King Saud University
College of Computer and Information Sciences
Information Technology Department

Big Data System IT 462

Airlines Customer satisfaction

Prepared by:

Name	ID
Yara Bahmaid	443200749
Reema Aljalajel	443201121
Aljohara Albanyan	443200994
Shahd Alfahd	443200491
Raseel Alrawdhan	443200592
Majd Aljuraysi	443200637

Supervised by

Dr. Mashael Alsaleh

Table of Contents

Table of Figure

1. Introduction

Airline customer satisfaction is a critical factor for airlines' success. in today's competitive market, airlines must constantly strive to improve their customer satisfaction to attract and retain passengers. The dataset provided pertains to Invistico Airlines, an airline organization. It contains customer information, including feedback and flight data from previous passengers. The primary objective of this dataset is to predict the likelihood of satisfaction for potential future customers based on a range of parameter values. Furthermore, the airline aims to identify specific aspects of their services that require greater emphasis to ensure higher levels of customer satisfaction. In this report, we first describe the dataset, including its size, format, and key attributes. Then, we apply data preprocessing techniques such as handling missing values, removing outliers, and transforming the data through encoding, normalization, and discretization. Additionally, we perform data reduction through sampling and address class imbalance to improve model performance.

2. Dataset

2.1 - Data size and format

The dataset is in CSV format containing text and numerical data.

It was selected from Kaggle and countian **23 column** and **129,880 row**.

Kaggle URL: https://www.kaggle.com/datasets/sjleshrac/airlines-customer-satisfaction?resource=download .

2.2 - Customer Satisfaction Attributes

The Table 1 below provides a structured overview of key attributes influencing customer satisfaction It details each attribute's description, type, and possible values.

Table 1: dataset attribute

Attribute	Description	Type	Possible Values
Satisfaction	Whether the client is satisfied or not	Binary	Satisfied - Dissatisfied
Gender	The gender of the client	Binary	Female - Male
Customer Type	Whether the client is loyal or not	Binary	Loyal Customer - Disloyal Customer
Age	The age of the customer	Numeric	Between 7 - 85
Type of travel	Purpose of the flight	Binary	Personal travel - Business travel
Class	Type of airplane seat	Nominal	Eco - Eco Plus - Business
Flight Distance	How long is the flight distance	Numeric	From 50 to 6951
Seat comfort	Is the seat comfortable or not	Ordinal	From 0 to 5
Departure/Arrival time convenient	If the time is convenient	Ordinal	From 0 to 5
Food and drink	What is the quality of the food and drink	Ordinal	From 0 to 5
Gate location	The client's rate about the gate location	Ordinal	From 0 to 5
Inflight WiFi service	The client's rate for this service	Ordinal	From 0 to 5
Inflight entertainment	If the flight contains entertainment services	Ordinal	From 0 to 5
Online support	The client's rate for this service	Ordinal	From 0 to 5
Ease of Online Booking	The client's rate about online booking	Ordinal	From 0 to 5
On-board service	The client's rate about the on-board service	Ordinal	From 0 to 5
Leg room service	Does the client have space for their legs	Ordinal	From 0 to 5
Baggage handling	Does the flight have this service and how is it	Ordinal	From 0 to 5
Check-In service	The client's rate for this service	Ordinal	From 0 to 5
Cleanliness	How clean is the plane	Ordinal	From 0 to 5
Online boarding	The client's rate for this service	Ordinal	From 0 to 5
Departure Delay in minutes	How many minutes the departure was delayed	Numeric	From 0 to 1592
Arrival Delay in minutes	How many minutes the arrival was delayed	Numeric	From 0 to 1584

3- Data Preprocessing

Data preprocessing is a critical step in the machine learning workflow, essential for optimizing model performance and ensuring reliable predictions. Common scenarios demanding preprocessing include addressing missing values by imputation or removal, handling outliers to prevent them from skewing model behavior, normalizing or scaling features to bring them to a consistent scale, encoding categorical variables for numerical compatibility, and engineering new features to enhance model understanding.

3.1 - Data Cleaning

3.1.1 - Handling Missing Values

Missing data can introduce challenges during data analysis or the construction of machine learning models, as they may lead to inaccurate results or errors. In our dataset, we found that most attributes had no missing values, except for Arrival Delay in Minutes, which had 393 missing entries. To address this, we delete the missing values since we don't show any related attributes with the Arrival Delay in Minutes.

```
[16] data=data.dropna(subset=['Arrival Delay in Minutes'])
```

Figure 1:Handling missing value.

3.1.2 - Remove outliers

Outliers are extreme values that can distort analysis and negatively impact model performance. To address this, we applied the Z-score method with a threshold of 3 to detect and remove outliers. We identified Departure Delay in Minutes, Arrival Delay in Minutes, and Flight Distance as highly variable attributes prone to outliers. After removal, the dataset size was reduced from 129,880 to 122,665 rows, ensuring cleaner and more reliable data for analysis.

```
from scipy.stats import zscore

# Compute Z-scores
z_scores = np.abs(zscore(data.select_dtypes(include=np.number)))

# Set threshold (e.g., Z-score > 3)
outliers_z = (z_scores > 3)

# Count outliers per column
print("Outliers per column using Z-score:\n", outliers_z.sum())
```

Figure 2: Z-scores.

```
[13] # Make a copy of the data to preserve the original
     data_cleaned = data.copy()

     # List of specific columns to remove outliers from
     cols_to_clean = ["Departure Delay in Minutes", "Arrival Delay in Minutes", "Flight Distance"]

     # Remove outliers for each selected column
     for col in cols_to_clean:
         z_scores = np.abs(zscore(data_cleaned[col]))  # Compute Z-scores for the column
         data_cleaned = data_cleaned[z_scores < 3]  # Keep rows where Z-score < 3

     # Print shape before and after removing outliers
     print(f"Original dataset shape: {data.shape}")
     print(f"Cleaned dataset shape: {data_cleaned.shape}")
     # Update 'data' to the cleaned version
     data = data_cleaned

     Original dataset shape: (129880, 23)
     Cleaned dataset shape: (122665, 23)
```

Figure 3:Remove outliers.

3.2 - Data Transformation

Data transformation is the process of converting data into a structured and meaningful format for efficient processing and analysis. It enhances data consistency, accuracy, and usability across different systems and applications. In our dataset, we applied Encoding, Normalization using Min-Max Scaling, and Discretization to standardize the data, ensure uniformity, and simplify analysis. These transformations enhance data consistency and improve the effectiveness of subsequent processing and modelling.

3.2.1 – Encoding

Encoding is crucial in machine learning because it changes raw data into a format that algorithms can understand. This often means turning categories or words into numbers, making it easier for computers to work with and analyze the information.

```
[14] # Encode 'Gender' column (1 for Female, 2 for Male)
     data['Gender'] = data['Gender'].astype('category').cat.codes + 1

[15] # Encode 'Customer Type' column (1 for Loyal Customer, 2 for Disloyal Customer)
     data['Customer Type'] = data['Customer Type'].astype('category').cat.codes + 1

[16] # Encode 'Type of Travel' column (1 for Personal Travel, 2 for Business Travel)
     data['Type of Travel'] = data['Type of Travel'].astype('category').cat.codes + 1

[17] # Encode 'Class' column (1 for Eco, 2 for Business, 3 for Eco Plus)
     data['Class'] = data['Class'].astype('category').cat.codes + 1

[18] # Define a dictionary to manually recode 'satisfied' to 1 and 'dissatisfied' to 0
     satisfaction_dict = {'dissatisfied': 0, 'satisfied': 1}

[19] # Replace values in 'Satisfaction' column using the dictionary
     data['satisfaction'] = data['satisfaction'].map(satisfaction_dict)

[20] # Display first few rows to confirm changes
     print(data.head())
```

Figure 4:Encoding.

3.2.2 - Normalization Min-Max Scaling

We used a technique called max-min normalization to make sure our data was consistently scaled. This method adjusts the values of certain attributes to fit within a range from 0 to 1. We applied this normalization to three specific attributes: Arrival Delay in Minutes, and Departure Delay in Minutes. because These columns have high variance and large numerical ranges, which can affect models that rely on distance metrics. Normalizing the dataset in this way makes the attributes more uniform and comparable, which helps us perform accurate analysis and modeling for predicting satisfaction, as demonstrated in the results.

```
[52] # Define normalization function
     def normalize(x):
         return (x - x.min()) / (x.max() - x.min())

[53] data['Arrival Delay in Minutes'] = normalize(data['Arrival Delay in Minutes'])

[54] data['Departure Delay in Minutes'] = normalize(data['Departure Delay in Minutes'])

[55] # Display first few rows to confirm normalization
     print(data[['Arrival Delay in Minutes', 'Departure Delay in Minutes']].head(10))

     Arrival Delay in Minutes  Departure Delay in Minutes
0                   0.000000                    0.000000
2                   0.000000                    0.000000
3                   0.000000                    0.000000
4                   0.000000                    0.000000
5                   0.000000                    0.000000
6                   0.192308                    0.133858
7                   0.000000                    0.000000
8                   0.000000                    0.000000
9                   0.333333                    0.236220
10                  0.615385                    0.370079
```

Figure 5: Normalization Min-Max Scaling.

