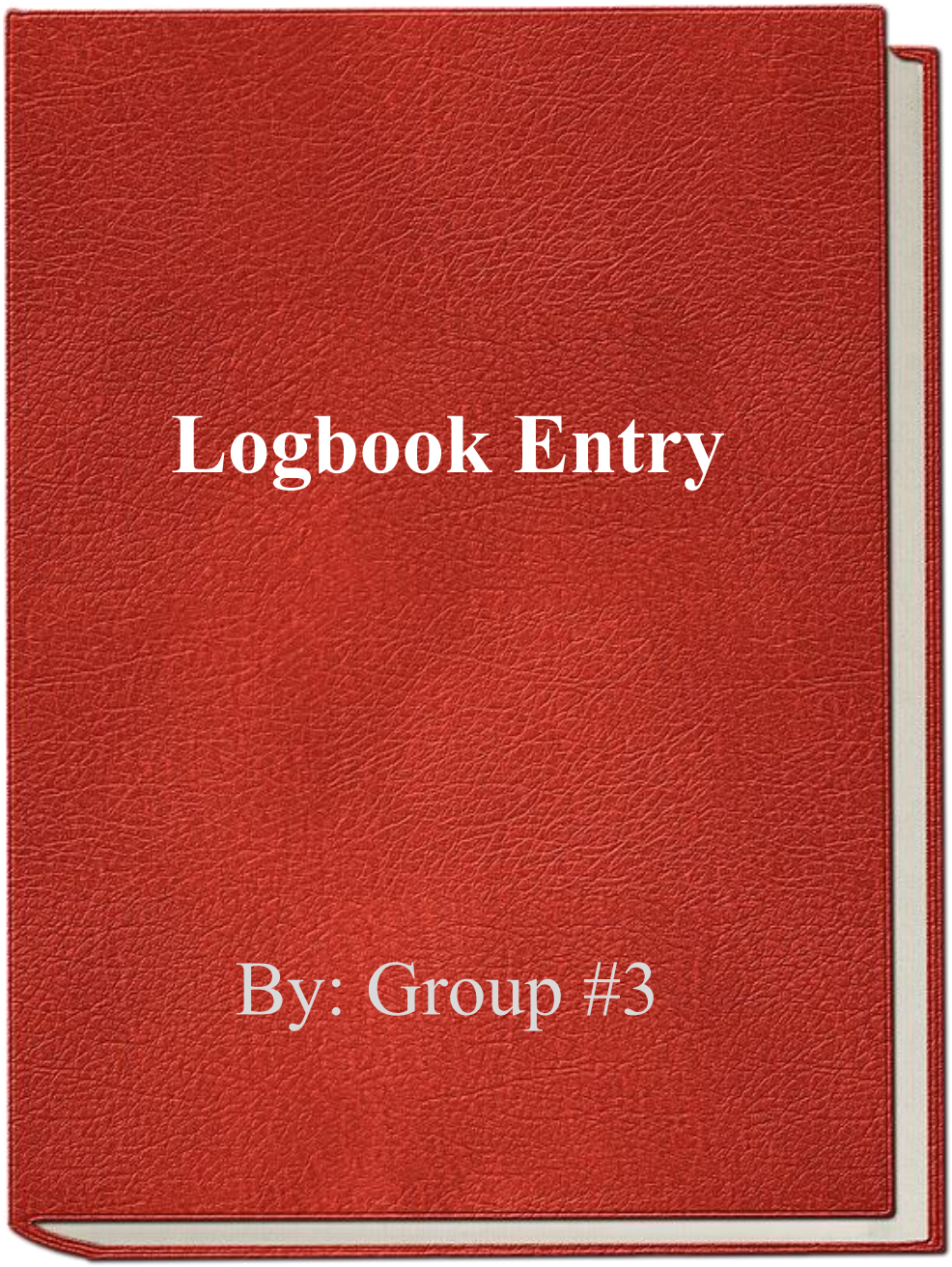
King Saud University
College of Computer and Information Sciences
Department of Information Technology

IT362 – First semester 1446



Data Collection Process:

12 September 2024:
Our initial attempt involved using the SkyScanner API, but we discovered that access is restricted to businesses, excluding individuals, researchers, and students. We then attempted manual web scraping using Python on Almusafer, but this resulted in an empty JSON file, indicating we were blocked. We also searched for an Almusafer API but found none available.

13 September 2024:
After several unsuccessful attempts, we discovered SerpAPI, an AI tool capable of scraping Google Flights. However, the tool scrapes data one day at a time, which would be highly time-consuming for our needs.

14 September 2024:
We found a workaround by combining SerpAPI generated code with Python to scrape Google Flights for multiple dates at once. While this worked, it had a limit of 100 records, which was insufficient for our project as we needed data for a full year across many destinations. Due to this limitation and the associated cost, we set this solution aside to explore other options.

