] imputer = SimpleImputer(strategy='most_frequent') # Initialise imputer
df_imputed = pd.DataFrame(imputer.fit_transform(df), columns=df.columns) # Perform mode imputation
nulls_after_imputation = df_imputed.isnull().sum() # Check if any nulls remain
df = df_imputed #override df
print(nulls_after_imputation) # check if the imputation was successful

... Overall_Experience      0
Seat_comfort              0
Seat_Class                0
Arrival_time_convenient  0
Catering                  0
Platform_location         0
Onboardwifi_service       0
Onboard_entertainment     0
Online_support            0
Onlinebooking_Ease        0
Onboard_service           0
Leg_room                  0
Baggage_handling          0
Checkin_service           0
Cleanliness               0
Online_boarding           0
Gender                    0
CustomerType              0
Age                       0
TypeTravel                0
Travel_Class              0
Travel_Distance           0
DepartureDelay_in_Mins    0
ArrivalDelay_in_Mins     0
dtype: int64
```

Figure 17: Imputing missing values using mode

The code above replaces/imputes missing values identified in figure 7. The chosen strategy for imputation is 'most_frequent', which means that all missing values will be filled using the mode of the column. One reason for this choice is because it does not make sense to impute categorical columns using other attributes such as the mean or median. A further justification is that it preserves the original distribution of that column. For example, through having a lower impact on variance, particularly if the missing values are a small portion of the data, as in this case (see figure 7). Another advantage is that while replacing the missing values using the mode, the mode for that column remains the same.

3.3 Converting to correct datatype

```
▷ ~
• # List of columns to convert to native Python int type
  int_columns = ['Travel_Distance', 'Age', 'DepartureDelay_in_Mins', 'ArrivalDelay_in_Mins']

  # Convert binary columns including Overall_Experience to categorical with 0 and 1 values
  binary_columns_with_mapping = {
      'Overall_Experience': {0: 0, 1: 1}, # Explicitly adding Overall_Experience here
      'Seat_Class': {'Green Car': 1, 'Ordinary': 0},
      'Gender': {'Female': 0, 'Male': 1},
      'TypeTravel': {'Personal Travel': 0, 'Business travel': 1},
      'Travel_Class': {'Eco': 0, 'Business': 1},
      'CustomerType': {'Loyal Customer': 0, 'disloyal Customer': 1}
  }

  # Convert object columns to categorical
  categorical_columns = [
      'Seat_comfort', 'Arrival_time_convenient', 'Catering', 'Platform_location',
      'Onboardwifi_service', 'Onboard_entertainment', 'Online_support',
      'Onlinebooking_Ease', 'Onboard_service', 'Leg_room', 'Baggage_handling',
      'Checkin_service', 'Cleanliness', 'Online_boarding', 'CustomerType'
  ]

  # Convert int columns to integer and round if necessary
  for col in int_columns:
      df[col] = df[col].round().astype(int)

  # Convert binary columns to categorical using their mapping
  for col, mapping in binary_columns_with_mapping.items():
      df[col] = df[col].map(mapping).astype('bool')

  # Convert categorical columns
  for col in categorical_columns:
      df[col] = df[col].astype('category')
```



```
df.info()

[24] ✓ 0.0s

... <class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 24 columns):
#   Column                                Non-Null Count  Dtype
---  ---                                -
0   Overall_Experience                    30000 non-null  bool
1   Seat_comfort                         30000 non-null  category
2   Seat_Class                           30000 non-null  bool
3   Arrival_time_convenient              30000 non-null  category
4   Catering                             30000 non-null  category
5   Platform_location                    30000 non-null  category
6   Onboardwifi_service                  30000 non-null  category
7   Onboard_entertainment                 30000 non-null  category
8   Online_support                       30000 non-null  category
9   Onlinebooking_Ease                   30000 non-null  category
10  Onboard_service                       30000 non-null  category
11  Leg_room                             30000 non-null  category
12  Baggage_handling                     30000 non-null  category
13  Checkin_service                      30000 non-null  category
14  Cleanliness                          30000 non-null  category
15  Online_boarding                      30000 non-null  category
16  Gender                               30000 non-null  bool
17  CustomerType                         30000 non-null  category
18  Age                                  30000 non-null  int32
19  TypeTravel                           30000 non-null  bool
20  Travel_Class                         30000 non-null  bool
21  Travel_Distance                      30000 non-null  int32
22  DepartureDelay_in_Mins                30000 non-null  int32
23  ArrivalDelay_in_Mins                  30000 non-null  int32
dtypes: bool(5), category(15), int32(4)
memory usage: 1.0 MB
```

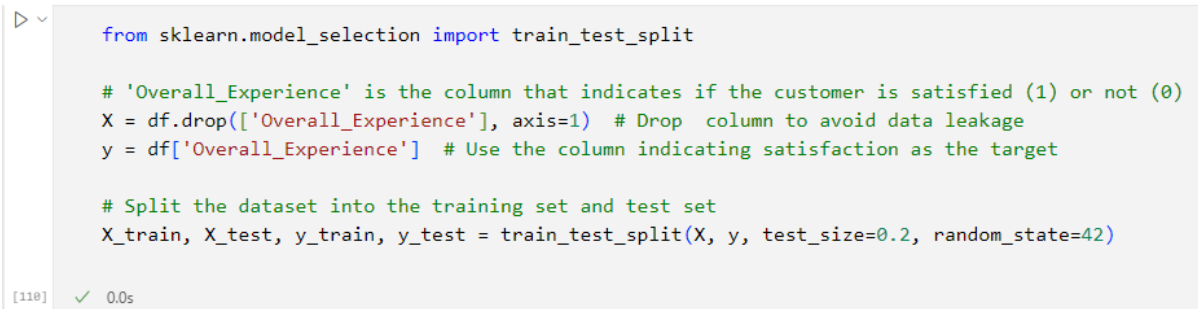
Figure 18: Converting datatypes

The initial datatypes of categorical columns were imported as objects (a representation of its string). They have been converted to categorical for the **following reasons**:

Encoding Semantics: Object types represent strings (in this case), which are arbitrary to mathematical algorithms. By converting to categorical, it allows for encoding (i.e one-hot, labelling, etc.).

Prevention of Data Leakage: By categorising variables in preprocessing, it prevents data leakage, where the model inadvertently gains information from the test data during training. This is not the case with categorical since them and their encoding are strictly controlled.

3.4 Train/test split

A screenshot of a Jupyter Notebook code cell. The code imports `train_test_split` from `sklearn.model_selection`. It then drops the 'Overall_Experience' column from the dataframe `df` to create `X`, and uses that column as the target variable `y`. Finally, it splits the data into training and testing sets using `train_test_split` with `test_size=0.2` and `random_state=42`. The output shows the code executed successfully in 0.0 seconds.

```
from sklearn.model_selection import train_test_split

# 'Overall_Experience' is the column that indicates if the customer is satisfied (1) or not (0)
X = df.drop(['Overall_Experience'], axis=1) # Drop column to avoid data leakage
y = df['Overall_Experience'] # Use the column indicating satisfaction as the target

# Split the dataset into the training set and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

[110] ✓ 0.0s

Figure 19: Splitting train/test data

The dataset is divided into two parts, one for training and the other for testing its performance. The split is 20% for testing and 80% for training, and is defined by the; `test_size=0.2`; line. The random state is set to 42 to ensure that the split is reproducible, for example the same random set ensures that the split is the same each time the code is run. This will aid in finetuning and debugging later. The data has been split in order to maximise generalisation through introducing unseen data. This will also help to prevent overfitting (where a model memorises rather than learns). It is also worth noting that the data was split before other steps i.e. one hot encoding and normalisation as it helps to prevent data leakage and an unbiased model evaluation.