3.2.3 – Discretization

We have categorized flight distances into four finite elements: 0, 1400, 2800, 4200, 7000 and, inf. Into the showing labels 1 as short , 2 as medium, 3 as moderate, 4 as long , and 5 as very long ,this allows for detailed analysis and insights into various aspects of air travel.

```
[56] # Define bin edges and labels
     bin_edges = [0, 1400, 2800, 4200, 7000, float('inf')]  # Define bins
     bin_labels = [1, 2, 3, 4, 5]  # Labels corresponding to bins

     # Perform discretization using pd.cut()
     data['Flight Distance'] = pd.cut(data['Flight Distance'], bins=bin_edges, labels=bin_labels)
```

Figure 6:Discretization.

3.3 - Data integration

Data integration is unnecessary for our dataset as it already contains over 100,000 rows with comprehensive features relevant to airline satisfaction analysis. Adding external data would introduce complexity, redundancy, and potential inconsistencies without adding significant value.

3.4 - Data reduction

We will be reducing the number of rows in the dataset while keeping all columns intact. The dataset already contains comprehensive features relevant to airline satisfaction analysis, so removing columns is unnecessary. Instead, reducing the number of rows will help optimize processing while still maintaining the essential data needed for analysis.

3.4.1 - Data Sampling

We will use Stratified Sampling to reduce the number of rows while maintaining the original class distribution. This ensures that the sampled dataset remains representative of the full dataset. By applying stratified sampling, we can optimize processing while preserving the integrity of the analysis.

```
[57] from sklearn.model_selection import train_test_split

     # Define the sample size
     sample_size = 0.2

     # Perform stratified sampling based on the "satisfaction" column
     data_sampled, _ = train_test_split(data, test_size=(1 - sample_size), stratify=data["satisfaction"], random_state=7)

     # Print dataset sizes before and after sampling
     print(f"Original dataset size: {data.shape[0]}")
     print(f"Sampled dataset size: {data_sampled.shape[0]}")

     # Check class distribution in the sample
     print("\nClass distribution in sampled dataset:")
     print(data_sampled["satisfaction"].value_counts(normalize=True))
     data = data_sampled

     Original dataset size: 122665
     Sampled dataset size: 24533

     Class distribution in sampled dataset:
     satisfaction
     1    0.553377
     0    0.446623
```

Figure 7:Stratified Sampling .

3.4.2 - Data imbalanced

We checked the class imbalance by calculating the ratio between the majority and minority classes. To balance the data, we applied undersampling, reducing the majority class to match the minority class, achieving a 1:1 ratio.

```
[58] # Compute the ratio of the minority class to the majority class
     class_counts = data["satisfaction"].value_counts()
     imbalance_ratio = class_counts.min() / class_counts.max()

     print(f"Imbalance Ratio: {imbalance_ratio:.2f}")

     Imbalance Ratio: 0.81
```

Figure 8:Imbalance ratio.

```
from imblearn.under_sampling import RandomUnderSampler

# Define the target variable
X = data.drop(columns=["satisfaction"])  # Features
y = data["satisfaction"]  # Target

# Apply Random Undersampling
undersampler = RandomUnderSampler(sampling_strategy='auto', random_state=7)
X_resampled, y_resampled = undersampler.fit_resample(X, y)

# Combine into a new DataFrame
data_undersampled = pd.concat([X_resampled, y_resampled], axis=1)

# Check new class distribution
print("New class distribution after undersampling:")
print(data_undersampled["satisfaction"].value_counts())

# Print dataset shape before and after
print(f"\nOriginal dataset size: {data.shape[0]} rows")
print(f"Reduced dataset size: {data_undersampled.shape[0]} rows")
data = data_undersampled

New class distribution after undersampling:
satisfaction
0    10957
1    10957
Name: count, dtype: int64

Original dataset size: 24533 rows
Reduced dataset size: 21914 rows
```

Figure 9:Undersampling.

```
[60] # Compute the ratio of the minority class to the majority class
     class_counts = data["satisfaction"].value_counts()
     imbalance_ratio = class_counts.min() / class_counts.max()

     print(f"Imbalance Ratio: {imbalance_ratio:.2f}")

     Imbalance Ratio: 1.00

[61] data.shape

     (21914, 23)
```

Figure 10:Ratio After balance.

4- RDD Operations

4.1 Introduction

Customer satisfaction is a key driver of competitiveness in the airline industry. This section utilizes passenger reviews from Invistico Airlines to better understand the factors that result in satisfaction. The data set comprises passenger demographic and flight information, as well as ratings of specific aspects of service, including seat comfort, food quality, and overall experience.

By analyzing and performing data analysis methods, such as Resilient Distributed Datasets (RDDs), we will uncover patterns and insights that can help the airline address areas and concerns related to satisfaction. The information obtained from the analysis will guide the airline's ability to focus on priorities that will build customer satisfaction and retention, as well as lead to improved efficiency and experience for travelers.

4.2 Analysis of Customer Satisfaction

In this section, we detail a set of operations we performed on the Invistico Airlines customer satisfaction dataset that reveal valuable insights for service enhancement in this context. This is a valuable process to help identify customer satisfaction influencers and provide recommendations to the airline, based on the findings of our analysis, to enhance its service.

Using RDD, we performed various transformations and actions on the data to analyze relationships between customer satisfaction and several factors such as flight class, age, and quality of service. For each operation, we provided an analysis of the outcome with recommendations for the airline on how best to leverage the results to optimize service.

4.2.1 Loading the Cleaned Dataset in Scala

This code snippet shows how the cleaned CSV file was read into a DataFrame using Spark's built-in read method with header and schema inference options enabled.

```
scala> val df = spark.read
df: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@21890674

scala>   .option("header", "true")
res0: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@21890674

scala>   .option("inferSchema", "true")
res1: org.apache.spark.sql.DataFrameReader = org.apache.spark.sql.DataFrameReader@21890674

scala>   .csv("cleaned_data.csv")
res2: org.apache.spark.sql.DataFrame = [Gender: int, Customer Type: int ... 21 more fields]
```

Figure 11: Scala Code of Loading the data

4.2.2 Age Group Satisfaction Analysis

Objective

The purpose of this analysis is to compute the average satisfaction for each age group. The dataset provides both the age of each passenger and the associated satisfaction rating. We can group the passengers into age categories and analyze their satisfaction levels and how satisfaction varies by age.

RDD Transformations and Actions

- **Transformations**
    - **map()**: This transformation is used to map each passenger's age to a specific age group and their corresponding satisfaction rating. The age groups are defined as follows:
        - 18-29
        - 30-39
        - 40-49
        - 50-59
        - 60+

The age groups used in this analysis (18–29, 30–39, 40–49, 50–59, and 60+) follow a standard demographic segmentation approach commonly adopted in market research and social science studies [2]. These categories help highlight behavioral trends and satisfaction levels across distinct life stages.

    - **groupByKey()**: This transformation groups the data by **age group** so that we can calculate the satisfaction for each group independently.
    - **mapValues()**: This transformation is applied after the groupByKey() to calculate the total satisfaction and the count for each group using the **reduce()** function.

- **Actions**
  - **reduce()**: We use **reduce()** to aggregate the satisfaction ratings for each age group. The reduce() operation combines the satisfaction scores and counts incrementally, ensuring efficient aggregation of data.
  - **collect()**: This action is used to bring the final results back to the driver node. It collects the average satisfaction for each age group after calculating the averages and prints the results.