The code:

```python
from serpapi import GoogleSearch
import pandas as pd

# Define the search parameters
params = {
  "api_key": "abcf3b01167d5acfe77a37e6b7c8a998f6a7257c3bd373964a884d87368690b8",  # Replace with your SerpAPI key
  "engine": "google_flights",
  "hl": "en",
  "gl": "sa",
  "departure_id": "CDG",
  "arrival_id": "AUS",
  "outbound_date": "2024-09-13",
  "return_date": "2024-09-19",
  "currency": "USD",
  "travel_class": "1"
}

# Perform the search
search = GoogleSearch(params)
results = search.get_dict()

# Extract flight data
flights = []
for result in results.get('flights_results', []):
    flight = {
        "airline": result.get('airline', ''),
        "departure_time": result.get('departure_time', ''),
        "arrival_time": result.get('arrival_time', ''),
        "price": result.get('price', ''),
        "currency": result.get('currency', ''),
        "link": result.get('link', '')
    }
    flights.append(flight)

# Convert to DataFrame and save as CSV
df = pd.DataFrame(flights)
df.to_csv('flights_data.csv', index=False)
print("Data saved to flights_data.csv")
```

16 September 2024:
We attempted to use FlightAPI but found it too complex and confusing to implement.

17 September 2024:
We explored two other options: : Oxylabs and RapidAPI. However, both were too complicated to understand, so we opted not to pursue them.

18 September 2024:
We implemented a script from Crawlbase to scrape Google Flights, but encountered issues with the arrival city data, which either returned random text, the departure city, or "N/A" from an if statement in the code. We had to modify the selectors by inspecting the webpage to extract the correct variables, everything worked but the arrival city.

19 September 2024:
Using the same Crawlbase code, we tried applying a driver method to scrape data for the entire year. Unfortunately, the output did not display symbols correctly, hindering the process.

20 September 2024:
We resolved the issue with the arrival city, which now displays correctly. However, the task of scraping multiple cities in Saudi Arabia to various destinations for an entire year would still be extremely time-consuming if done manually.

23 September 2024:
We adapted the code from 20 September to include a loop that iterates through dates and web pages, automating the scraping process. It finally worked successfully! ^_^

24 September 2024: (Date of collection)
After reviewing our summarized research papers, we decided to add two new attributes: “arrival_time” and “departure_time” to help identify potential patterns later on. We officially began data collection.

25 September 2024:
All the JSON files collected were merged into a single dataset and converted into a .csv file.

Link for raw data: Raw Data

Final code used:

In [...]

```python
# Import necessary libraries
from bs4 import BeautifulSoup
import requests
import json
from datetime import timedelta, date

# Function to scrape listing elements from Google Flights
def scrape_listings(soup):
    return soup.select('li.pIav2d')

# Function to scrape company name from a flight listing
def scrape_company_name(listing):
    airline_element = listing.select_one('div.Ir0Voe div.sSHqwe')
    return airline_element.text.strip() if airline_element else "N/A"

# Function to scrape flight duration from a flight listing
def scrape_flight_duration(listing):
    duration_element = listing.select_one('div.AdWm1c.gvkrdb')
    return duration_element.text.strip() if duration_element else "N/A"

# Function to scrape price from a flight listing
def scrape_price(listing):
    price_element = listing.select_one('span[data-gs][aria-label*="riyals"]')
    return price_element.text.strip().replace('\u00A0', ' ') if price_element else "N/A"

# Function to scrape flight stops from a flight listing
def scrape_flight_stops(listing):
    stops_element = listing.select_one('div.EfT7Ae span.ogfYpf')
    return stops_element.text.strip() if stops_element else "Non-stop"

# Function to scrape departure city from a flight listing
def scrape_departure_city(listing):
    departure_city_element = listing.select_one('div.G2WY5c.sSHqwe.ogfYpf.tPgKwe div')
    return departure_city_element.text.strip() if departure_city_element else "N/A"

# Function to scrape arrival city from a flight listing
def scrape_arrival_city(listing):
    arrival_city_element = listing.select_one('div.c8rWCd.sSHqwe.ogfYpf.tPgKwe div')
    return arrival_city_element.text.strip() if arrival_city_element else "N/A"

# Function to scrape departure time from a flight listing
def scrape_departure_time(listing):
    departure_time_element = listing.select_one('div.wtdjmc.YMlIz.ogfYpf.tPgKwe')
    return departure_time_element.text.strip() if departure_time_element else "N/A"

# Function to scrape arrival time from a flight listing
def scrape_arrival_time(listing):
    arrival_time_element = listing.select_one('div.XWcVob.YMlIz.ogfYpf.tPgKwe')
    return arrival_time_element.text.strip() if arrival_time_element else "N/A"

# Function to generate date range
def daterange(start_date, end_date):
    for n in range(int((end_date - start_date).days) + 1):
        yield start_date + timedelta(n)

# Main function
def main():
    # Define start and end dates for scraping
    start_date = date(2024, 9, 24)
    end_date = date(2025, 9, 24)

    # Loop through dates and scrape data for each date
    flight_data = []

    for single_date in daterange(start_date, end_date):
        travel_date = single_date.strftime("%Y-%m-%d")

        # Use the provided URL without modifying it with dates
        # Example link with dynamic date
        url = f'https://www.google.com/travel/flights/search?tfs=CBwQAhojEgoyMDI0LTA5LTI1agcIARIDQUJUcgwIAhIIL20vMGRsbV9AAUgBcAGCAQsI__________AZgBAg&departure_date={travel_date}'



        # Make a request to the updated Google Flights URL and parse HTML
        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'html.parser')

        # Scrape flight listings
        listings = scrape_listings(soup)

        # Iterate through each listing and extract flight information
        for listing in listings:
            company_name = scrape_company_name(listing)
            flight_duration = scrape_flight_duration(listing)
            price = scrape_price(listing)
            stops = scrape_flight_stops(listing)
            departure_city = scrape_departure_city(listing)
            arrival_city = scrape_arrival_city(listing)
            departure_time = scrape_departure_time(listing)
            arrival_time = scrape_arrival_time(listing)

            # Use the travel_date instead of scraping for the departure date
            flight_info = {
                'company_name': company_name,
                'flight_duration': flight_duration,
                'price': price,
                'departure_date': travel_date,  # Set the departure date to the travel_date
                'departure_time': departure_time,
                'arrival_time': arrival_time,
                'stops': stops,
                'departure_city': departure_city,
                'arrival_city': arrival_city
            }

            flight_data.append(flight_info)

        print(f"Scraping data for {travel_date} completed")

    # Save results to a JSON file
    with open('Al Bahah_ToRiyadh.json', 'w') as json_file:
        json.dump(flight_data, json_file, indent=4)

if __name__ == "__main__":
    main()
```

```
Scraping data for 2024-09-24 completed
Scraping data for 2024-09-25 completed
Scraping data for 2024-09-26 completed
```