3.4 Encoding categorical values

```

from sklearn.preprocessing import OrdinalEncoder
import pandas as pd

custom_order = {
    # Define custom order for ordinal columns
    'Seat_comfort': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Arrival_time_convenient': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Catering': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Platform_location': ['Inconvenient', 'manageable', 'need improvement', 'Convenient', 'very convenient'],
    'Onboardwifi_service': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Onboard_entertainment': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Online_support': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Onlinebooking_Ease': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Onboard_service': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Leg_room': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Baggage_handling': ['poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Checkin_service': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Cleanliness': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Online_boarding': ['extremely poor', 'poor', 'need improvement', 'acceptable', 'good', 'excellent'],
    'Seat_Class': ['Ordinary', 'Green Car'], # Custom order for the "Class" column
    'CustomerType': ['disloyal Customer', 'Loyal Customer'], # Custom order for the "CustomerType" column
    'TypeTravel': ['Business travel', 'Personal Travel'], # Custom order for the "TypeTravel" column
    'Travel_Class': ['Eco', 'Business'] # Custom order for the "Travel_Class" column
}

# Initialize OrdinalEncoder with custom order
ordinal_encoder = OrdinalEncoder(categories=[custom_order[col] for col in custom_order.keys()])
ordinal_columns = list(custom_order.keys()) # Define ordinal columns

X_train[ordinal_columns] = ordinal_encoder.fit_transform(X_train[ordinal_columns])
X_test[ordinal_columns] = ordinal_encoder.transform(X_test[ordinal_columns]) # Encode ordinal columns

non_ordinal_columns = ['Gender']
X_train = pd.get_dummies(X_train, columns=non_ordinal_columns, drop_first=True) # One-hot encode non-ordinal columns
X_test = pd.get_dummies(X_test, columns=non_ordinal_columns, drop_first=True)

```

```

X_train.head()

```

Onlinebooking_Ease	Onboard_service	...	Cleanliness	Online_boarding	CustomerType	Age	TypeTravel	Travel_Class	Travel_Distance	DepartureDelay_in_Mins	ArrivalDelay_in_Mins
4.0	5.0	...	2.0	4.0	1.0	22	0.0	1.0	3702	7	38
4.0	4.0	...	4.0	2.0	1.0	27	1.0	0.0	2394	6	10
5.0	2.0	...	2.0	5.0	0.0	47	0.0	0.0	2126	0	0
5.0	5.0	...	5.0	3.0	1.0	42	0.0	1.0	3408	8	0
5.0	1.0	...	2.0	5.0	1.0	20	1.0	0.0	2902	0	0

Figure 20: Initial (Label Encoding)

Initially, all ordinal categorical columns were **label encoded** (order manually defined in code). However, upon further testing on the development of the five models. Though the columns were ordinal, **One-Hot encoding provided with significantly more accuracy** (from 80% with label encoding to 90% with One-Hot encoding (see below).

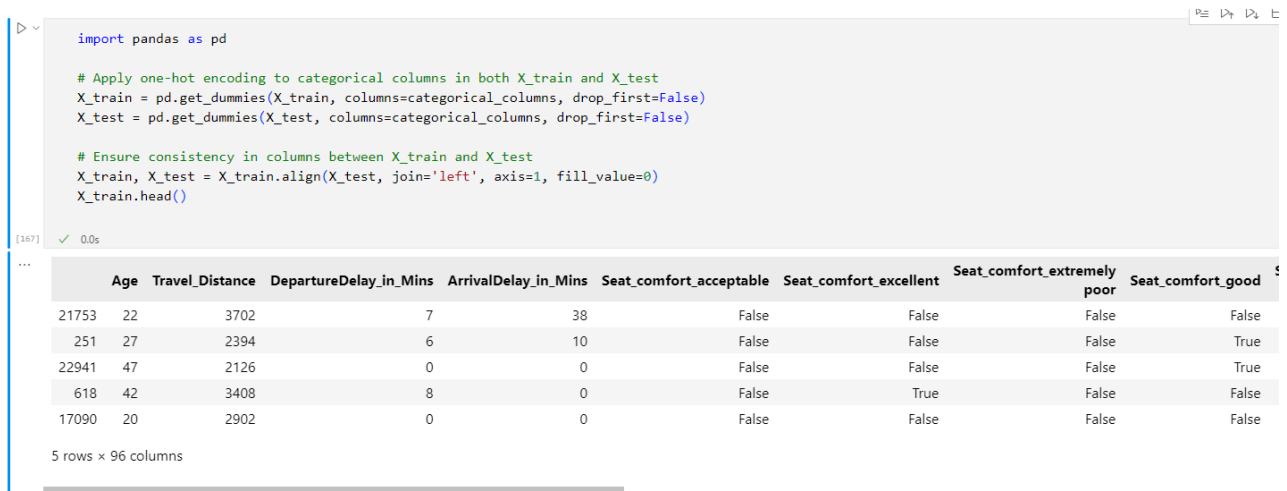


Figure 21 Final Decision (One-Hot encoding)

One hot encoding was then (the final choice) applied to the dataset. It is when categorical variables are converted in to binary for each possible value (through adding more columns). One reason this was done is that it eliminated misleading ordinality, for example, if the data was inserted in a specific, the ML model may interpret one category as being more significant than another. One-hot encoding also aids in improving the models performance, particularly in linear models, as it captures the individual impact of each category and disregards ordinal relationships.

3.6 Data standardisation



Figure 22: Data Scaling

Initially min-max scaling was applied to the numerical columns, however after further consideration testing, since not all numerical columns have the same range (max-min), it would lead to decreased performance and would make more sense to **apply standardisation (StandardScaler)**, this was further confirmed through testing (around a 3% improvement in accuracy, F1, etc. This is because it will help to preserve relationships, and would be less sensitive to outliers. As stated previously, preprocessing steps are conducted **post** splitting the train/test set.

4. Algorithms

4.1 Discriminant Analysis

Discriminant Analysis is a statistical technique used to distinguish between two or more naturally occurring groups (Fisher, 1936). Fisher's linear discriminant function is particularly significant in the field and can be expressed as $y(x) = w^T x + w_0$, where w is the weight vector that maximizes the ratio of between-class to within-class variance (McLachlan, 1992). The weight vectors can be computed from the eigenvectors of $S_w^{-1} S_b$ (Johnson and Wichern, 2007), where S_w and S_b are the within-group scatter matrix and the between-group scatter matrix, respectively. Discriminant Analysis is a fancy way of figuring out what characteristics can tell us which group something or someone belongs to.

4.2 Linear SVM

(James et al., 2013) explained the Linear SVM as a supervised learning algorithm which is commonly used for binary classification. It tried to find the best separating hyperplane that maximizes the margin between the 2 classes. The aim is to find a decision boundary (that best separates the classes. The decision boundary is defined by the following equation:

$$f(x) = w^T x + b = 0$$

Where:

w : is the weight vector. This can be thought of as a list of numbers that represent how important each feature is for separating the points. If the weight is large, that feature is very important for the separation.

w^T : "T" is transpose which flips the rows and columns of the weight vector.

x : is the feature vector which is the list of the characteristics or attributes of the you're trying to categorise

b : is the biased term which is the number that allows movement of the hyperplane back and forth to get the best position for separating the points.

The " $= 0$ " part defines the hyperplane itself—it's the set of points that are exactly halfway between the two categories

SVM aims to place this edge as perfectly as possible between the two groups of points.

4.3 Logistic Regression

Logistic regression models the probability that an outcome belongs to a particular category. (Hosmer, Lemeshow, and Sturdivant, 2013) explains that the probability estimation is framed using the logistic function, which is an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits: Unlike linear regression, which predicts continuous outcomes, logistic regression predicts probabilities that determine into which of two possible categories a case falls.

The formula for logistic regression can be described as the following (Agresti, 2002):

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n)}}$$

Where:

$p(x)$: is the probability that the outcome is a '1' for a given value of x

$\beta_0, \beta_1, \beta_2, \dots, \beta_k$: are coefficients that reflect the influence of predictor variables x_1, x_2, \dots, x_k

e : is the base for the natural logarithm.

The logistic function is particularly useful because it can take an input with any value from negative infinity to positive infinity, whereas the output is confined between 0 and 1.

4.4 RBF SVM

An RBF (Radial Basis Function) SVM (Support Vector Machine) is a type of SVM classifier that uses the RBF kernel to perform nonlinear classification.

Kernel Function

The RBF kernel is defined as:

$$K(x, x') = \exp(-\gamma \|x - x'\|^2)$$

where:

- x, x' are feature vectors in the input space.
- γ (gamma) is a parameter that sets the "spread" of the kernel. A larger gamma value makes the decision boundary more sensitive to individual data points, leading to a more complex model, whereas a smaller gamma extends the influence of each data point to a larger region.

4.5 Gradient-Boosted Trees

Gradient Boosting involves sequentially adding predictions from individual trees (or simple models) to correct errors made by the previous trees. The models are added one at a time, and existing models in the ensemble are not changed. This method combines multiple weak or "shallow" trees, where each tree corrects its predecessor, effectively improving the model's predictive accuracy at each step.

The ensemble model after training for N iterations (or trees) can be expressed as:

$$F(x) = \sum_{n=1}^N \gamma_n h_n(x)$$

Where:

- $F(x)$ is the final model's output for the input x .
- $h_n(x)$ represents the output of the n -th tree.
- γ_n is the weight of the n -th tree, which scales its contribution to the final model.