Scala Code

```
scala> // Step 1: Define age range function

scala> def ageRange(age: Int): String = {
     |   if (age < 18) "Under 18"
     |   else if (age < 30) "18-29"
     |   else if (age < 40) "30-39"
     |   else if (age < 50) "40-49"
     |   else if (age < 60) "50-59"
     |   else "60+"
     | }
ageRange: (age: Int)String

scala> // Step 2: Convert DataFrame to RDD and safely map each row to (age_range, satisfaction)

scala> val ageSatisfactionRDD = df.rdd.map(row => {
     |   val age = row.getAs[Int]("Age")
     |   val satisfaction = row.getAs[Any]("satisfaction") match {
     |     case d: Double => d
     |     case i: Int => i.toDouble
     |     case _ => 0.0  // Default/fallback
     |   }
     |   (ageRange(age), satisfaction)
     | })
ageSatisfactionRDD: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[45] at map at <console>:24
```

```
scala> // Step 3: Group by age range

scala> val grouped = ageSatisfactionRDD.groupByKey()
grouped: org.apache.spark.rdd.RDD[(String, Iterable[Double])] = ShuffledRDD[46] at groupByKey at <console>:23

scala> // Step 4: Reduce to get average satisfaction

scala> val reduced = grouped.mapValues(iter => {
     |   val (total, count) = iter.foldLeft((0.0, 0)) {
     |     case ((sum, cnt), value) => (sum + value, cnt + 1)
     |   }
     |   total / count
     | })
reduced: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[47] at mapValues at <console>:23

scala> // Step 5: Print results

scala> reduced.collect().foreach { case (ageGroup, avg) =>
     |   println(s"Age Group: $ageGroup, Average Satisfaction: %.2f".format(avg))
     | }
Age Group: 30-39, Average Satisfaction: 0.44
Age Group: 40-49, Average Satisfaction: 0.61
Age Group: 50-59, Average Satisfaction: 0.64
Age Group: 60+, Average Satisfaction: 0.42
Age Group: 18-29, Average Satisfaction: 0.40
Age Group: Under 18, Average Satisfaction: 0.40
```

Figure 12: Scala Code of Age group analysis

Findings and analysis

- High Satisfaction for Ages 50-59: The age group 50 to 59 is indicated as having the highest satisfaction, which shows that the airline's service is perhaps more tailored to their needs. It could be due to factors like comfort, service quality, and amenities that are more suited to this demographic.

- Dissatisfaction in Youth Age Groups: The age groups 18-29 and under 18 had the lowest satisfaction ratings. This suggests that younger travelers have different preferences or expectations, and the airline's service does not fully meet those needs. The reasons for dissatisfaction may include:

    1. A lack of entertainment options on board departing flights: This may be due to younger customers being more tech or interested in entertainment that is engaging during the flight.
    2. Menu or food preferences of younger travelers: This may contribute to the travelers being dissatisfied with the quality or variety of food.
    3. Seat comfort: younger travelers may find seat comfort not as accommodating, especially for those traveling to international or long road travel.

Recommendations and Next Steps

- For younger passengers (aged 18-29 and 17 and younger), try considering additional entertainment options, everything from a larger selection of movies and TV shows to refreshing the menu to more trendy or customizable meal options. Younger passengers may be more engaged with food preferences and entertainment methods in this age range.

- For older passengers (aged 50-59), since this group has higher levels of satisfaction, the airline company needs to analyze the amenities offered to ensure that overall premium service amenities (like comfort of seating) continue to be a consideration in the offerings and premium services to this group.

4.2.3 Baggage Handling Satisfaction Analysis

Objective

The focus of this analysis is to determine the passenger satisfaction level with respect to the Baggage Handling services. By classifying passengers' ratings as "Good" or "Bad" by their level of satisfaction (≥ 4 = "Good"), we can ascertain how many of the passengers were satisfied with the baggage handling service (≥ 4 = "Good", < 4 = "Bad").

We chose 4 as the threshold for a "Good" rating based on standard practices in customer satisfaction measurement. According to Zendesk's Customer Satisfaction Score methodology [1], ratings of 4 or 5 on a 5-point scale are typically considered indicators of customer satisfaction.

RDD Transformations and Actions

- Transformations:
    - **map():** This transformation is used to categorize each baggage handling rating as "Good" or "Bad". We check if the rating is greater than or equal to 4 to classify it as "Good" and less than 4 as "Bad".
    - **filter():** We filter the ratings into Good and Bad categories.
- Actions:
    - **count():** is used to get the total number of Good and Bad ratings for baggage handling.

Scala Code

```
scala> def baggageCategory(rating: Int): String = {
     |   if (rating >= 4) "Good" else "Bad"
     | }
baggageCategory: (rating: Int)String

scala> // Convert DataFrame to RDD and map each row to (baggageCategory, 1)

scala> val baggageSatisfactionRDD = df.rdd.map(row => {
     |   val rating = row.getAs[Int]("Baggage handling")
     |   (baggageCategory(rating), 1)
     | })
baggageSatisfactionRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[48] at map at <console>:24

scala> // Count the number of Good and Bad ratings

scala> val goodCount = baggageSatisfactionRDD.filter(_._1 == "Good").count()
goodCount: Long = 13842

scala> val badCount = baggageSatisfactionRDD.filter(_._1 == "Bad").count()
badCount: Long = 8072

scala> // Print the results

scala> println(s"Number of Good Baggage Handling Ratings: $goodCount")
Number of Good Baggage Handling Ratings: 13842

scala> println(s"Number of Bad Baggage Handling Ratings: $badCount")
Number of Bad Baggage Handling Ratings: 8072
```

Figure 13: Scala Code of Baggage Handling Satisfaction

Findings and analysis

1. General Satisfaction: Of the 13842 passengers who rated the baggage service as Good, it can be concluded that a larger majority of customers are satisfied with the service. This can be viewed as a positive sign for the airline, suggesting that the airline is most likely handling baggage well in many cases.

2. Clearly Improvement is Needed: Despite the positive rating by most passengers, the passengers giving the baggage service a Bad rating totaled 8072, which is a large number of passengers that are still unhappy with this part of the service. This will be a significant number of passengers that will require improvement to improve customer satisfaction.

3. Potential Causes of Dissatisfaction:
   - Delayed Baggage
   - Damage or Loss
   - Long wait time after passengers have claimed their baggage

Recommendations and next steps

Improve the efficiency of baggage handling and ensure employees that are physically handling baggage are well trained and follow standards of operations for baggage handling in order to minimize human error.

4.2.4 Comparing Satisfaction Between Different Service Types

Objective

The aim of this analysis is to identify which specific in-flight services receive the lowest satisfaction ratings from passengers. By pinpointing these underperforming services, airlines can focus their efforts on targeted improvements that will have the most impact on overall passenger experience.

RDD Transformations and Actions

- Transformations:
  - **flatMap():** Used to extract service-related ratings from each record and map them into key-value pairs of the form (service, rating) for the selected keys: 'Seat comfort', 'Food and drink', 'Inflight WiFi service', and 'On-board service'.
  - **aggregateByKey():** Aggregated the satisfaction ratings for each service by calculating the total sum and count of scores using both a sequence and combination operation.
  - **mapValues():** Used to calculate the average satisfaction by dividing the total sum of ratings by the number of ratings for each service.

- Action:
  - **takeOrdered():** Applied to retrieve the three services with the lowest average satisfaction scores by ordering them in ascending order.

Scala Code

```
scala> // Define the service keys we're interested in

scala> val serviceKeys = List("Seat comfort", "Food and drink", "Inflight WiFi service", "On-board service")
serviceKeys: List[String] = List(Seat comfort, Food and drink, Inflight WiFi service, On-board service)

scala>

scala> // flatMap - Extract (service, rating) pairs if the key exists in the row

scala> val serviceRatings = rdd.flatMap(row => {
     |   serviceKeys.flatMap(key =>
     |     if (row.schema.fieldNames.contains(key) && row.getAs[Any](key) != null)
     |       Some((key, row.getAs[Any](key).toString.toDouble))
     |     else None
     |   )
     | })
serviceRatings: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[60] at flatMap at <console>:24

scala>

scala> // aggregateByKey - (sum, count)

scala> val zeroValue = (0.0, 0)
zeroValue: (Double, Int) = (0.0,0)

scala> val seqOp = (acc: (Double, Int), rating: Double) => (acc._1 + rating, acc._2 + 1)
seqOp: ((Double, Int), Double) => (Double, Int) = $Lambda/0x000001aa02299688@398f2bd3
```

```
scala> val combOp = (acc1: (Double, Int), acc2: (Double, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
combOp: ((Double, Int), (Double, Int)) => (Double, Int) = $Lambda/0x000001aa0229b4c0@5dbd1f7e

scala>

scala> val serviceAggregated = serviceRatings.aggregateByKey(zeroValue)(seqOp, combOp)
serviceAggregated: org.apache.spark.rdd.RDD[(String, (Double, Int))] = ShuffledRDD[61] at aggregateByKey at <console>:26

scala>

scala> // mapValues - Compute average

scala> val serviceAvgSatisfaction = serviceAggregated.mapValues { case (sum, count) => sum / count }
serviceAvgSatisfaction: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[62] at mapValues at <console>:23

scala>

scala> // takeOrdered - Get 3 lowest average satisfaction services

scala> val lowestServices = serviceAvgSatisfaction.takeOrdered(3)(Ordering.by(_._2))
lowestServices: Array[(String, Double)] = Array((Seat comfort,2.810623345806334), (Food and drink,2.8429314593410604), (
On-board service,3.4262571871862737))
```

```
scala> // Print results

scala> println("Top 3 Services with the Lowest Satisfaction:")
Top 3 Services with the Lowest Satisfaction:

scala> lowestServices.foreach { case (service, avgRating) =>
     |   println(f"Service: $service, Average Satisfaction: $avgRating%.2f")
     | }
Service: Seat comfort, Average Satisfaction: 2.81
Service: Food and drink, Average Satisfaction: 2.84
Service: On-board service, Average Satisfaction: 3.43
```