Exploratory Data Analysis (EDA):

This exploratory data analysis (EDA) focuses on understanding patterns in flight-related data, specifically exploring the relationships between flight duration, price, and airline companies. The goal of this analysis is to identify correlations that may exist between these variables and derive insights that could help us understand how various factors influence flight prices and durations. This section provides a detailed description of the EDA process, including data preprocessing, statistical analysis, and visualization techniques.

1. Primary Data

Tools and Libraries Used:

For this analysis, the following Python libraries were employed to ensure efficient data processing, visualization, and statistical computations:

- **pandas**: Utilized for data manipulation, cleaning, and transformation.
- **matplotlib and seaborn:** Employed for creating visualizations that help in understanding the distributions and correlations within the dataset.
- **numpy:** Used for performing numerical operations, including calculations and data transformations.

These libraries are standard tools for conducting data analysis, offering a wide range of functions that support various stages of the analytical process.

Data Cleaning and Preprocessing:

The dataset initially contained variables such as flight duration (represented as strings), price, and categorical information, including airline companies and city names. Several steps were taken to ensure the dataset was clean and ready for analysis:

1. **Flight Duration Conversion:** The flight duration was initially represented as a string (e.g., "2hr 30min"). A function was used to convert these values into total minutes to allow for numerical analysis.
2. **Handling Missing Values:** Rows containing missing or null values in the flight_duration or price columns were removed to ensure the accuracy of the correlation analysis. This is an important step, as missing data could introduce bias into the results or lead to invalid conclusions.
3. **Categorical Variable Encoding:** Categorical columns such as company_name and arrival_city were factorized into numerical values to enable correlation calculations. Factorization replaces each unique category with a numeric code, facilitating quantitative analysis.

These preprocessing steps ensured that the dataset was in the correct format for further statistical and graphical analysis.

Dataset shape:
Our dataset contains 505,504 rows, which represent individual records, and 9 columns, which represent different features or attributes of the data.

Exploratory Data Analysis (EDA):

The primary objective of the EDA phase was to explore the distribution of key variables and the relationships between them.

Number of Flights by Airline:

A histogram was created to analyze the distribution of flights across different airlines. The results showed that "Saudia" is the most frequently used airline by a significant margin, followed by "Flynas". (Figure 1)

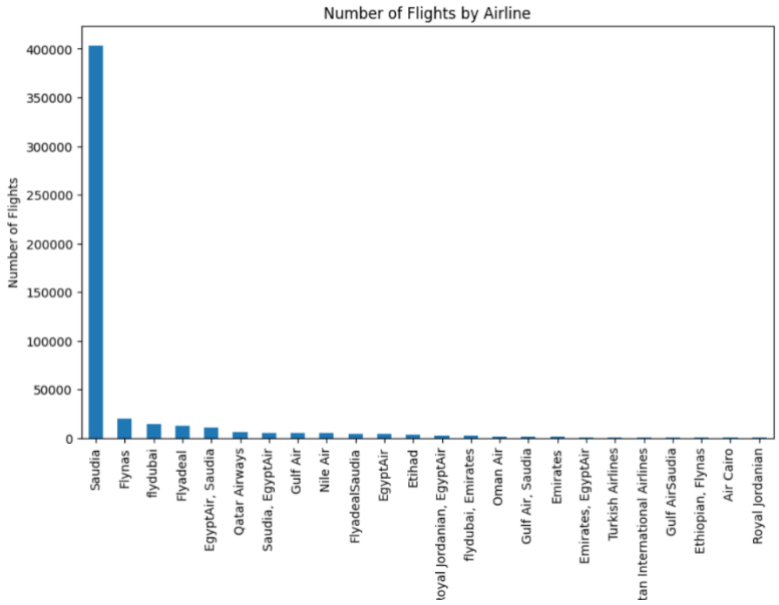


Figure 1

Flight Duration Distribution:

A histogram of the flight durations was generated to examine the distribution. The plot revealed that most flights have a duration between 50 and 250 minutes, and the most common is 1 hr 50 min, with a smaller number of outlier flights having longer durations. The data appeared slightly right-skewed, indicating the presence of a few long-haul flights that last significantly longer than the majority. (Figure 2)