5.Implementation / EXPERIMENTATION/ Model development (AND SOME RESULTS)

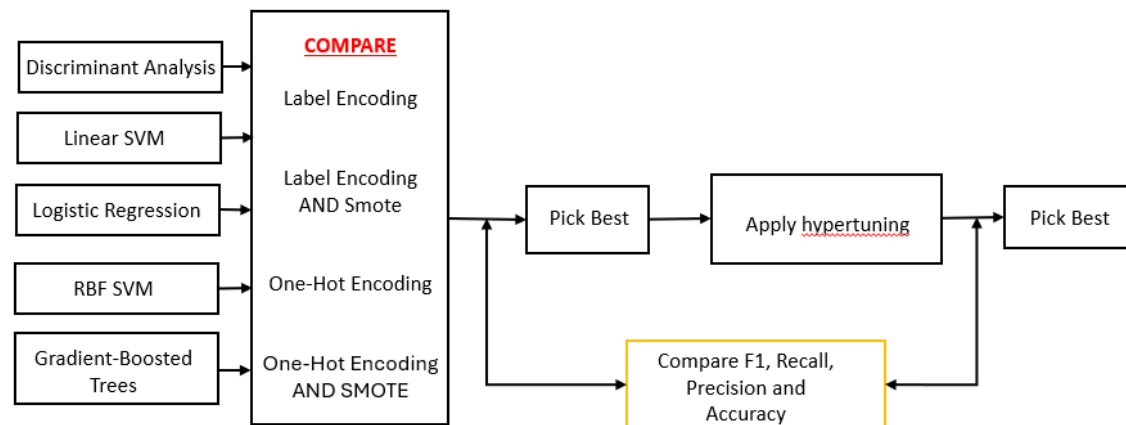


Figure 23: Modelling plan

The plan for experimentation is to train the model multiple times, with combinations of Label Encoding, One-Hot and SMOTE, take the best model, then apply hype tuning to the best one. Worst and best are shown here, **rest in appendix.**

5.1 Discriminant Analysis

5.1.1 Best Model

One-Hot + LSQR + SMOTE

```
> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
  from imblearn.over_sampling import SMOTE

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply SMOTE only on training data to handle class imbalance
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize LDA using the LSQR solver
lda = LinearDiscriminantAnalysis(solver='lsqr')

# Train the LDA model on the resampled training data
lda_model = lda.fit(X_train_smote, y_train_smote)

evaluate_model(lda_model, X_train_smote, y_train_smote, X_test, y_test)
```

[219] ✓ 1.0s

... Accuracy: 0.89
Precision: 0.90
Recall: 0.90
F1 Score: 0.90

Figure 24: Best LDA Model

The best results were from using One-Hot Encoding and SMOTE since they helped the model understand different categories and balanced the data, and LSQR was good at solving the model quickly and accurately.

5.1.2 Worst Model

Label Encoding

```

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE

## Initialize SMOTE
smote = SMOTE(random_state=42)

## Apply SMOTE only on training data
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the Logistic Regression model
lda = LinearDiscriminantAnalysis()

# Train the LDA
log_reg_model = lda.fit(X_train, y_train)

# Evaluate the model
evaluate_model(log_reg_model, X_train, y_train, X_test, y_test)

```

[156] ✓ 0.7s

... Accuracy: 0.84
Precision: 0.85
Recall: 0.86
F1 Score: 0.85

Figure 25: Worst LDA Model

The worst model created during the experimentation was LDA with Label Encoding and no SMOTE. The LDA model's poorer performance with Label Encoding compared to One-Hot Encoding suggests that the latter provided a more discriminative feature representation, which is particularly crucial for linear models like LDA to differentiate between classes effectively.

5.1.3. Experiment Outcome (Discriminant Analysis)

LDA	Label Encoding	Label Encoding + Smote	One-Hot Encoding	One-Hot Encoding + SMOTE
Accuracy	0.84	0.84	0.89	0.89
Precision	0.85	0.86	0.90	0.90
Recall	0.86	0.84	0.90	0.90
F1 Score	0.85	0.85	0.90	0.90

LDA (Hyper Tuning)	One-Hot + SVD + SMOTE	One-Hot + lqsr + SMOTE
Accuracy	0.89	0.89
Precision	0.90	0.90
Recall	0.90	0.90
F1 Score	0.90	0.90

Table 3: LDA experiment results

The best performing model configuration for the Linear Discriminant Analysis (LDA) is the one utilizing One-Hot Encoding combined with SMOTE, as indicated by the uniformly high scores of 0.89 for accuracy and 0.90 for precision, recall, and F1 score. This approach has outperformed the other methods, including those with only label encoding, label encoding with SMOTE, and one with additional hyperparameter tuning, showcasing the effectiveness of One-Hot Encoding with SMOTE in handling categorical data and class imbalance.

5.2 Linear SVM

5.2.1 Best Model

One-Hot Encoding + GridSearch (c) + SMOTE

```

> ~
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

# Resampling with SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Setting up the parameter grid
param_grid = {
    'C': [1, 10] # Reduced number of values for simplicity
}

# Initialize GridSearchCV with LinearSVC and fewer folds
grid_search = GridSearchCV(LinearSVC(), param_grid, cv=3, scoring='accuracy', verbose=1, n_jobs=-1)

# Fit GridSearchCV on the resampled training data
grid_search.fit(X_train_smote, y_train_smote)

# Best model after grid search
best_svm_model = grid_search.best_estimator_

# Evaluate the best model
evaluate_model(best_svm_model, X_train_smote, y_train_smote, X_test, y_test)

```

[48] ✓ 4.9s

```

... Fitting 3 folds for each of 2 candidates, totalling 6 fits
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm\_classes.py:32: FutureWarning: The class attribute of the Kernel class is deprecated. Use the 'kernel' attribute instead.
warnings.warn(
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm\_base.py:1250: ConvergenceWarning: SVC failed to converge with stopping criterion 0.0001.
warnings.warn(
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm\_classes.py:32: FutureWarning: The class attribute of the Kernel class is deprecated. Use the 'kernel' attribute instead.
warnings.warn(
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm\_base.py:1250: ConvergenceWarning: SVC failed to converge with stopping criterion 0.0001.
warnings.warn(
Accuracy: 0.90
Precision: 0.91
Recall: 0.91
F1 Score: 0.91

```

Figure 25: Best Linear SVM Model

One-Hot Encoding, combined with SMOTE, effectively dealt with categorical data and class imbalance, while GridSearch optimized the 'C' parameter for the SVM, resulting in the best Linear SVM model performance by fine-tuning it to the specifics of the data.

5.2.2 Worst Model

Label Encoding Without SMOTE

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)

X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the GridSearchCV object
linear_svm = SVC(kernel='linear', C=1)

linear_svm_model = linear_svm.fit(X_train, y_train)
evaluate_model(linear_svm_model, X_train, y_train, X_test, y_test)
```

[16] ✓ 58.0s

... Accuracy: 0.84
Precision: 0.86
Recall: 0.85
F1 Score: 0.85

Figure 26: Worst Linear SVM Model

The Linear SVM with Label Encoding and without SMOTE likely underperformed because Label Encoding can introduce an artificial order to categories that the SVM might misinterpret as a numerical relationship, and without SMOTE, any imbalance in the data would not be corrected, possibly leading the model to be biased towards the majority class.

5.2.3 Experiment Outcome (Linear SVM)

Linear SVM	Label Encoding	Label Encoding + Smote	One-Hot Encoding	One-Hot Encoding + SMOTE
Accuracy	0.84	0.84	0.90	0.90

Precision	0.86	0.87	0.91	0.91
Recall	0.85	0.84	0.90	0.90
F1 Score	0.85	0.85	0.90	0.90

Linear SVM (Hyper Tuning)	One-Hot + c + SMOTE	One-Hot + tol + SMOTE
Accuracy	0.90	0.90
Precision	0.91	0.90
Recall	0.90	0.90
F1 Score	0.90	0.91

Table 4: Linear SVM experiment results

The table (table 4) compares the performance of a Linear SVM classifier with different data preprocessing strategies, indicating that One-Hot Encoding combined with SMOTE yields the best results with an accuracy, precision, recall, and F1 score all at 0.90 or above. Further hyperparameter tuning maintains these high metrics. No performance increase with hypertuning

5.3 Logistic Regression

5.3.1 Best Model

One Hot encoding With SMOTE

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
# Fit the grid search to the data
log_reg.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(log_reg, X_train_smote, y_train_smote, X_test, y_test)
```

[81] ✓ 1.8s

... Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.91

Figure 27: Best Logistic Regression Model

The best Logistic Regression model used One-Hot Encoding and SMOTE because they helped handle categorical data and balanced classes. Hyperparameter tuning didn't improve results possibly because PCA, which was applied during preprocessing, may have already optimized feature representation.