Figure 14: Scala Code of Satisfaction Between Different Service Types

Findings and analysis

The analysis revealed that the three in-flight services with the **lowest average satisfaction** scores were:

- **Seat comfort**: Average Satisfaction = **2.81**
- **Food and drink**: Average Satisfaction = **2.84**
- **On-board service**: Average Satisfaction = **3.43**

These ratings (out of 5) are significantly low in all services, suggesting a consistent pattern of dissatisfaction. Passengers may be experiencing discomfort, limited meal variety, or underwhelming on-board assistance. These insights help the airline focus on improving the specific services that are currently affecting overall passenger satisfaction the most.

Comparison to Overall Satisfaction

When compared to the overall average satisfaction score (3.9 across all services), the sharp drop in these three areas indicates a critical service gap. This divergence suggests that even if passengers are satisfied with certain parts of the travel experience (e.g., check-in process, baggage handling), in-flight discomfort significantly drags down overall impressions.

Recommendations and Next Steps

To boost overall passenger satisfaction, it is recommended that the airline prioritize improvements in the three lowest-rated in-flight services: seat comfort, food and drink, and on-board service. Enhancing these areas is likely to have the most significant impact on customer experience and satisfaction metrics.

4.2.5 Analyzing the Effect of Flight Class on Seat Comfort Satisfaction

Objective

This analysis aims to evaluate the relationship between flight class and passenger satisfaction with seat comfort. Understanding how seat comfort varies by class can help the airline determine whether premium classes are meeting customer expectations and identify areas for comfort improvement.

RDD Transformations and Actions

- Transformations:
  - **map()**: Each record was mapped to a key-value pair consisting of the flight class and its corresponding seat comfort rating.
  - **groupByKey()**: Grouped the seat comfort ratings by flight class so that we could calculate average satisfaction per class.
  - **mapValues()**: Calculated the average seat comfort rating for each flight class by dividing the total by the number of ratings.
- Action:
  - **take(3)**: Retrieved the average seat comfort ratings for the three available flight classes in the dataset.

Scala Code

```
scala> // Step 1: Map flight class to seat comfort rating

scala> val classSeatComfortRDD = rdd.map(row => {
     |   val flightClass = row.getAs[Any]("Class").toString
     |   val seatComfort = row.getAs[Any]("Seat comfort").toString.toDouble
     |   (flightClass, seatComfort)
     | })
classSeatComfortRDD: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[64] at map at <console>:23

scala>

scala> // Step 2: Group by flight class

scala> val groupedByClass = classSeatComfortRDD.groupByKey()
groupedByClass: org.apache.spark.rdd.RDD[(String, Iterable[Double])] = ShuffledRDD[65] at groupByKey at <console>:23

scala>

scala> // Step 3: Compute average seat comfort per class

scala> val avgSeatComfortByClass = groupedByClass.mapValues(ratings => {
     |   val ratingList = ratings.toList
     |   ratingList.sum / ratingList.size
     | })
avgSeatComfortByClass: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[66] at mapValues at <console>:23
scala> // Step 4: Take 3 results (e.g., one for each class type)

scala> val result = avgSeatComfortByClass.take(3)
result: Array[(String, Double)] = Array((2,2.8405782944221944), (3,2.8886138613861387), (1,2.7676175330484045))

scala>

scala> // Step 5: Print the results

scala> println("Seat Comfort Satisfaction by Flight Class:")
Seat Comfort Satisfaction by Flight Class:

scala> result.foreach { case (flightClass, avg) =>
     |   println(f"Flight Class: $flightClass, Average Seat Comfort: $avg%.2f")
     | }
Flight Class: 2, Average Seat Comfort: 2.84
Flight Class: 3, Average Seat Comfort: 2.89
Flight Class: 1, Average Seat Comfort: 2.77
```

Figure 15: Scala Code of Flight Class on Seat Comfort Satisfaction

Findings and analysis

The average seat comfort satisfaction scores by flight class were as follows:

- **Eco (Class 1)**: Average Seat Comfort = **2.84**
- **Business (Class 2)**: Average Seat Comfort = **2.77**
- **Eco Plus (Class 3)**: Average Seat Comfort = **2.89**

Surprisingly, Business class received the lowest seat comfort rating, while Eco Plus received the highest but still considered low. This suggests that seat comfort does not meet expectations in all classes.

Recommendations and Next Steps

Improve All Classes' Comfort by Reevaluating class seating design, spacing, and materials to address potential issues behind the low satisfaction score.

4.2.6 Gender-Based Satisfaction Analysis

Objective

To analyze whether there is a noticeable difference in satisfaction levels between male and female passengers. Understanding gender-based preferences can help the airline tailor experiences more effectively.

RDD Transformations and Actions

- **Transformations**:
  - **map()**: Transformed each record into a key-value pair of (Gender, Satisfaction) after converting gender codes (1 and 2) into meaningful labels ("Male" and "Female").
  - **groupByKey()**: Grouped satisfaction ratings by gender.
  - **mapValues()**: Calculated the average satisfaction score for each gender group by summing all values and dividing by the total count.
- **Action**:
  - **collect()**: Retrieved the final average satisfaction results for display and interpretation.

Scala Code

```
scala> // STEP 1: Convert DataFrame to RDD

scala> val rdd = df.rdd
rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[48] at rdd at <console>:23

scala>

scala> // STEP 2: Map to (Gender Label, Satisfaction Score)

scala> val genderSatisfaction = rdd.map(row => {
     |   val genderCode = row.getAs[Any]("Gender").toString
     |   val gender = genderCode match {
     |     case "1" => "Male"
     |     case "2" => "Female"
     |     case _   => "Unknown"
     |   }
     |   val satisfaction = row.getAs[Any]("satisfaction").toString.toDouble
     |   (gender, satisfaction)
     | })
genderSatisfaction: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[71] at map at <console>:23

scala>

scala> // STEP 3: Group satisfaction values by gender

scala> val grouped = genderSatisfaction.groupByKey()
grouped: org.apache.spark.rdd.RDD[(String, Iterable[Double])] = ShuffledRDD[72] at groupByKey at <console>:23
```

```
scala> // STEP 4: Compute average satisfaction per gender

scala> val avgSatisfactionByGender = grouped.mapValues(values => {
     |   val list = values.toList
     |   list.sum / list.size
     | }).collect()
avgSatisfactionByGender: Array[(String, Double)] = Array((Male,0.6071658837616284), (Female,0.394772542280908))

scala>

scala> // STEP 5: Print the final results

scala> println("Average Satisfaction by Gender:")
Average Satisfaction by Gender:

scala> avgSatisfactionByGender.foreach { case (gender, avg) =>
     |   println(f"$gender: $avg%.2f")
     | }
Male: 0.61
Female: 0.39
```

Figure 16: Scala code of Gender-Based Satisfaction Analysis

Findings and analysis

The average satisfaction scores were:

- **Male**: **0.61**
- **Female**: **0.39**

This result indicates a significant difference in satisfaction levels, with male passengers reporting a notably higher satisfaction score than female passengers. This disparity may reflect differences in service experience, expectations, or communication that should not be overlooked.

Recommendations and Next Steps

- **Investigate the Gap:** Conduct interviews or feedback sessions with female passengers to better understand the root causes of dissatisfaction.
- **Train Staff for Inclusivity:** Implement service training to ensure equal treatment and attention to both male and female passengers.

4.2.7 Loyalty and Satisfaction Correlation

Objective

This analysis aims to explore the relationship between customer loyalty and satisfaction. By examining how satisfaction levels vary between loyal and disloyal customers, the airline can better understand which customer segments are more satisfied and identify opportunities to improve retention.

RDD Transformations and Actions

- Transformations:
  - **map()**: Transformed each record into a pair of (Customer Type, Satisfaction Score), where numeric customer type codes were mapped to meaningful labels such as “Loyal Customer” and “Disloyal Customer.”
  - **reduceByKey()**: Aggregated satisfaction scores by customer type to compute the total satisfaction for each group.
- Actions:
  - **countByKey()**: Counted how many passengers belong to each customer type.
  - **foreach()**: Displayed the average satisfaction for each customer type.