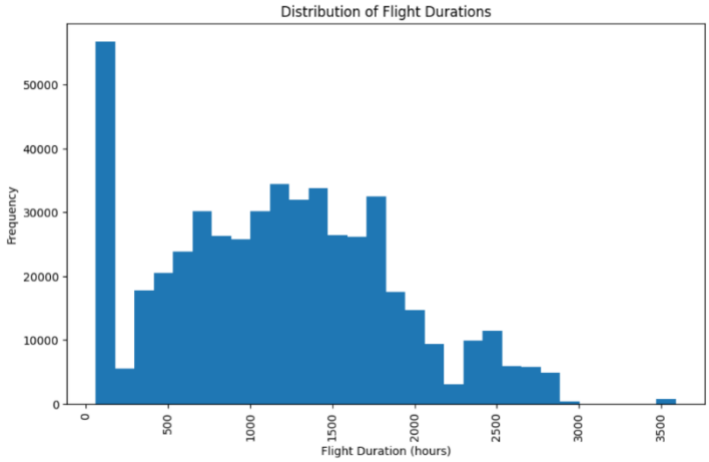


Figure 2

Price Distribution:

Similarly, a histogram of flight prices was created, showing that the majority of flights are priced between 500 SAR and 1500 SAR with the most frequent being 1018 SAR. The price distribution was also right-skewed, with a few flights priced significantly higher. This distribution suggests that while most flights are moderately priced, there are premium flights with much higher costs, likely reflecting additional factors such as time of booking, or demand. (Figure 3)

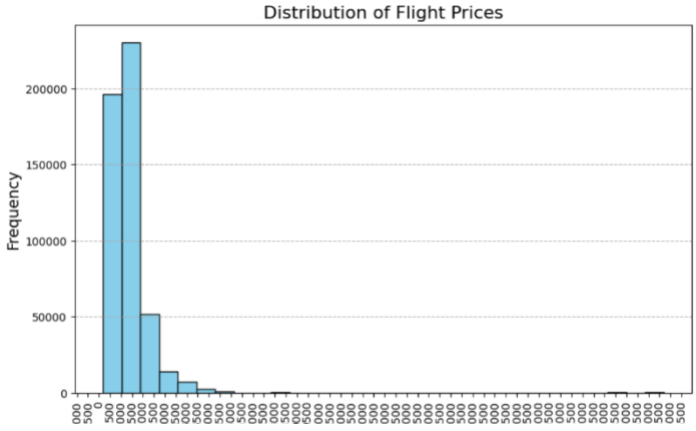


Figure 3

Number of Flights by Departure Date:

A bar chart was created to visualize the number of flights based on the departure date. The chart revealed a clear pattern in flight scheduling over time with the highest frequency being between October and December 2024. This suggests that airlines schedule more flights during periods of high demand, which aligns with the assumption that holiday periods see greater travel volumes. (Figure 4)

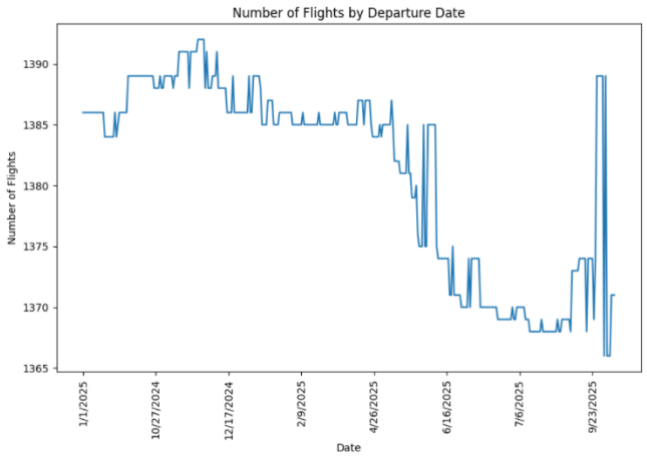


Figure 4

Most Common Departure Times:

An analysis of the most common departure times revealed distinct patterns, with peak times occurring during the early morning (9 AM) and early evening (4 PM - 9 PM). These times correspond to the periods when travelers are likely heading to work or returning home, or when business travelers are catching flights for early meetings. (Figure 5)

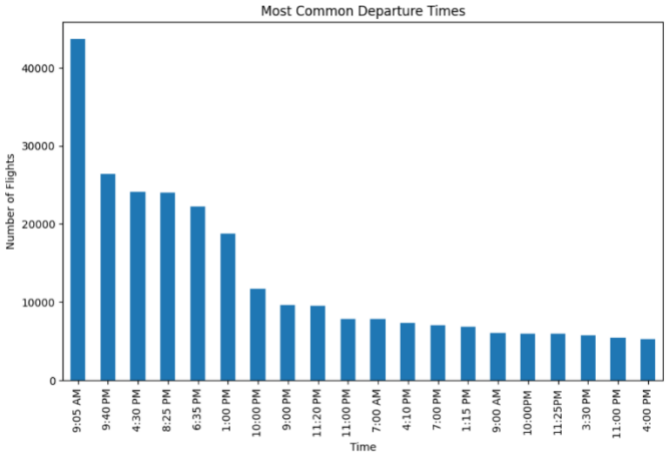


Figure 5

Most Common Arrival Times:

Similar to departure times, the most common arrival times tended to cluster around early morning and evening hours. The distribution of arrival times reflects the common practice of flights being scheduled for early morning or evening landings, allowing travelers to maximize their days. (Figure 6)