5.3.2 Worst Model

Label Encoding Without SMOTE

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

# Fit the grid search to the data
log_reg.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(log_reg, X_train, y_train, X_test, y_test)
```

[64] ✓ 0.5s

... Accuracy: 0.83
Precision: 0.85
Recall: 0.86
F1 Score: 0.85

Figure 28: Best Logistic Regression Model

The worst Logistic Regression model used Label Encoding without SMOTE, likely struggling due to the model interpreting the encoded labels as ordinal data, and the absence of SMOTE not to addressing any underlying class imbalances in the dataset.

5.3.3 Experiment Results (Logistic Regression)

Logistic Regression	Label Encoding	Label Encoding + Smote	One-Hot Encoding	One-Hot Encoding + SMOTE
Accuracy	0.83	0.84	0.90	0.90
Precision	0.85	0.86	0.91	0.91
Recall	0.86	0.84	0.90	0.90
F1 Score	0.85	0.85	0.91	0.91

LOG Reg (Hyper Tuning)	One-Hot + c + SMOTE
Accuracy	0.90
Precision	0.91
Recall	0.90
F1 Score	0.90

Table 5: Logistic Regression SVM experiment results

One-Hot Encoding with SMOTE outperforms the others, achieving 0.90 accuracy, and 0.91 for precision, recall, and F1 score .Hyper Tuning was applied, however, it saw a slight decrease in performance. SO model was discarded. Final decision was no hyper tuning.

5.4 RBF SVM

5.4.1 Best Model

One Hot encoding without SMOTE

RBF SVM

```
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train the SVC model with an RBF kernel
rbf_svm = SVC(kernel='rbf', gamma='scale', C=1)
rbf_svm_model = rbf_svm.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(rbf_svm_model, X_train, y_train, X_test, y_test)
```

[48] ✓ 43.6s

... Accuracy: 0.94
Precision: 0.96
Recall: 0.94
F1 Score: 0.95

Figure 29: Best RBF SVM Model

The best performance using an RBF SVM with One-Hot Encoding, but without SMOTE, suggests that in this case, the SVM was able to effectively manage the clear separation of categories provided by One-Hot Encoding and the inherent data distribution was sufficient without the need for SMOTE to balance class representation. Smote slightly reduced performance.

5.4.2 Worst Model

Label Encoding Without SMOTE

```
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train the SVC model with an RBF kernel
rbf_svm = SVC(kernel='rbf', gamma='scale', C=1)
rbf_svm_model = rbf_svm.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(rbf_svm_model, X_train, y_train, X_test, y_test)
```

[56] ✓ 21.7s

... Accuracy: 0.91
Precision: 0.92
Recall: 0.92
F1 Score: 0.92

Figure 30: Worst RBF SVM Model

The worst model for RBF SVM used Label Encoding without SMOTE, maybe underperforming due to inappropriate numerical assumptions from Label Encoding and unaddressed class imbalances.

5.4.3 Experiment Results (RBF SVM)

RBF SVM	Label Encoding	Label Encoding + Smote	One-Hot Encoding	One-Hot Encoding + SMOTE
Accuracy	0.91	0.91	0.94	0.94
Precision	0.92	0.93	0.96	0.96
Recall	0.92	0.91	0.94	0.93
F1 Score	0.92	0.92	0.95	0.95



LOG Reg (Hyper Tuning)	One-Hot + PCA + Random-search	One-Hot + PCA + Grid-search
Accuracy	0.94	0.92
Precision	0.95	0.94
Recall	0.93	0.91
F1 Score	0.94	0.92

Table 6: RBF SVM experiment results

The best DBF SVM model produced was done using One-Hot encoding without SMOTE. Hyper tuning (random search and grid search) reduced performance. HOWEVER, this could be because PCA had to be applied in order to decrease performance overhead. DUE to computational limitations, it was not possible to test A One Hot encoded model without PCA and with random search & grid search.

5.5 Gradient-Boosted Trees

5.5.1 Best Model

WITH Hyper Tuning (One Hot + Random Search)

```
#Random Search:
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier

# Define the hyperparameters and their ranges
param_dist = {
    'n_estimators': [100, 200, 300, 400, 500],
    'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.3],
    'max_depth': [3, 4, 5, 6, 7],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize the classifier
gbc = GradientBoostingClassifier()

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=gbc, param_distributions=param_dist,
                                   n_iter=10, cv=5, n_jobs=-1, random_state=42)

# Perform Random Search
random_search.fit(X_train, y_train)

# Use the best estimator to evaluate model
best_gbc = random_search.best_estimator_
evaluate_model(best_gbc, X_train, y_train, X_test, y_test)
```

[177] ✓ 3m 27.4s

... Accuracy: 0.95
Precision: 0.95
Recall: 0.95
F1 Score: 0.95

Figure 31: Best Gradient-Boosted Trees Model

The best model for Gradient Boosted Trees involved using One-Hot Encoding combined with hype tuning optimisation through Random Search, effectively capturing the complexity of categorical data and fine-tuning the model parameters for optimal performance.

5.5.2. Worst Model

Label Encoding Without SMOTE

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier

# Initialize the classifier with subsampling
gbc = GradientBoostingClassifier()

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Fit RandomizedSearchCV on the SMOTE-enhanced training data
gbc.fit(X_train, y_train)

# Best model after random search

# Assuming evaluate_model is a predefined function that prints model performance
evaluate_model(gbc, X_train, y_train, X_test, y_test)
```

[147] ✓ 7.4s

... Accuracy: 0.91
Precision: 0.92
Recall: 0.92
F1 Score: 0.92

Figure 32: Worst Gradient-Boosted Trees Model

The worst model for Gradient Boosted Trees used Label Encoding without SMOTE, likely faltering due to the poor handling of categorical variables and failure to address class imbalances in the dataset.

5.5.3 Experiment Results (Gradient-Boosted Trees)

Gradient Booster	Label Encoding	Label Encoding + Smote	One-Hot Encoding	One-Hot Encoding + SMOTE
Accuracy	0.91	0.91	0.92	0.91
Precision	0.92	0.93	0.93	0.93
Recall	0.92	0.91	0.92	0.91
F1 Score	0.92	0.92	0.92	0.92

LOG Reg (Hyper Tuning)	One-Hot + Random search
Accuracy	0.95
Precision	0.95
Recall	0.95
F1 Score	0.95

Table7: Gradient-Boosted Trees experiment results

The table shows that a Gradient Booster model with One-Hot Encoding performs best among its comparisons, with an accuracy of 0.92 and precision, recall, and F1 all at 0.93; however, a Logistic Regression model with hyperparameter tuning via random search surpasses this, achieving 0.95 across all metrics.

6. Results

6.1 Discriminant Analysis (Best)

... Accuracy: 0.89
Precision: 0.90
Recall: 0.90
F1 Score: 0.90

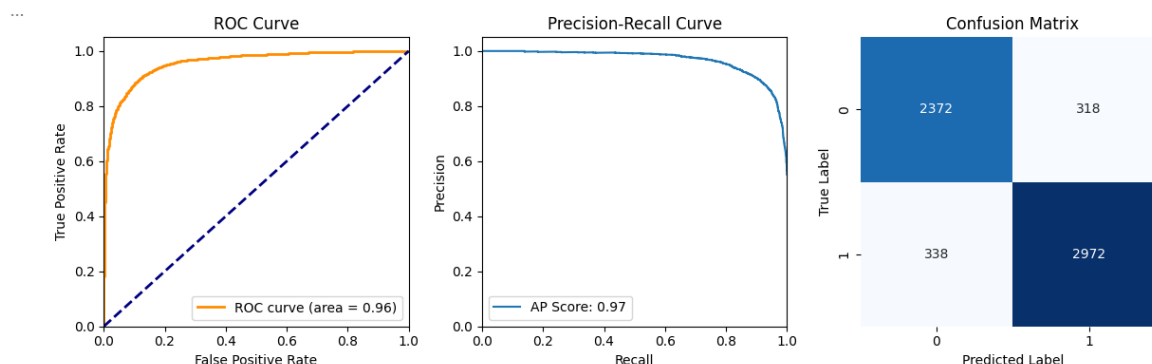


Figure 33: LDA Results Final

The performance metrics for LDA classification model are: an accuracy of 0.89 and precision, recall, and F1 score all at 0.90, along with visual representations showing a ROC curve with an area of 0.96, a Precision-Recall curve with an average precision score of 0.97, and a Confusion Matrix with 2372 true negatives, 2972 true positives, 318 false positives, and 338 false negatives.