Scala Code

```
scala> // STEP 4: Calculate average satisfaction per customer type

scala> val avgSatisfaction = sums.map { case (custType, total) =>
     |   val count = counts(custType)
     |   val avg = total / count
     |   (custType, avg)
     | }
avgSatisfaction: scala.collection.Map[String,Double] = Map(2 -> 0.20289855072463777, 1 -> 0.5735185290601696)

scala>

scala> // STEP 5: Print results with readable labels

scala> println("Customer Type ? Average Satisfaction")
Customer Type ? Average Satisfaction

scala> avgSatisfaction.foreach { case (custType, avg) =>
     |   val label = custType match {
     |     case "1" => "Loyal Customer"
     |     case "2" => "Disloyal Customer"
     |     case _   => "Unknown"
     |   }
     |   println(f"$label ? Average Satisfaction: $avg%.2f")
     | }
Disloyal Customer ? Average Satisfaction: 0.20
Loyal Customer ? Average Satisfaction: 0.57
```

```
scala> // STEP 1: Map (Customer Type, Satisfaction Score as Double)

scala> val mapped = rdd.map(row => {
     |   val custType = row.getAs[Any]("Customer Type").toString
     |   val satisfaction = row.getAs[Any]("satisfaction").toString.toDouble
     |   (custType, satisfaction)
     | })
mapped: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[86] at map at <console>:23

scala>

scala> // STEP 2: Count number of records per customer type

scala> val counts = mapped.countByKey()
counts: scala.collection.Map[String,Long] = Map(2 -> 4347, 1 -> 17567)

scala>

scala> // STEP 3: Sum satisfaction scores per customer type

scala> val sums = mapped.reduceByKey(_ + _).collectAsMap()
sums: scala.collection.Map[String,Double] = Map(2 -> 882.0, 1 -> 10075.0)
```

Figure 17: Scala code of Loyalty and Satisfaction Correlation

Findings and analysis

The calculated average satisfaction scores by customer loyalty status were:

- **Loyal Customer**: **0.57**
- **Disloyal Customer**: **0.20**

These results reveal a significant disparity between the two customer types. Loyal customers report much higher satisfaction, while disloyal customers are largely dissatisfied. This suggests that customer loyalty may be closely tied to perceived service quality, or that negative experiences are leading to customer churn.

Recommendations and Next Steps

- **Target Disloyal Customers:** Investigate the key pain points for disloyal passengers through follow-up surveys or interviews.
- **Retention Strategy:** Implement loyalty-based incentives such as points, upgrades, or personalized service to boost satisfaction among less loyal customers.

5- SQL operations

1. Satisfaction by Travel Class

Show which class (Economy (1), Business (2), Economy Plus (3)) has the highest satisfaction.

This helps identify whether seat class affects overall passenger satisfaction.

```
scala> spark.sql("""
     | SELECT Class,
     |        COUNT(*) AS Total,
     |        SUM(CASE WHEN satisfaction = 1 THEN 1 ELSE 0 END) AS Satisfied,
     |        ROUND(SUM(CASE WHEN satisfaction = 1 THEN 1 ELSE 0 END) * 100.0 / COUNT(*), 2) AS Satisfaction_Percentage
     | FROM airline
     | GROUP BY Class
     | ORDER BY Satisfaction_Percentage DESC
     | """).show()
+-----+-----+---------+----------------------+
|Class|Total|Satisfied|Satisfaction_Percentage|
+-----+-----+---------+----------------------+
|    1|10061|     6750|                 67.09|
|    3| 1616|      615|                 38.06|
|    2|10237|     3592|                 35.09|
+-----+-----+---------+----------------------+
```

Figure 18: SQL code of Satisfaction by travel class

The result shows that Economy Class has the highest satisfaction rate at 67.09%, followed by Economy Plus at 38.06% and Business Class at 35.09%. This outcome is unexpected, as Business Class typically offers more comfort and premium services. One possible explanation is that Economy passengers have lower expectations and are more easily pleased, while Business Class travelers may be more critical. It may be helpful for the airline to reassess its premium services to better align with customer expectations.

2. Dissatisfaction by Travel Type

Measure if dissatisfaction is more common among personal (1) or business travelers(2). Traveling can influence expectations.

```
scala> spark.sql("""
     | SELECT `Type of Travel`,
     |        COUNT(*) AS Total_Travelers,
     |        SUM(CASE WHEN satisfaction = 0 THEN 1 ELSE 0 END) AS Dissatisfied_Count
     | FROM airline
     | GROUP BY `Type of Travel`
     | """).show()
+--------------+---------------+------------------+
|Type of Travel|Total_Travelers|Dissatisfied_Count|
+--------------+---------------+------------------+
|             1|          15036|              6974|
|             2|           6878|              3983|
+--------------+---------------+------------------+
```

Figure 19: SQL code of Dissatisfaction by travel type

The result shows that 6,974 out of 15,036 personal passengers and 3,983 out of 6,878 business travelers expressed dissatisfaction. But when it comes to the percentage, business travelers appear to be less satisfied overall, which may be because they frequently have more demanding schedules in terms of comfort, quality, and timing. On the other hand, personal travelers may be more relaxed and adaptable, which may result in less dissatisfaction. The reason for travel influences people's perceptions of their experiences, so the airline should support more effectively for business travelers to enhance their overall experience

3. Cleanliness and Satisfaction

Determine how satisfaction changes with varying cleanliness ratings.

```
scala> spark.sql("""
     | SELECT Cleanliness,
     |        COUNT(*) AS Total,
     |        SUM(CASE WHEN satisfaction = 1 THEN 1 ELSE 0 END) AS Satisfied_Count,
     |        ROUND(SUM(CASE WHEN satisfaction = 1 THEN 1 ELSE 0 END) * 100.0 / COUNT(*), 2) AS Satisfaction_Rate
     | FROM airline
     | GROUP BY Cleanliness
     | ORDER BY Cleanliness DESC
     | """).show()
+-----------+-----+---------------+-----------------+
|Cleanliness|Total|Satisfied_Count|Satisfaction_Rate|
+-----------+-----+---------------+-----------------+
|          5| 5941|           4071|            68.52|
|          4| 8156|           4433|            54.35|
|          3| 4211|           1178|            27.97|
|          2| 2338|            825|            35.29|
|          1| 1268|            450|            35.49|
+-----------+-----+---------------+-----------------+
```

Figure 20: SQL code of Cleanliness and satisfaction

The result shows that passenger satisfaction appears to be strongly linked to cleanliness ratings. When cleanliness was rated at the highest level (5) the satisfaction rate was 68.52%, and when it is level 3 the satisfaction rate was 27.97%. Even moderate levels of cleanliness (levels 1 and 2) showed relatively low satisfaction rates of around 35%. These results indicate a positive correlation between cleanliness and passenger satisfaction. Improving cleanliness on board could be an effective way to boost overall customer satisfaction.

4. Most Common Departure Delay

By categorizing delays to "On Time", "Minor", or "Major" delay, we better understand the airline's performance.

```
scala> spark.sql("""
     |   SELECT
     |     CASE
     |       WHEN `Departure Delay in Minutes` = 0 THEN 'On Time'
     |       WHEN `Departure Delay in Minutes` > 0 AND `Departure Delay in Minutes` <= 0.3 THEN 'Minor Delay'
     |       ELSE 'Major Delay'
     |     END AS Delay_Category,
     |     COUNT(*) AS Count
     |   FROM airline
     |   GROUP BY Delay_Category
     | """).show()
+--------------+-----+
|Delay_Category|Count|
+--------------+-----+
|       On Time|12949|
|   Major Delay| 1523|
|   Minor Delay| 7442|
+--------------+-----+
```

Figure 21: SQL code of Most Common Departure Delay

The results show that most passengers reported that their flights departed on time, with 12,949 entries being "on time." 7,442 passengers reported minor delays, while 1,523 passengers reported major delays. Although many passengers may have experienced the same flight, the data still provides a general idea of how passengers perceive delays. The relatively low number of major delay reports suggests that passengers, in general, view the airline's departure timing as reliable. Maintaining this perception can play an important role in building trust and satisfaction among passengers.