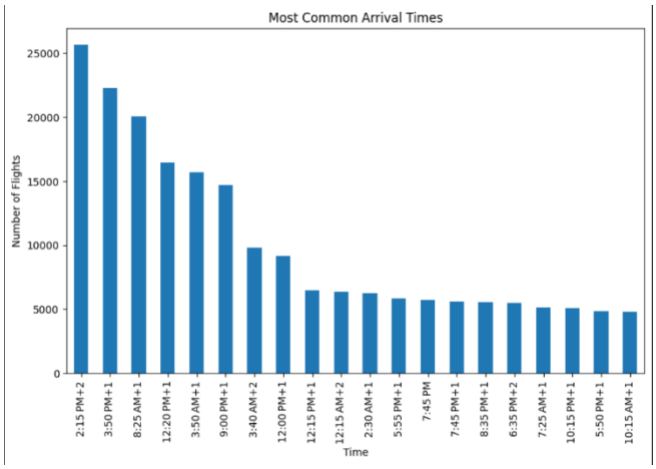


Figure 6

Number of Flights by Stops:

Flights were categorized based on the number of stops (non-stop, one-stop, two-stop flights, etc.). The majority of flights in the dataset were 2 stops. (Figure 7)

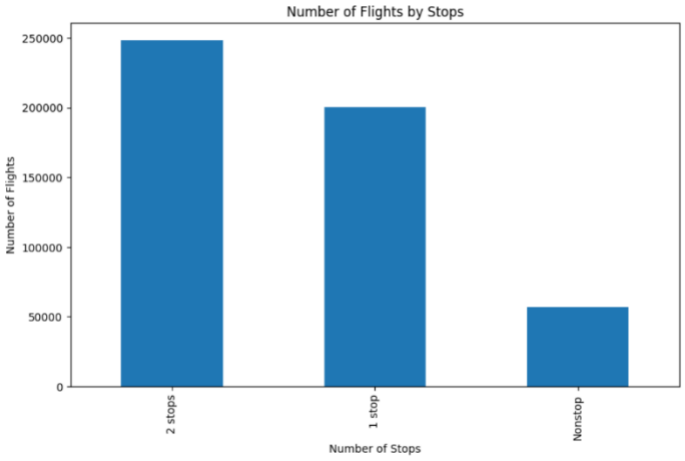


Figure 7

Flights by Departure City:

The distribution of flights across departure cities was visualized using a bar chart. Larger cities such as RUH, EAM, and JED naturally had the highest number of departing flights. (Figure 8)

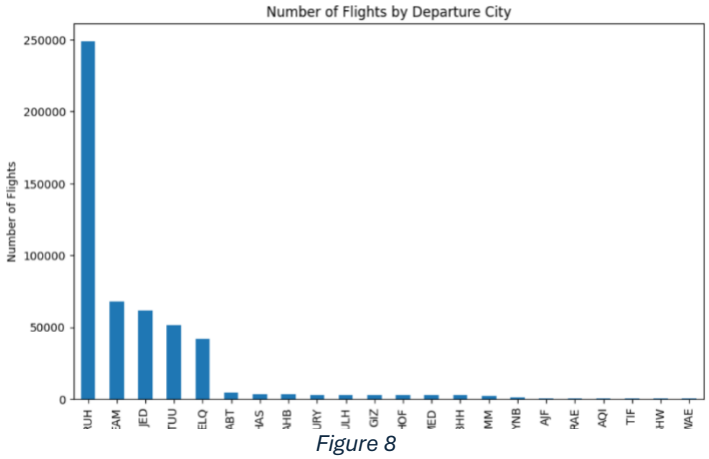


Figure 8

Flights by Arrival City:

Similarly, the number of flights by arrival city followed a similar pattern, with RUH receiving the bulk of flights. This is consistent with the notion that large areas attract more traffic due to both business and leisure travel. (Figure 9)

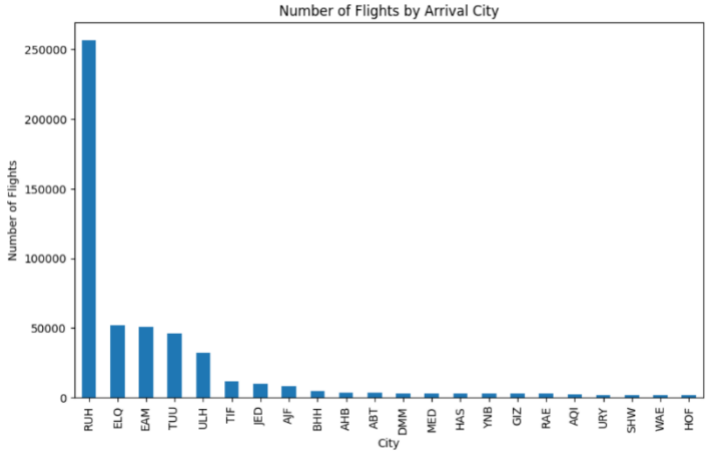


Figure 9

Cross-tabulation between Departure City and Arrival City:

A cross-tabulation was performed to examine the relationship between departure cities and arrival cities. The result highlighted specific patterns in city pairs, indicating the frequency with which flights connect particular cities. This type of analysis helps identify major flight routes between the destinations. The heatmap shows a dark red cell at the intersection of RUH and EAM, indicating a substantial number of flights on this route. (Figure 10)

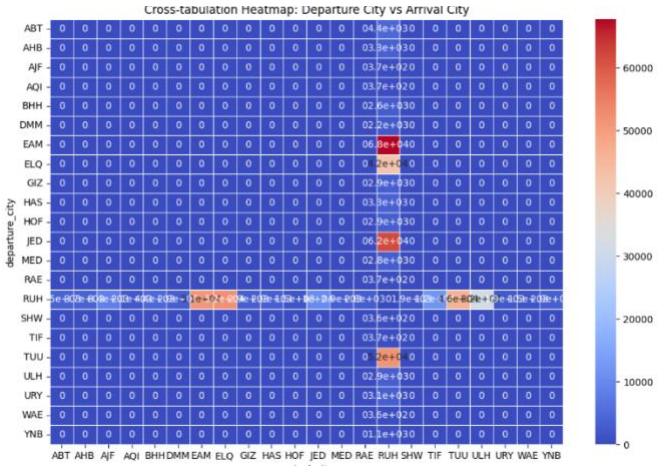


Figure 10

Correlation Analysis:

To further explore the relationships between variables, we conducted a correlation analysis.