6.2 Linear SVM (Best)

Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.91

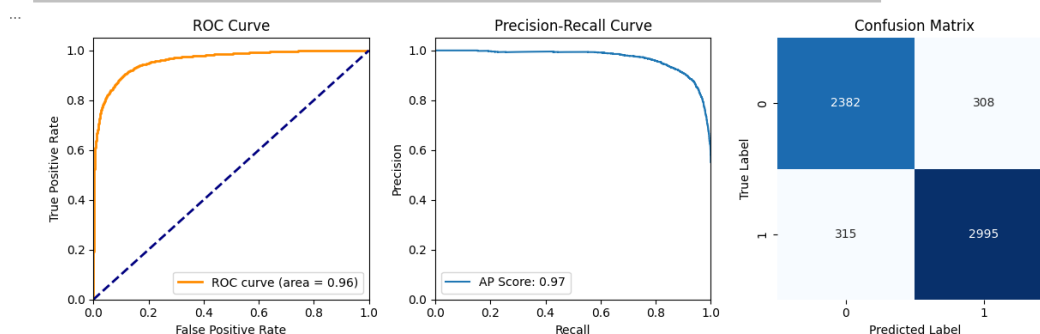


Figure 34: Linear SVM Results Final

Results of Linear SVM classifier with an accuracy of 0.90, and precision, recall, and F1 score all at 0.91. It includes a ROC curve with an area under the curve of 0.96, a Precision-Recall curve with an average precision score of 0.97, and a Confusion Matrix with 2382 true negatives, 2995 true positives, 308 false positives, and 315 false negatives.

6.3 Logistic Regression (Best)

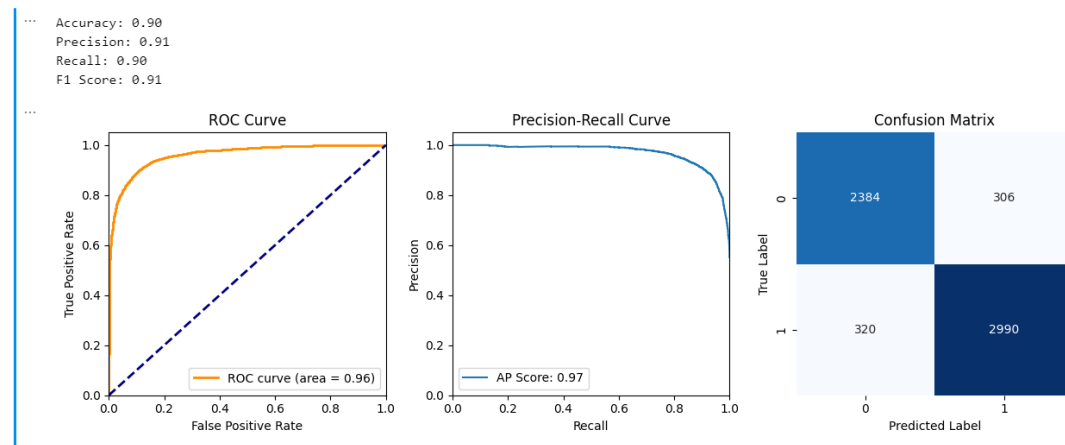


Figure 35: Logistic Regression Results Final

Performance of the Logistic Regression model: accuracy of 0.90, precision of 0.91, recall of 0.90, and an F1 score of 0.91. The visualizations include a ROC Curve with an area of 0.96, a Precision-Recall Curve with a score of 0.97, and a Confusion Matrix with 2384 true negatives, 2990 true positives, 306 false positives, and 320 false negatives

6.4 RBF SVM (Best)

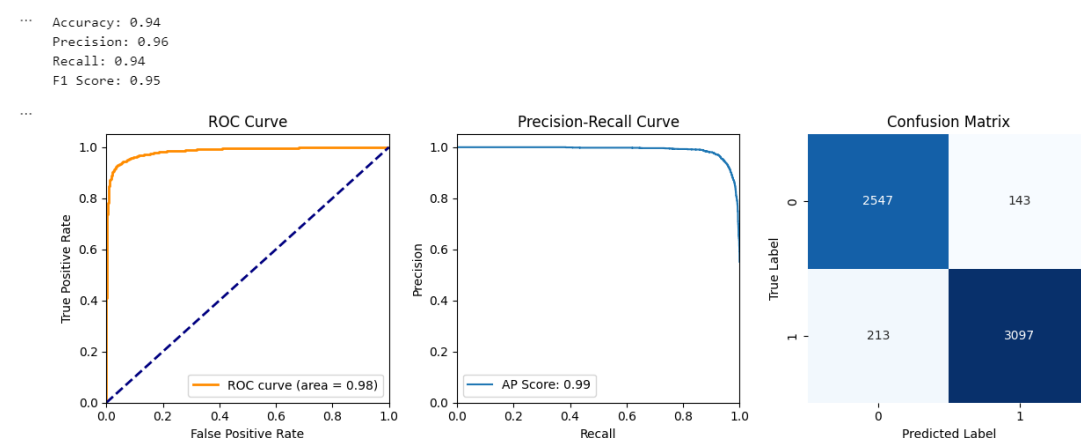


Figure 36: RBF SVM Results Final

RBF: an accuracy of 0.94, precision of 0.96, recall of 0.94, and an F1 score of 0.95. Visually, the ROC curve displays an area of 0.98, the Precision-Recall Curve shows an average precision score of 0.99, and the Confusion Matrix records 2547 true negatives, 3097 true positives, 143 false positives, and 213 false negatives.

6.5 Gradient booster classifier (Best)

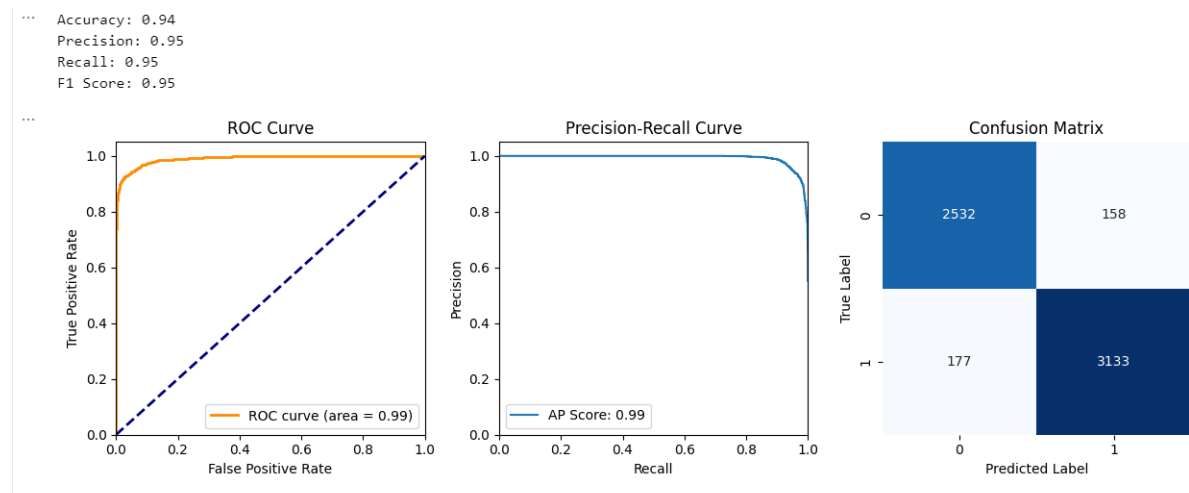


Figure 37: Gradient Booster Results Final

The gradient booster classifier: performance with an accuracy of 0.94, and precision, recall, and F1 score all at 0.95. The ROC curve has an excellent area of 0.99, the Precision-Recall Curve reports an average precision score of 0.99, and the Confusion Matrix shows 2532 true negatives, 3133 true positives, along with 158 false positives and 177 false negatives.