5. Rating Distribution: Seat Comfort

Count how many passengers gave each seat comfort score (1 to 5). This distribution reveals whether passengers lean toward low, mid, or high comfort experiences

```
scala> spark.sql("""
     |   SELECT `Seat comfort` AS Rating,
     |          COUNT(*) AS Total_Passengers
     |   FROM airline
     |   WHERE `Seat comfort` > 0
     |   GROUP BY `Seat comfort`
     |   ORDER BY Rating
     | """).show()
+------+----------------+
|Rating|Total_Passengers|
+------+----------------+
|     1|            3600|
|     2|            5015|
|     3|            5103|
|     4|            4747|
|     5|            2733|
+------+----------------+
```

Figure 22: SQL code of Rating Distribution: Seat Comfort

The results show passengers' ratings of seat comfort on a scale of 1 to 5. The most common ratings were 3 and 2, with 5,103 and 5,015 passengers, respectively, indicating that many passengers had a neutral or slightly negative experience. Ratings of 4 and 1 were also slightly different, with 4,747 and 3,600 passengers, while only 2,733 passengers gave the highest rating of 5. This distribution indicates that passengers generally tend to have average or below-average comfort levels, with a smaller number reporting a very comfortable experience. This information highlights an opportunity for the airline to improve seat comfort to enhance the overall passenger experience.

6. Flight Distance and Dissatisfaction

To Identify which flight distance categories, have more dissatisfied passengers.

```
scala> spark.sql("""
     |   SELECT
     |     CASE
     |       WHEN `Flight Distance` < 0.2 THEN 'Short'
     |       WHEN `Flight Distance` < 0.4 THEN 'Medium'
     |       WHEN `Flight Distance` < 0.6 THEN 'Moderate'
     |       ELSE 'Long'
     |     END AS Distance_Category,
     |     COUNT(*) AS Total,
     |     SUM(CASE WHEN satisfaction = 0 THEN 1 ELSE 0 END) AS Dissatisfied,
     |     ROUND(SUM(CASE WHEN satisfaction = 0 THEN 1 ELSE 0 END) * 100.0 / COUNT(*), 2) AS Dissatisfaction_Rate
     |   FROM airline
     |   GROUP BY
     |     CASE
     |       WHEN `Flight Distance` < 0.2 THEN 'Short'
     |       WHEN `Flight Distance` < 0.4 THEN 'Medium'
     |       WHEN `Flight Distance` < 0.6 THEN 'Moderate'
     |       ELSE 'Long'
     |     END
     | """).show()
+-----------------+-----+------------+--------------------+
|Distance_Category|Total|Dissatisfied|Dissatisfaction_Rate|
+-----------------+-----+------------+--------------------+
|           Medium| 8436|        5021|               59.52|
|             Long| 3003|        1201|               39.99|
|            Short| 3902|        1178|               30.19|
|         Moderate| 6573|        3557|               54.12|
+-----------------+-----+------------+--------------------+
```

Figure 23: SQL code of Flight Distance and Dissatisfaction

The results show that dissatisfaction levels vary across flight distance categories. Medium flights recorded the highest dissatisfaction rate at 59.52%, followed by moderate flights at 54.12%. Dissatisfaction rates decreased for long flights at 39.99%, while short flights recorded the lowest dissatisfaction rate at 30.19%. These results indicate that passengers on medium and moderate flights tend to experience lower levels of satisfaction, possibly due to their expectations of greater comfort on short flights, which also lack the amenities typically offered on long flights, so improving service quality on medium and moderate flights could contribute to reducing dissatisfaction in these categories.

7. Delay Statistics by Satisfaction

To Compare average delay times for satisfied vs. dissatisfied passengers. It is a clear way to test if delays are a major cause of dissatisfaction.

```
scala> spark.sql("""
     |   SELECT satisfaction,
     |          COUNT(*) AS Total,
     |          ROUND(AVG(`Departure Delay in Minutes`), 3) AS Avg_Departure_Delay,
     |          ROUND(AVG(`Arrival Delay in Minutes`), 3) AS Avg_Arrival_Delay
     |   FROM airline
     |   GROUP BY satisfaction
     | """).show()
+------------+-----+-------------------+-----------------+
|satisfaction|Total|Avg_Departure_Delay|Avg_Arrival_Delay|
+------------+-----+-------------------+-----------------+
|           1|10957|              0.058|            0.093|
|           0|10957|              0.071|            0.121|
+------------+-----+-------------------+-----------------+
```

Figure 24: SQL code of Delay Statistics by Satisfaction

The results show that satisfied passengers experienced lower average delays compared to dissatisfied ones. Based on normalized delay values, the average departure delay for satisfied passengers was 0.058, and the arrival delay was 0.093. In contrast, dissatisfied passengers had slightly higher values 0.071 for departure and 0.121 for arrival. While the difference is small, it still points to a potential link between delays and overall passenger satisfaction. This suggests that minimizing delays and even minor ones may positively impact on the passenger's travel experience.

6- Machine Learning Operations

The goal of this machine learning task is to develop a predictive model that determines whether a passenger is satisfied or dissatisfied with their airline experience, based on various flight, service-related attributes. This makes the task a binary classification problem. The dataset consists of 120,000 entries collected from Invistico Airlines, we began by implementing and evaluating a Logistic Regression model as a baseline due to its simplicity and interpretability. To improve performance and capture more complex relationships, we then explored tree-based classifiers including Decision Tree, Random Forest, and Gradient Boosted Trees (GBT), to capture more complex relationships and improve predictive performance.

• Logistic Regression

As a starting point, we implemented a Logistic Regression model. It's a widely used machine learning algorithm for binary classification tasks due to its simplicity, efficiency, and interpretability. Making it a logical choice for our baseline. It works by estimating the probability of class membership using a linear combination of the input features. In our case, features like flight distance, seat comfort, customer type, and others provide the input, while satisfaction status is the output. The model is computationally efficient, scales well with large datasets like ours (120,000 rows), and offers valuable insight into how individual features affect predictions. Most importantly, it provides a strong and interpretable baseline for comparison with more complex models such as Decision Tree, Random Forest, and Gradient Boosted Trees (GBT).

Although the dataset was already cleaned and free of missing values, Spark MLlib requires all input features to be numeric and combined into a single vector. Therefore, we performed technical preprocessing steps using Spark's machine learning pipeline, columns such as Gender, Customer Type, Class, and Type of Travel were indexed using StringIndexer, and all relevant columns were merged into a feature vector using VectorAssembler. This processed data was then used to train and test the model.

```
scala> val labelIndexer = new StringIndexer().setInputCol("satisfaction").setOutputCol("label")
labelIndexer: org.apache.spark.ml.feature.StringIndexer = strIdx_ed6ee10977ec

scala> val genderIndexer = new StringIndexer().setInputCol("Gender").setOutputCol("GenderIndex")
genderIndexer: org.apache.spark.ml.feature.StringIndexer = strIdx_fb97980883ed

scala> val classIndexer = new StringIndexer().setInputCol("Class").setOutputCol("ClassIndex")
classIndexer: org.apache.spark.ml.feature.StringIndexer = strIdx_de814c642468

scala> val custTypeIndexer = new StringIndexer().setInputCol("Customer Type").setOutputCol("CustTypeIndex")
custTypeIndexer: org.apache.spark.ml.feature.StringIndexer = strIdx_e7e0ec8a224b

scala> val travelTypeIndexer = new StringIndexer().setInputCol("Type of Travel").setOutputCol("TravelTypeIndex")
travelTypeIndexer: org.apache.spark.ml.feature.StringIndexer = strIdx_ad59ba6cff3f
scala> val assembler = new VectorAssembler()
assembler: org.apache.spark.ml.feature.VectorAssembler = VectorAssembler: uid=vecAssembler_649589053f30, handleInvalid=error

scala>   .setInputCols(Array("Age", "Flight Distance", "Seat comfort", "Cleanliness", "Inflight entertainment",
     |                       "GenderIndex", "ClassIndex", "CustTypeIndex", "TravelTypeIndex"))
res69: assembler.type = VectorAssembler: uid=vecAssembler_649589053f30, handleInvalid=error, numInputCols=9

scala>   .setOutputCol("features")
res70: res69.type = VectorAssembler: uid=vecAssembler_649589053f30, handleInvalid=error, numInputCols=9

scala>

scala> val pipeline = new Pipeline().setStages(Array(
     |   labelIndexer, genderIndexer, classIndexer, custTypeIndexer, travelTypeIndexer, assembler
     | ))
pipeline: org.apache.spark.ml.Pipeline = pipeline_46f960786db6

scala>

scala> val processedData = pipeline.fit(df).transform(df).select("features", "label")
processedData: org.apache.spark.sql.DataFrame = [features: vector, label: double]
```

Figure 25:Indexing categorical columns and assembling feature vectors using Spark ML pipeline

And we evaluate model performance under different training sizes, we applied three train/test splits: 60 training / 40 testing, 70 / 30 and 80 / 20. A fixed seed was applied to each split to ensure reproducibility. The model was trained using LogisticRegression(). fit() and predictions were generated for each split

```
scala> val splits = Seq(
     |   ("60/40", Array(0.6, 0.4)),
     |   ("70/30", Array(0.7, 0.3)),
     |   ("80/20", Array(0.8, 0.2))
     | )
splits: Seq[(String, Array[Double])] = List((60/40,Array(0.6, 0.4)), (70/30,Array(0.7, 0.3)), (80/20,Array(0.8, 0.2)))

scala> import org.apache.spark.ml.classification.LogisticRegressionModel
import org.apache.spark.ml.classification.LogisticRegressionModel

scala>

scala> def computeMetrics(splitName: String, splitRatio: Array[Double]): (String, Double, Double, Double, Double) = {
     |   val Array(trainData, testData) = processedData.randomSplit(splitRatio, seed)
     |
     |   val logistic = new LogisticRegression()
     |   val model = logistic.fit(trainData)
```

Figure 26:Splitting the data into train/test sets and Training the Logistic Regression model.