Before conducting correlation analysis, the categorical variables (such as airline companies and city names) were encoded numerically. This process, known as factorization, ensures that these categorical data points can be used in numerical calculations. Each company or city was assigned a unique numeric code to facilitate the correlation analysis.

Correlation Matrix and Heatmap:

A correlation matrix was computed to examine the relationships between flight duration, price, and airline company. The following key observations were made:

Flight Duration and Price:

A weak positive correlation (0.073) was observed between flight duration and price. This suggests that, on average, longer flights tend to be slightly more expensive, though the relationship is weak. Other factors, such as airline pricing strategies or market competition, may play a more significant role in determining flight prices. (Figure 11)

Price and Airline Company:

A moderate positive correlation (0.38) was identified between flight price and airline company. This indicates that certain airlines tend to charge more for their flights, which may reflect differences in brand value, or operating costs. The analysis suggests that the airline is a more important determinant of price than flight duration. (Figure 11)

Flight Duration and Airline Company:

A weak negative correlation (-0.11) was found between flight duration and airline company. This implies that the duration of a flight is not strongly tied to the specific airline, indicating that airlines operate flights of varying durations without any specific pattern.

The correlation heatmap provided a visual representation of these relationships, highlighting the weak associations between the variables. (Figure 11)

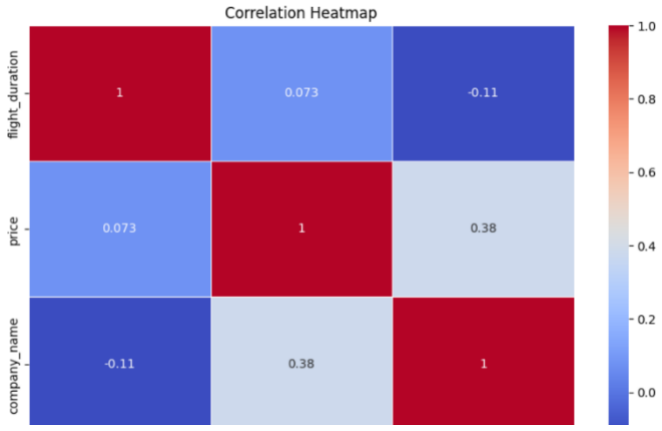


Figure 11

2. Secondary Data

Tools and Libraries Used:

For this analysis, the following Python libraries were employed to ensure efficient data processing, visualization, and statistical computations:

- **pandas**: Utilized for data manipulation, cleaning, and transformation.
- **matplotlib and seaborn:** Employed for creating visualizations that help in understanding the distributions and correlations within the dataset.
- **numpy:** Used for performing numerical operations, including calculations and data transformations.

These libraries are standard tools for conducting data analysis, offering a wide range of functions that support various stages of the analytical process.

Data Cleaning and Preprocessing:

The currency was converted from US dollars $ to SAR In order to facilitate comparisons with our primary dataset.

Dataset shape:

Our dataset contains 317,260 rows, which represent individual records, and 11 columns, which represent different features or attributes of the data.

Exploratory Data Analysis (EDA):

The primary objective of the EDA phase was to explore the distribution of key variables and the relationships between them.

Price distribution:

We plotted the distribution of flight prices using a histogram displaying the frequency of flight prices across specified bins. This visualization further emphasizes the distribution of prices, showing how many flights fall within different price ranges. Both plots serve to highlight the variations in flight prices within the dataset, aiding in understanding pricing patterns. (Figure 12)

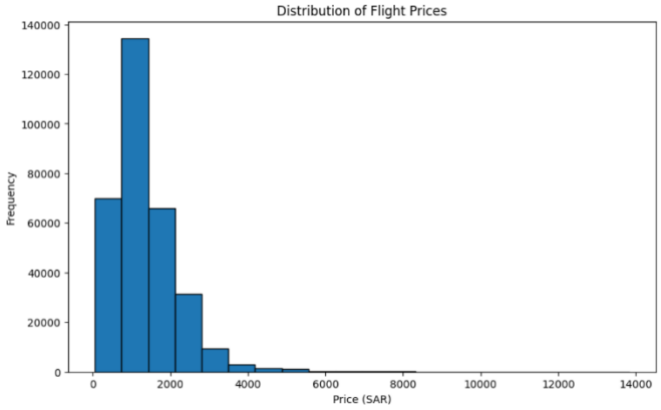


Figure 12

Number of stops distribution:

We visualized the distribution of flights using a bar chart based on the number of stops. It has helped to understand how many flights operate with zero, one, two, or more stops, providing insights into flight options available to travelers. The most frequent is flights with 1 stop. (Figure 13)