6.6 Best Predictors

```
feature_importances = best_gbc.feature_importances_  
  
# Get feature names from the dataset  
feature_names = X_train.columns.tolist()  
  
# Create a DataFrame of features and their importance scores  
importances_df = pd.DataFrame({  
    'Feature': feature_names,  
    'Importance': feature_importances  
})  
  
# Sort the DataFrame by importance in descending order  
importances_df = importances_df.sort_values(by='Importance', ascending=False)  
  
# Display the top 5 features  
print("Best predictors")  
print(importances_df.head(5))  
print(" ")  
print(" ")
```

[54] ✓ 0.0s

```
... Best predictors
```

	Feature	Importance
39	Onboard_entertainment_excellent	0.217161
41	Onboard_entertainment_good	0.207584
10	Seat_comfort_excellent	0.074420
11	Seat_comfort_extremely poor	0.061037
12	Seat_comfort_good	0.056617

Figure 38: GBC Importance predictors of Overall_Exprience

When analysing the best produced model (Gradient booster classifier), it can be analysed that Onboard entertainment has the most influence on a passangers Overall Experience. The second most accurate predictor of a customers experience is how comfortable the seats are.

7. Discussion

7.1. Unsupervised learning insight

- Passenger delays significantly contribute to their Overall Experience
- Most reviews are good
- Most customers are **business** passengers
- Gender of passengers is even
- Most customers are loyal (likely repeat)
- Frequent customers tend to book online
- The higher the seat comfort, the higher the perception of catering service

7.2. Supervised learning insight

- Biggest contributor to a customer's overall experience is the onboard entertainment.
- Second is seat comfort
- Third is class of travel

8. Conclusion

In conclusion both supervised and unsupervised learning was successful.

Suggestions to Japan's bullet train firm include:

- Prioritise reducing delays
- Ensure a consistent onboard entertainment system (minimise faulty screens, etc.)
- Ensure seats are clean. Replace uncomfortable seats if possible
- Ensure you hold on to loyal customers, as they form a large portion of your customer base

9. Self Reflection

Overall, this module improved my knowledge of the algorithms and models trained in this project. It also cemented the importance of data preprocessing in developing a well performing predictive model. Hurdle encountered was word count limiting.

References

- [1] Fisher, R.A. (1938). The statistical utilization of multiple measurements. *Annals of Eugenics*, 8, 376-386.
- [2] Fisher, R.A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2), 179-188.
- [3] Rao, C.R. (1948). The utilization of multiple measurements in problems of biological classification. *Journal of the Royal Statistical Society. Series B (Methodological)*, 10(2), 159-203.
- [4] McLachlan, G.J. (1992). *Discriminant Analysis and Statistical Pattern Recognition*. New York: Wiley-Interscience.
- [5] Johnson, R.A. and Wichern, D.W. (2007). *Applied Multivariate Statistical Analysis*. 6th ed. Upper Saddle River, NJ: Pearson Prentice Hall.
- [6] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. New York: Springer.
- [7] Vapnik, V. (1995) introduced the concept of Support Vector Machines in his foundational work, which underpins much of the theory behind SVMs, including those with RBF kernels.
- [8] Boser, B. E., Guyon, I. M., & Vapnik, V. N. (1992) first introduced the idea of using a kernel trick, including RBF, to increase the linear separability of data in higher-dimensional spaces.
- [9] Schölkopf, B., & Smola, A. J. (2002) provide a comprehensive guide on the theory of kernel-based learning methods, including RBF SVMs.

10. Appendix (Proof of experimentation)

10.1 Discriminant Analysis

10.1.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation)

10.1.1.1. Label Encoding Without SMOTE

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE

# # Initialize SMOTE
smote = SMOTE(random_state=42)

# # Apply SMOTE only on training data
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the Logistic Regression model
lda = LinearDiscriminantAnalysis()

# Train the LDA
log_reg_model = lda.fit(X_train, y_train)

# Evaluate the model
evaluate_model(log_reg_model, X_train, y_train, X_test, y_test)
```

[156] ✓ 0.7s

... Accuracy: 0.84
Precision: 0.85
Recall: 0.86
F1 Score: 0.85

10.1.1.2. Label Encoding With SMOTE

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE

# # Initialize SMOTE
smote = SMOTE(random_state=42)

# # Apply SMOTE only on training data
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the Logistic Regression model
lda = LinearDiscriminantAnalysis()

# Train the LDA
log_reg_model = lda.fit(X_train, y_train)

# Evaluate the model
evaluate_model(log_reg_model, X_train_smote, y_train_smote, X_test, y_test)
```

[156] ✓ 0.5s

... Accuracy: 0.84
Precision: 0.86
Recall: 0.84
F1 Score: 0.85

This snippet demonstrates the use of Label encoding SMOTE for synthetic oversampling in a dataset to address class imbalance before training a Linear Discriminant Analysis model. The model achieved an accuracy of 0.84 and an F1 score of 0.85, indicating decent performance with label encoding and smote.

5.1.1.3. One-Hot Encoding Without Smote

```
> ~
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE
from sklearn.metrics import classification_report, confusion_matrix

## Initialize SMOTE
# smote = SMOTE(random_state=42)

## Apply SMOTE only on training data
# X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the Logistic Regression model
lda = LinearDiscriminantAnalysis()

# Train the LDA
log_reg_model = lda.fit(X_train, y_train)

# Evaluate the model
evaluate_model(log_reg_model, X_train, y_train, X_test, y_test)
```

[125] ✓ 0.9s

```
... Accuracy: 0.89
Precision: 0.90
Recall: 0.90
F1 Score: 0.90
```

the Linear Discriminant Analysis model is applied directly to a dataset without employing SMOTE for oversampling, resulting in high metric scores, including an accuracy and F1 score of 0.90.

5.1.1.4. One-Hot Encoding With SMOTE

```
> ~
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE
from sklearn.metrics import classification_report, confusion_matrix

## Initialize SMOTE
smote = SMOTE(random_state=42)

## Apply SMOTE only on training data
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the Logistic Regression model
lda = LinearDiscriminantAnalysis()

# Train the LDA
log_reg_model = lda.fit(X_train, y_train)

# Evaluate the model
evaluate_model(log_reg_model, X_train_smote, y_train_smote, X_test, y_test)
```

[126] ✓ 1.4s

```
... Accuracy: 0.89
Precision: 0.90
Recall: 0.90
F1 Score: 0.90
```

10.1.2. WITH Hyper Tuning (SVD + LSQR + SMOTE)

10.1.2.1. One-Hot Encoding + SVD + SMOTE

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply SMOTE only on training data to handle class imbalance
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize LDA using the SVD solver
lda = LinearDiscriminantAnalysis(solver='svd')

# Train the LDA model on the resampled training data
lda_model = lda.fit(X_train_smote, y_train_smote)

# Assuming you have an evaluate_model function that prints model performance
evaluate_model(lda_model, X_train_smote, y_train_smote, X_test, y_test)
```

[187] ✓ 1.3s

```
... Accuracy: 0.89
Precision: 0.90
Recall: 0.90
F1 Score: 0.90
```

The code above trains the LDA model using One-Hot Encoding, SVD hyper tuning and SMOT.

10.1.2.2. One-Hot Encoding + LSQR + SMOTE

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from imblearn.over_sampling import SMOTE

# Initialize SMOTE
smote = SMOTE(random_state=42)

# Apply SMOTE only on training data to handle class imbalance
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize LDA using the LSQR solver
lda = LinearDiscriminantAnalysis(solver='lsqr')

# Train the LDA model on the resampled training data
lda_model = lda.fit(X_train_smote, y_train_smote)

evaluate_model(lda_model, X_train_smote, y_train_smote, X_test, y_test)
```

[219] ✓ 1.0s

```
... Accuracy: 0.89
Precision: 0.90
Recall: 0.90
F1 Score: 0.90
```

The code above trains the LDA model using One-Hot Encoding, LSQR hyper tuning and SMOTE

10.2 Linear SVM

10.2.1. No Hyper Tuning (Label Encoding, SMOTE)

10.2.1.1. Label Encoding Without SMOTE

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)

X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the GridSearchCV object
linear_svm = SVC(kernel='linear', C=1)

linear_svm_model = linear_svm.fit(X_train, y_train)
evaluate_model(linear_svm_model, X_train, y_train, X_test, y_test)
```

[16] ✓ 58.0s

... Accuracy: 0.84
Precision: 0.86
Recall: 0.85
F1 Score: 0.85

The code above trains a Linear SVM model using only Label encoding (no hyper tuning or SMOTE)

10.2.1.2. Label Encoding With SMOTE

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)

X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the GridSearchCV object
linear_svm = SVC(kernel='linear', C=1)

linear_svm_model = linear_svm.fit(X_train, y_train)
evaluate_model(linear_svm_model, X_train_smote, y_train_smote, X_test, y_test)
```