The model was evaluated using Accuracy**,** Precision**,** Recall**,** and F1 Score. These metrics provide a comprehensive view of how well the model classifies satisfied and dissatisfied passengers.

```
scala> reportDF.show(truncate = false)
+-----+--------+---------+------+--------+
|Split|Accuracy|Precision|Recall|F1 Score|
+-----+--------+---------+------+--------+
|60/40|0.8068  |0.8003   |0.8154|0.8078  |
|70/30|0.8025  |0.7944   |0.8124|0.8033  |
|80/20|0.8043  |0.7943   |0.8148|0.8044  |
+-----+--------+---------+------+--------+
```

Figure 27:classification report obtained for each split

The results show that Logistic Regression provides stable and reliable performance across all data splits. Accuracy ranges from 80.25% to 80.68%, while the F1 Score which balances precision and recall consistently remains above 0.80, indicating solid classification capability. Among the splits, the 60/40 configuration performed the best, achieving the highest accuracy (0.8068) and F1 Score (0.8078).

In the following section, we implement and evaluate Decision Tree, Random Forest, and Gradient Boosted Trees (GBT) classifiers using the same dataset and methodology. These models are expected to capture more complex patterns in the data, and we aim to assess whether they offer a notable improvement over the Logistic Regression baseline in terms of predictive performance.

- ## Decision Tree

We decided to apply a Decision Tree classifier. Using a single Decision Tree allows us to analyse the model's performance in a more interpretable way. This helps us understand how well a simpler model can perform on its own and gives us insight into the trade-off between complexity and accuracy. In figure 28, we used Apache Spark's MLlib to implement the Decision Tree. The required classes were imported from SparkMlclassification. The dataset was split using a 70-30 train-test ratio, and we trained the classifier using DecisionTreeClassifier() with default parameters.

This exact code structure was repeated for the other two splits 60 - 40 and 20 - 80 to observe how the Decision Tree model performs across different train-test ratios.

```
scala> val splitRatio1 = 0.7
splitRatio1: Double = 0.7

scala> val Array(dt_train1, dt_test1) = dfFinal.randomSplit(Array(splitRatio1, 1 - splitRatio1), seed = 42)
dt_train1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
dt_test1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]

scala> val dt1 = new DecisionTreeClassifier().setLabelCol("label").setFeaturesCol("features")
dt1: org.apache.spark.ml.classification.DecisionTreeClassifier = dtc_77361e99486f

scala> val dt_model1 = dt1.fit(dt_train1)
dt_model1: org.apache.spark.ml.classification.DecisionTreeClassificationModel = DecisionTreeClassificationModel: uid=dtc_77361e99
486f, depth=5, numNodes=25, numClasses=2, numFeatures=22

scala> val dt_pred1 = dt_model1.transform(dt_test1)
dt_pred1: org.apache.spark.sql.DataFrame = [features: vector, label: double ... 3 more fields]

scala> val dt_acc1 = evaluatorAcc.evaluate(dt_pred1)
dt_acc1: Double = 0.8765584115745729

scala> val dt_f1_1 = evaluatorF1.evaluate(dt_pred1)
dt_f1_1: Double = 0.8765449883778806

scala> val dt_prec1 = evaluatorPrecision.evaluate(dt_pred1)
dt_prec1: Double = 0.8777329265964893

scala> val dt_rec1 = evaluatorRecall.evaluate(dt_pred1)
dt_rec1: Double = 0.8765584115745728
```

Figure 28:Training and evaluating a Decision Tree classifier in Spark MLlib.

The Decision Tree model demonstrated consistent and reliable performance across the three different train-test splits. In figure 29 both the 70-30 and 60-40 splits, the model achieved nearly identical results, with accuracy and recall at approximately 87.65%, and precision and F1 score remaining in the same range. These results suggest that the model is stable and not significantly impacted by moderate changes in data partitioning. In the 20-80 split, a slight decrease in all evaluation metrics was observed, with accuracy and F1 score dropping to around 86.9%. This indicates that having less training data may slightly affect the model's ability to generalize. Overall, the Decision Tree provided strong results and maintained a good balance between precision and recall across all splits.

```
scala> val dt_results = Seq(
     |   ("70-30", dt_acc1, dt_f1_1, dt_prec1, dt_rec1),
     |   ("60-40", dt_acc2, dt_f1_2, dt_prec2, dt_rec2),
     |   ("20-80", dt_acc3, dt_f1_3, dt_prec3, dt_rec3)
     | ).toDF("Split", "Accuracy", "F1 Score", "Precision", "Recall")
dt_results: org.apache.spark.sql.DataFrame = [Split: string, Accuracy: double ... 3 more fields]

scala>

scala> dt_results.show(truncate = false)
+-----+------------------+------------------+------------------+------------------+
|Split|Accuracy          |F1 Score          |Precision         |Recall            |
+-----+------------------+------------------+------------------+------------------+
|70-30|0.8765584115745729|0.8765449883778806|0.8777329265964893|0.8765584115745728|
|60-40|0.8765588914549653|0.876553155199491 |0.8775304454176742|0.8765588914549653|
|20-80|0.8694756125486605|0.869456600127422 |0.8697135647774142|0.8694756125486605|
+-----+------------------+------------------+------------------+------------------+
```

Figure 29:Classification report obtained for the Decision Tree model for each split.

- **Random Forest**

After analysing the performance of the Decision Tree model, we decided to experiment with Random Forest to further improve our results. While the Decision Tree showed stable and solid performance across different splits, we aimed to reduce possible variance and boost overall accuracy by using an ensemble method. Random Forest combines multiple decision trees and aggregates their outputs, which helps improve generalization and reduce overfitting. In figure 30, we used the 70-30 split and applied the Random Forest classifier. The same structure was reused for the 60-40 and 20-80 splits, following the same preprocessing pipeline and using Spark's MLlib libraries for model training and evaluation. We used RandomForestClassifier from the SparkMlClassification package and evaluated it using accuracy, F1 score, precision, and recall.

```
scala> val splitRatio1 = 0.7
splitRatio1: Double = 0.7

scala> val Array(train1, test1) = dfFinal.randomSplit(Array(splitRatio1, 1 - splitRatio1), seed = 42)
train1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
test1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]

scala> val rf1 = new RandomForestClassifier().setLabelCol("label").setFeaturesCol("features").setNumTrees(100)
rf1: org.apache.spark.ml.classification.RandomForestClassifier = rfc_84ba5be22949

scala> val model1 = rf1.fit(train1)
model1: org.apache.spark.ml.classification.RandomForestClassificationModel = RandomForestClassificationModel: uid=rfc_84ba5be2294
9, numTrees=100, numClasses=2, numFeatures=22

scala> val pred1 = model1.transform(test1)
pred1: org.apache.spark.sql.DataFrame = [features: vector, label: double ... 3 more fields]
```

Figure 30:Training a Random Forest classifier using Spark MLlib.

The results in figure 31 show that the Random Forest model performed consistently well across all splits, with accuracy values approaching 90%, and similarly high scores across all other metrics. Compared to the Decision Tree and the baseline model, this represents a noticeable improvement. The model's performance remained stable regardless of the split ratio used, indicating its robustness and ability to generalize effectively even when the amount of training data varies.

```
scala> val results = Seq(
     |   ("70-30", acc1, f1_1, prec1, rec1),
     |   ("60-40", acc2, f1_2, prec2, rec2),
     |   ("20-80", acc3, f1_3, prec3, rec3)
     | ).toDF("Split", "Accuracy", "F1 Score", "Precision", "Recall")
results: org.apache.spark.sql.DataFrame = [Split: string, Accuracy: double ... 3 more fields]

scala>

scala> results.show(truncate = false)
+-----+------------------+------------------+------------------+------------------+
|Split|Accuracy          |F1 Score          |Precision         |Recall            |
+-----+------------------+------------------+------------------+------------------+
|70-30|0.8953363090657226|0.8953456819815313|0.8958499286288086|0.8953363090657227|
|60-40|0.8957274826789838|0.8957378651096779|0.8961582109230681|0.8957274826789838|
|20-80|0.898786352186856 |0.898747339968311 |0.8994393003802805|0.898786352186856 |
+-----+------------------+------------------+------------------+------------------+
```

Figure 31:Classification report obtained for the Random Forest model for each split.