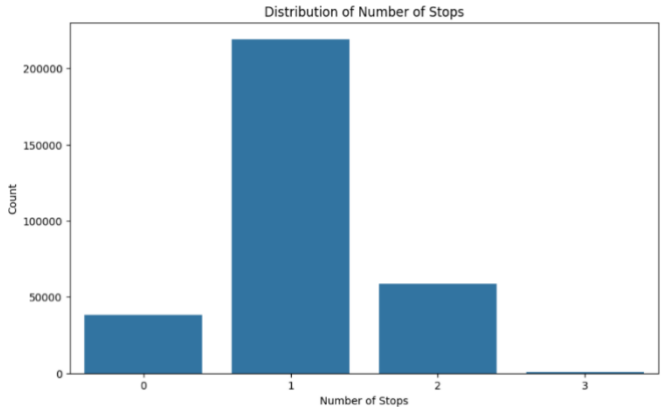


Figure 13

Prices trend over time:

We implemented a line plot depicts the trend of flight prices over time based on the departure date. It helps identify any fluctuations or patterns in pricing, allowing for a better understanding of how prices vary with departure dates. Notably, prices between July and August 2024 were the highest, indicating a peak in demand during that period. (Figure 14)

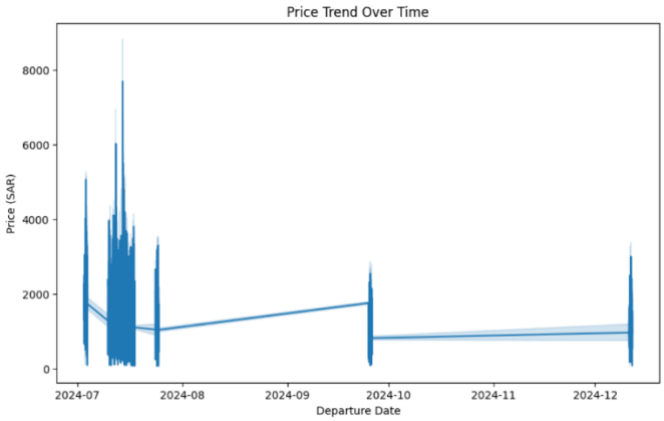


Figure 14

Flight lands next day graph:

We implemented a bar chart to illustrate the frequency of flights landing the next day, with '1' indicating a yes and '0' a no. The data reveals that most flights do not land the next day, highlighting the distribution of overnight travel in the dataset. (Figure 15)

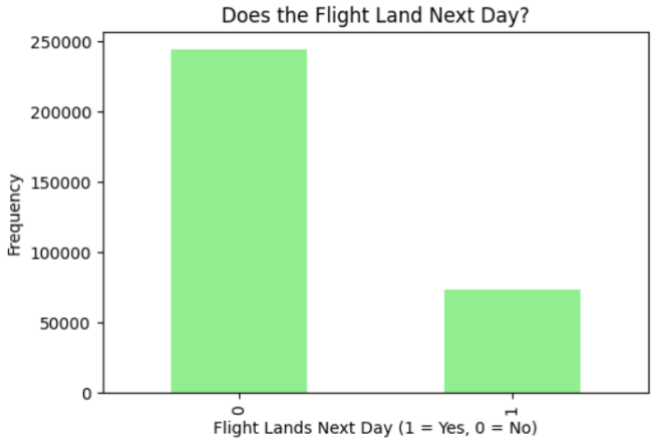


Figure 15

Airline distribution:

We implemented a bar chart illustrates the distribution of airlines in the dataset, The results showed that "United Airlines" is the most frequently used airline by a significant margin, followed by "Alaska Airlines". (Figure 16)

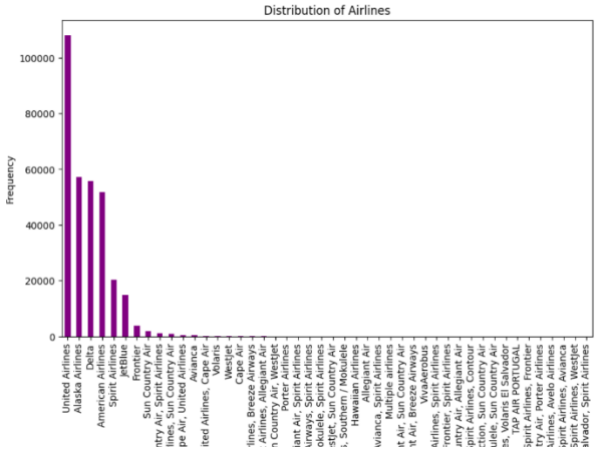


Figure 16

Correlation Analysis:

To further explore the relationships between variables, we conducted a correlation analysis.

Before conducting correlation analysis, the categorical variables (such as airline companies and city names) were encoded numerically. This process, known as factorization, ensures that these categorical data points can be used in numerical calculations. Each company or city was assigned a unique numeric code to facilitate the correlation analysis.

Number of stops and Price:

A moderate positive correlation (0.36) was identified between the number of stops and airline company. indicating that flights with more stops tend to be more expensive. (Figure 17)

Price and Airline Company:

A weak positive correlation (0.23) between airline company and price indicates that prices may vary slightly based on the airline. However, this relationship is weak, implying that other factors, such as service quality and operational costs, likely have a more significant impact on flight pricing. (Figure 17)

Number of stops and Airline Company:

A correlation analysis value of 0.23 between the number of stops and airline company indicates a weak positive correlation. This suggests that there is a slight tendency for the number of stops to vary across different airlines. However, this weak relationship implies that the number of stops is not a strong predictor of which airline is operating a flight, indicating that airlines have diverse routing practices that are influenced by factors other than the airline itself. The correlation heatmap visually illustrates these weak associations between the variables. (Figure 17)