[16] ✓ 1m 3.4s

... Accuracy: 0.84
Precision: 0.87
Recall: 0.84
F1 Score: 0.85

The code above trains the model using both Label Encoding and SMOTE. It also shows a slight improvement in the precision metric

10.2.1.3. One-Hot Encoding Without Smote

```
from sklearn.svm import SVC

linear_svm = SVC(kernel='linear', C=1)
linear_svm_model = linear_svm.fit(X_train, y_train)
evaluate_model(linear_svm_model, X_train, y_train, X_test, y_test)
```

[224] ✓ 1m 26.6s

... Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.90

The code above trains the SVM model (One Hot encoded in other cell) with One Hot encoding (but no SMOTE). There is a significant performance improvement

5.2.1.4. One-Hot Encoding With SMOTE

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score

X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Initialize the GridSearchCV object
linear_svm = SVC(kernel='linear', C=1)

linear_svm_model = linear_svm.fit(X_train, y_train)
evaluate_model(linear_svm_model, X_train_smote, y_train_smote, X_test, y_test)
```

[63] ✓ 1m 27.5s

... Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.90

The code above trains the model using both One-Hot encoding and SMOTE. There was no improvement in performance when using SMOTE.

10.2.2. WITH Hyper Tuning (SVD + LSQR + SMOTE)

1.2.2.1. One-Hot Encoding + GridSearch (c) + SMOTE

```
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

# Resampling with SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Setting up the parameter grid
param_grid = {
    'C': [1, 10] # Reduced number of values for simplicity
}

# Initialize GridSearchCV with LinearSVC and fewer folds
grid_search = GridSearchCV(LinearSVC(), param_grid, cv=3, scoring='accuracy', verbose=1, n_jobs=-1)

# Fit GridSearchCV on the resampled training data
grid_search.fit(X_train_smote, y_train_smote)

# Best model after grid search
best_svm_model = grid_search.best_estimator_

# Evaluate the best model
evaluate_model(best_svm_model, X_train_smote, y_train_smote, X_test, y_test)
```

[48] ✓ 4.9s

... Fitting 3 folds for each of 2 candidates, totalling 6 fits
[c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_classes.py:32](#): FutureWarning: warn(
warnings.warn(
[c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_base.py:1250](#): Conver
warnings.warn(
[c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_classes.py:32](#): FutureWarning: warn(
warnings.warn(
[c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_base.py:1250](#): Conver
warnings.warn(
Accuracy: 0.90
Precision: 0.91
Recall: 0.91
F1 Score: 0.91

The code above trains the model using One-Hot Encoding, as it produced the best performance (see previous experiments), C hyper tuning through GridSearch and SMOTE (though no performance improvement was observed, only downside is slight decrease in training time).

10.2.2.2. One-Hot Encoding + Random Search (tol) + SMOTE

```
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from imblearn.over_sampling import SMOTE

# Resampling with SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Setting up the parameter grid
param_grid = {
    'tol': [1, 10] # Reduced number of values for simplicity
}

# Initialize GridSearchCV with LinearSVC and fewer folds
grid_search = GridSearchCV(LinearSVC(), param_grid, cv=3, scoring='accuracy', verbose=1, n_jobs=-1)

# Fit GridSearchCV on the resampled training data
grid_search.fit(X_train_smote, y_train_smote)

# Best model after grid search
best_svm_model = grid_search.best_estimator_

# Evaluate the best model
evaluate_model(best_svm_model, X_train_smote, y_train_smote, X_test, y_test)
```

[50] ✓ 4.2s

... Fitting 3 folds for each of 2 candidates, totalling 6 fits
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_classes.py:32: Future
warnings.warn(
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_base.py:1250: Conver
warnings.warn(
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_classes.py:32: Future
warnings.warn(
c:\Users\frosty\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\svm_base.py:1250: Conver
warnings.warn(
Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.91

The code above trains the model using One-Hot Encoding, tol hyper tuning through GridSearch and SMOTE . No overall improvement in performance when compared to c hypertuning.

10.3 Logistic Regression

10.3.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation)

10.3.1.1. Label Encoding Without SMOTE

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

# Fit the grid search to the data
log_reg.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(log_reg, X_train, y_train, X_test, y_test)
```

[64] ✓ 0.5s

... Accuracy: 0.83
Precision: 0.85
Recall: 0.86
F1 Score: 0.85

10.3.1.2. Label Encoding With SMOTE

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
# Fit the grid search to the data
log_reg.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(log_reg, X_train_smote, y_train_smote, X_test, y_test)
```

[66] ✓ 0.7s

... Accuracy: 0.84
Precision: 0.86
Recall: 0.84
F1 Score: 0.85

10.3.1.3. One Hot encoding without SMOTE

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
# Fit the grid search to the data
log_reg.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(log_reg, X_train, y_train, X_test, y_test)
```

[82] ✓ 1.7s

... Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.91

10.3.1.4. One Hot encoding With SMOTE

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
# Fit the grid search to the data
log_reg.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(log_reg, X_train_smote, y_train_smote, X_test, y_test)
```

[81] ✓ 1.8s

... Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.91

10.1.2. WITH Hyper Tuning (SVD + LSQR + SMOTE)

10.1.2.1 One-Hot + GridSeach(c) + SMOTE



```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# Define the parameter grid
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100], # Regularization parameter
    'penalty': ['l1', 'l2'], # Penalty norm
    'solver': ['liblinear', 'saga'] # Solver for optimization
}

# Initialize the Logistic Regression model
log_reg = LogisticRegression(max_iter=3000)

# Set up the grid search with cross-validation
grid_search = GridSearchCV(log_reg, param_grid, cv=5)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_
print("Best Parameters:", best_params)

# Get the best model
best_log_reg_model = grid_search.best_estimator_

# Evaluate the model using the evaluate_model function
evaluate_model(best_log_reg_model, X_train, y_train, X_test, y_test)
```

[176] ✓ 15m 1.2s

```
... Best Parameters: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
Accuracy: 0.90
Precision: 0.91
Recall: 0.90
F1 Score: 0.91
```

10.4 RBF SVM

10.4.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation)

10.4.1.1. Label Encoding Without SMOTE

```
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train the SVC model with an RBF kernel
rbf_svm = SVC(kernel='rbf', gamma='scale', C=1)
rbf_svm_model = rbf_svm.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(rbf_svm_model, X_train, y_train, X_test, y_test)
```

[56] ✓ 21.7s

... Accuracy: 0.91
Precision: 0.92
Recall: 0.92
F1 Score: 0.92

10.4.1.2. Label Encoding With SMOTE

RBF SVM

```
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train the SVC model with an RBF kernel
rbf_svm = SVC(kernel='rbf', gamma='scale', C=1)
rbf_svm_model = rbf_svm.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(rbf_svm_model, X_train_smote, y_train_smote, X_test, y_test)
```

[57] ✓ 22.8s

... Accuracy: 0.91
Precision: 0.93
Recall: 0.91
F1 Score: 0.92

The code above trains a RBF SVM model using label encoding and SMOTE. There was a 0.1 increase in precision but a 0.1 decrease in the recall score. No overall performance improvement through SMOTE (adding synthetic data to balance classes)

10.4.1.3. One Hot encoding without SMOTE

RBF SVM

```
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train the SVC model with an RBF kernel
rbf_svm = SVC(kernel='rbf', gamma='scale', C=1)
rbf_svm_model = rbf_svm.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(rbf_svm_model, X_train, y_train, X_test, y_test)
```

[48] ✓ 43.6s

... Accuracy: 0.94
Precision: 0.96
Recall: 0.94
F1 Score: 0.95

The code above uses One Hot encoding instead of label (No SMOTE), slight improvement in performance. Accuracy=+0.3, Precision =+0.3, Recall =+0.3, F1 =+0.3.

10.4.1.4. One Hot encoding With SMOTE

RBF SVM

```
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Train the SVC model with an RBF kernel
rbf_svm = SVC(kernel='rbf', gamma='scale', C=1)
rbf_svm_model = rbf_svm.fit(X_train, y_train)

# Evaluate the model using the evaluate_model function
evaluate_model(rbf_svm_model, X_train_smote, y_train_smote, X_test, y_test)
```

[39] ✓ 47.2s

... Accuracy: 0.94
Precision: 0.96
Recall: 0.93
F1 Score: 0.95

Applying smote in combination with one hot encoding actually resulted in a slight decrease in the recall score

10.4.2. WITH Hyper Tuning (Random Search + Grid Search + SMOTE)

10.4.2.1. ONEHOT + RANDOMSEARCH + PCA