- **Gradient Boosted**

Following the results of the previous models, we decided to explore Gradient Boosted Trees (GBT) as a third classification technique. GBT is known for its high predictive power and its ability to focus on correcting errors made by earlier trees in the sequence. Unlike Random Forests, which use multiple trees in parallel, GBT builds trees sequentially, optimizing for performance at each step. As shown in figure 32, we started with the 70-30 train-test split and used Spark's GBTClassifier from the SparkMLClassification package. The model was trained with 50 iterations (maxIter=50), using the same preprocessed dataset and feature column structure as in previous models. After training, we evaluated the model's performance using accuracy, precision, recall, and F1 score. The same procedure was applied to the 60-40 and 20-80 splits to maintain consistency across evaluations.

```
scala> import org.apache.spark.ml.classification.GBTClassifier
import org.apache.spark.ml.classification.GBTClassifier

scala> val Array(gbt_train1, gbt_test1) = dfFinal.randomSplit(Array(0.7, 0.3), seed = 42)
gbt_train1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]
gbt_test1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [features: vector, label: double]

scala> val gbt1 = new GBTClassifier().setLabelCol("label").setFeaturesCol("features").setMaxIter(50)
gbt1: org.apache.spark.ml.classification.GBTClassifier = gbtc_ba0739974485

scala> val gbt_model1 = gbt1.fit(gbt_train1)
gbt_model1: org.apache.spark.ml.classification.GBTClassificationModel = GBTClassificationModel: uid = gbtc_ba0739974485, numTrees
=50, numClasses=2, numFeatures=22

scala> val gbt_pred1 = gbt_model1.transform(gbt_test1)
gbt_pred1: org.apache.spark.sql.DataFrame = [features: vector, label: double ... 3 more fields]

scala> val gbt_acc1 = evaluatorAcc.evaluate(gbt_pred1)
gbt_acc1: Double = 0.9295059258119132

scala> val gbt_f1_1 = evaluatorF1.evaluate(gbt_pred1)
gbt_f1_1: Double = 0.9295110077217046

scala> val gbt_prec1 = evaluatorPrecision.evaluate(gbt_pred1)
gbt_prec1: Double = 0.9295458658356732

scala> val gbt_rec1 = evaluatorRecall.evaluate(gbt_pred1)
gbt_rec1: Double = 0.9295059258119132
```

Figure 32:Training and evaluating a Gradient Boosted Trees (GBT)classifier in Spark MLlib.

The results in figure 33 show that GBT achieved the highest performance among all models tested. All evaluation metrics exceeded 91% across every split. Notably, the model maintained nearly identical results across 70-30, 60-40, and 20-80 ratios, demonstrating exceptional consistency and robustness. This confirms that Gradient Boosted Trees not only handle the classification task effectively but also perform reliably regardless of the data partitioning strategy.

```
scala>

scala> val gbt_results = Seq(
     |   ("70-30", gbt_acc1, gbt_f1_1, gbt_prec1, gbt_rec1),
     |   ("60-40", gbt_acc2, gbt_f1_2, gbt_prec2, gbt_rec2),
     |   ("20-80", gbt_acc3, gbt_f1_3, gbt_prec3, gbt_rec3)
     | ).toDF("Split", "Accuracy", "F1 Score", "Precision", "Recall")
gbt_results: org.apache.spark.sql.DataFrame = [Split: string, Accuracy: double ... 3 more fields]

scala>

scala> gbt_results.show(truncate = false)
+-----+------------------+------------------+------------------+------------------+
|Split|Accuracy          |F1 Score          |Precision         |Recall            |
+-----+------------------+------------------+------------------+------------------+
|70-30|0.9295059258119132|0.9295110077217046|0.9295458658356732|0.9295059258119132|
|60-40|0.9262124711316397|0.9262156316106765|0.9262277111165749|0.9262124711316397|
|20-80|0.9193954659949622|0.9193942296842917|0.9194146890648556|0.9193954659949622|
+-----+------------------+------------------+------------------+------------------+
```

Figure 33:Classification report obtained for the Gradient Boosted Trees (GBT) model for each split.

- **Final Model Comparison Summary**

The results shown in figure 34 present a clear side-by-side comparison of the three classification models (Random Forest, Decision Tree, and Gradient Boosted Trees (GBT)) across all data splits. Among the three, GBT consistently achieved the highest performance, with all metrics exceeding 91%, regardless of the split ratio. Random Forest also performed strongly, with accuracy and other metrics hovering around 89.5% to 89.9%, showing it is a reliable and well-generalizing model. Meanwhile, the Decision Tree achieved slightly lower scores across all splits, with accuracy and F1 score around 87.6%, indicating good but less optimal performance compared to the ensemble methods.

Overall, the GBT model proved to be the most effective approach for this classification task, providing both high accuracy and consistent results across varying data partitions. These findings support the benefit of using more advanced ensemble techniques when aiming for optimal performance in predictive modeling.

```
scala> finalResults.show(truncate = false)
+-------------+-----+------------+------------+-------------+----------+
|Model        |Split|Accuracy (%)|F1 Score (%)|Precision (%)|Recall (%)|
+-------------+-----+------------+------------+-------------+----------+
|Random Forest|70-30|89.53       |89.53       |89.58        |89.53     |
|Random Forest|60-40|89.57       |89.57       |89.62        |89.57     |
|Random Forest|20-80|89.88       |89.87       |89.94        |89.88     |
|Decision Tree|70-30|87.66       |87.65       |87.77        |87.66     |
|Decision Tree|60-40|87.66       |87.66       |87.75        |87.66     |
|Decision Tree|20-80|86.95       |86.95       |86.97        |86.95     |
|GBT          |70-30|92.95       |92.95       |92.95        |92.95     |
|GBT          |60-40|92.62       |92.62       |92.62        |92.62     |
|GBT          |20-80|91.94       |91.94       |91.94        |91.94     |
+-------------+-----+------------+------------+-------------+----------+
```

Figure 34:comparison of the three classification models.

7- Conclusion

In this report, we explored the process of data preprocessing to ensure the dataset is clean, structured, and suitable for analysis. We addressed missing values by replacing them with the mean, removed outliers using the Z-score method to enhance data reliability, and applied transformation techniques such as Encoding, Normalization using Min-Max Scaling, and Discretization to standardize the data.

Additionally, we performed data reduction to optimize processing without losing essential information. Instead of removing columns, we reduced the number of rows while maintaining all key attributes. Stratified Sampling was applied to preserve the class distribution, ensuring a representative dataset. Furthermore, we addressed class imbalance by applying undersampling, reducing the majority class to match the minority class. As the project progressed, we incorporated both SQL and RDD operations into Scala to analyze and process data. These operations contributed to a more comprehensive analysis by enabling efficient data processing and large-scale processing. Both approaches supported our goal of validating preprocessing steps and gaining a deeper understanding of the dataset.

Most importantly, the report also included a machine learning phase, where we trained and evaluated predictive models. We began with Logistic Regression to establish a strong and interpretable performance baseline. We then implemented a Decision Tree classifier, which allowed us to model non-linear relationships while maintaining a level of interpretability. This was followed by the Random Forest classifier, an ensemble-based method designed to reduce overfitting and enhance accuracy by combining the predictions of multiple decision trees. Lastly, we incorporated Gradient Boosted Trees (GBT) a powerful boosting algorithm that builds trees sequentially to correct the errors of previous ones, often leading to higher accuracy and better generalization. All models were tested using multiple train/test splits and evaluated using metrics such as accuracy, precision, recall, and F1 score.

By systematically preparing the data through these preprocessing and modeling steps, we have enhanced its consistency, accuracy, and usability ultimately optimizing it for efficient processing, meaningful analysis, and predictive modeling with improved performance.

8- References

[1] "What is CSAT? (+ how to measure it)," *Zendesk*, Jan. 31, 2025. https://www.zendesk.com/blog/customer-satisfaction-score/#

[2] Pew Research Center, "Pew Research Center | Numbers, Facts and Trends Shaping Your world," *Pew Research Center*, Oct. 23, 2024. https://www.pewresearch.org/