```
# Create a pipeline that includes PCA, SMOTE, and SVC
pipeline = ImbPipeline([
    ('pca', PCA(n_components=0.95)), # Reduces dimensionality
    ('smote', SMOTE(random_state=42)), # Applies SMOTE for balancing the classes
    ('svc', SVC(kernel='rbf', random_state=42)) # The SVC classifier
])

# Print the pipeline structure
print("Pipeline structure:")
print(pipeline)

# Define parameter distributions for RandomizedSearchCV, note the double underscore usage
param_distributions = {
    'svc__C': np.logspace(-1, 2, 4), # Simplified range for C
    'svc__gamma': ['scale', 'auto'], # Reduced options for gamma
    'pca__n_components': [0.85, 0.95] # Fewer options for PCA components
}

# Set up RandomizedSearchCV with the pipeline
random_search = RandomizedSearchCV(
    pipeline,
    param_distributions=param_distributions,
    n_iter=5, # Reduced number of iterations
    cv=2, # Reduced number of folds
    verbose=2,
    random_state=42
)

# Fit RandomizedSearchCV on the training data
random_search.fit(X_train, y_train)

# Best model after random search
best_model = random_search.best_estimator_

# Evaluate the best model
evaluate_model(best_model, X_train, y_train, X_test, y_test)
```

```
# Fit RandomizedSearchCV on the training data
random_search.fit(X_train, y_train)

# Best model after random search
best_model = random_search.best_estimator_

# Evaluate the best model
evaluate_model(best_model, X_train, y_train, X_test, y_test)
```

[88] ✓ 1m 58.6s

... Pipeline structure:

```
Pipeline(steps=[('pca', PCA(n_components=0.95)),
                 ('smote', SMOTE(random_state=42)),
                 ('svc', SVC(random_state=42))])
```

Fitting 2 folds for each of 5 candidates, totalling 10 fits

```
[CV] END pca__n_components=0.85, svc__C=0.1, svc__gamma=scale; total time= 9.4s
[CV] END pca__n_components=0.85, svc__C=0.1, svc__gamma=scale; total time= 9.8s
[CV] END pca__n_components=0.85, svc__C=0.1, svc__gamma=auto; total time= 9.5s
[CV] END pca__n_components=0.85, svc__C=0.1, svc__gamma=auto; total time= 10.1s
[CV] END pca__n_components=0.85, svc__C=10.0, svc__gamma=auto; total time= 6.4s
[CV] END pca__n_components=0.85, svc__C=10.0, svc__gamma=auto; total time= 6.5s
[CV] END pca__n_components=0.95, svc__C=100.0, svc__gamma=scale; total time= 7.4s
[CV] END pca__n_components=0.95, svc__C=100.0, svc__gamma=scale; total time= 7.4s
[CV] END pca__n_components=0.95, svc__C=10.0, svc__gamma=auto; total time= 6.3s
[CV] END pca__n_components=0.95, svc__C=10.0, svc__gamma=auto; total time= 6.3s
Accuracy: 0.94
Precision: 0.95
Recall: 0.93
F1 Score: 0.94
```

10.4.2.2. ONEHOT + GRIDSEARCH + PCA

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE
import numpy as np

# Adjusting the pipeline
pipeline = ImbPipeline([
    ('pca', PCA(n_components=0.80)), # More aggressive reduction
    ('smote', SMOTE(random_state=42, sampling_strategy='auto')), # Reduced SMOTE impact
    ('svc', SVC(kernel='rbf', random_state=42))
])

# Simplified parameter grid
param_grid = {
    'svc__C': [0.1, 1, 10], # Fewer options for C
    'svc__gamma': ['scale', 'auto'] # Simplified gamma options
}

# Set up GridSearchCV with fewer CV folds and increased parallel processing
grid_search = GridSearchCV(
    pipeline,
    param_grid=param_grid,
    cv=2, # Reduced number of folds
    verbose=2,
    n_jobs=-1 # Use all available cores
)

# Fitting the grid search to the training data
grid_search.fit(X_train, y_train)

# Output best model parameters
print("Best Parameters:", grid_search.best_params_)

# Evaluate the best model
evaluate_model(grid_search.best_estimator_, X_train, y_train, X_test, y_test)
```

```
... Fitting 2 folds for each of 6 candidates, totalling 12 fits
Best Parameters: {'svc__C': 10, 'svc__gamma': 'auto'}
Accuracy: 0.92
Precision: 0.94
Recall: 0.91
F1 Score: 0.92
```

The code (see figure X) trains the RBF SVM model using One-Hot encoding, Grid Search and PCA. PCA was applied to improve training times. The accuracy achieved was 92% with precision of 94%

10.5 Gradient-Boosted Trees

10.5.1. No Hyper Tuning (Label Encoding, One-Hot, SMOTE Experimentation)

10.5.1.1. Label Encoding Without SMOTE

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier

# Initialize the classifier with subsampling
gbc = GradientBoostingClassifier()

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Fit RandomizedSearchCV on the SMOTE-enhanced training data
gbc.fit(X_train, y_train)

# Best model after random search

# Assuming evaluate_model is a predefined function that prints model performance
evaluate_model(gbc, X_train, y_train, X_test, y_test)
```

[147] ✓ 7.4s

... Accuracy: 0.91
Precision: 0.92
Recall: 0.92
F1 Score: 0.92

The code above trains the model with label encoding but not smote.

10.5.1.2. Label Encoding With SMOTE

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier

# Initialize the classifier with subsampling
gbc = GradientBoostingClassifier()

smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Fit RandomizedSearchCV on the SMOTE-enhanced training data
gbc.fit(X_train_smote, y_train_smote)

# Best model after random search

# Assuming evaluate_model is a predefined function that prints model performance
evaluate_model(gbc, X_train_smote, y_train_smote, X_test, y_test)
```

[145] ✓ 10.2s

... Accuracy: 0.91
Precision: 0.93
Recall: 0.91
F1 Score: 0.92

The code above trains with label encoding and SMOTE.

10.5.1.3. OneHot Encoding Without SMOTE

Gradient-boosted Trees

```
>   
#Random Search:  
from sklearn.model_selection import RandomizedSearchCV  
from sklearn.ensemble import GradientBoostingClassifier  
  
# Initialize the classifier  
gbc = GradientBoostingClassifier()  
  
# Perform Random Search  
gbc.fit(X_train, y_train)  
  
# Use the best estimator to evaluate model  
evaluate_model(gbc, X_train, y_train, X_test, y_test)
```

85] ✓ 10.6s

```
.. Accuracy: 0.92  
Precision: 0.93  
Recall: 0.92  
F1 Score: 0.92
```

10.5.1.3. OneHot Encoding With SMOTE

```
▷   
from sklearn.model_selection import RandomizedSearchCV  
from sklearn.ensemble import GradientBoostingClassifier  
  
# Initialize the classifier with subsampling  
gbc = GradientBoostingClassifier()  
  
smote = SMOTE(random_state=42)  
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)  
  
# Fit RandomizedSearchCV on the SMOTE-enhanced training data  
gbc.fit(X_train_smote, y_train_smote)  
  
# Best model after random search  
  
# Assuming evaluate_model is a predefined function that prints model performance  
evaluate_model(gbc, X_train_smote, y_train_smote, X_test, y_test)
```

[129] ✓ 12.8s

```
... Accuracy: 0.91  
Precision: 0.93  
Recall: 0.91  
F1 Score: 0.92
```

10.2.2. WITH Hyper Tuning (One Hot + Random Search)

```
#Random Search:
from sklearn.model_selection import RandomizedSearchCV
from sklearn.ensemble import GradientBoostingClassifier

# Define the hyperparameters and their ranges
param_dist = {
    'n_estimators': [100, 200, 300, 400, 500],
    'learning_rate': [0.01, 0.05, 0.1, 0.2, 0.3],
    'max_depth': [3, 4, 5, 6, 7],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize the classifier
gbc = GradientBoostingClassifier()

# Initialize RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=gbc, param_distributions=param_dist,
                                   n_iter=10, cv=5, n_jobs=-1, random_state=42)

# Perform Random Search
random_search.fit(X_train, y_train)

# Use the best estimator to evaluate model
best_gbc = random_search.best_estimator_
evaluate_model(best_gbc, X_train, y_train, X_test, y_test)
```

[177] ✓ 3m 27.4s

```
... Accuracy: 0.95
Precision: 0.95
Recall: 0.95
F1 Score: 0.95
```