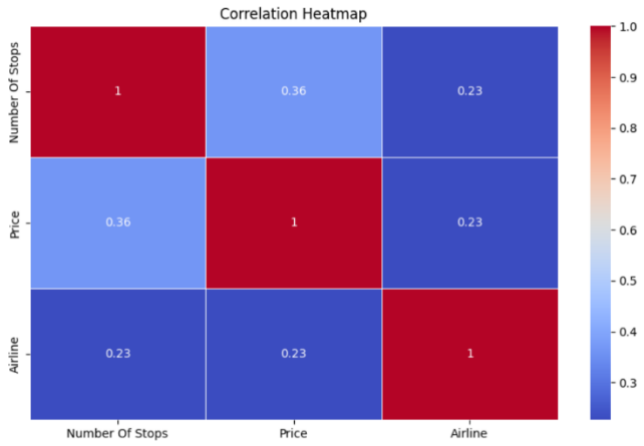


Figure 17

Flight Lands Next Day VS Price:

We implemented a scatter to visualize the relationship between whether a flight lands the next day (binary variable) and its price. Showing if overnight flights tend to be priced differently compared to same-day arrivals. The distribution shows that there are price variations for both scenarios (landing the same day or the next day), but it seems that flights that land the next day may have slightly higher prices concentrated in certain ranges, suggesting a potential impact of overnight flights on pricing. (Figure 18)

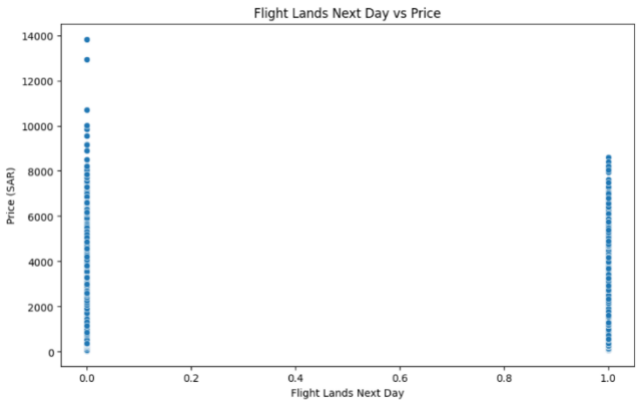


Figure 18

Identifying Outlier:

We implemented a box plot to visually identify outliers in the numerical columns, particularly focusing on the "Price" variable. The boxplot effectively highlights the distribution of flight prices, indicating the presence of several significant outliers above the upper whisker of the plot. Most prices are concentrated around a lower range, with a long tail of high-value prices representing outliers.

The thick box in the plot shows the interquartile range (IQR), which encompasses the middle 50% of the data, while the whiskers extend to 1.5 times the IQR. Any points outside of this range are plotted as individual circles and are considered potential outliers. In this case, flight prices beyond approximately 2,500 units are clearly outliers, with some flights reaching prices as high as 14,000. (Figure 19)

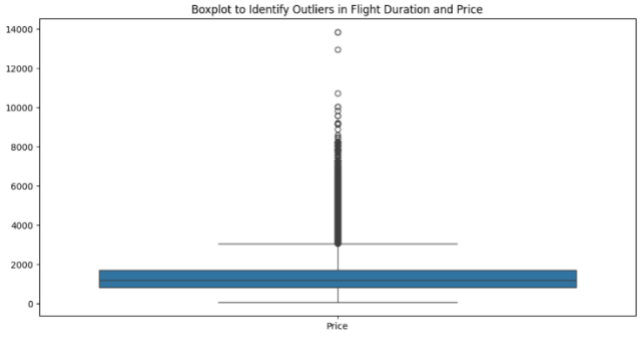


Figure 19

Data Processing and Cleaning:

1. Primary Data

Step 1: Import Required Libraries

To begin the data processing and cleaning phase, essential libraries were imported to facilitate various tasks. The pandas library was imported for data manipulation and analysis, providing powerful data structures like DataFrames. The numpy library was included for numerical operations, offering support for large, multi-dimensional arrays and matrices. Additionally, the re library was imported for handling regular expressions, which are useful for string manipulation and pattern matching. These libraries were crucial for efficiently managing and processing the dataset.

```python
import pandas as pd
import numpy as np
import re
```

Step 2: Load the Dataset

The next step involved loading the dataset into a pandas DataFrame for further processing. This was necessary to bring the data into a structured format that allows for efficient manipulation and analysis. The dataset was loaded from a CSV file using the read_csv function from the pandas library. This function reads the data into a DataFrame, which is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

```python
# Load the dataset
df = pd.read_csv('FlightsData.csv')
```

Step 3: Handling Missing Values

After the dataset was loaded, we identified missing values. Missing data can be problematic because it can distort analyses or lead to inaccurate models, so it's important to address these issues early. We used the **isnull().sum()** function to check each column and see how many missing values were present. This gave us a clear view of where data might be incomplete, setting the stage for deciding how to deal with these gaps.

```
missing_data = df.isnull().sum()
print(missing_data)
```

✓ 0.1s Python

```
company_name          0
flight_duration       0
price              1007
departure_date        0
departure_time        0
arrival_time          0
stops                 0
departure_city        0
arrival_city          0
dtype: int64
```

Step 4: Data Cleaning

During the data cleaning process, several key steps were implemented to ensure the dataset was properly formatted. First, the **'price'** column was cleaned by removing the "**SAR**" prefix and commas, allowing the values to be converted to floats for numerical analysis. Missing prices were filled using the median price of similar flights based on **company_name** and **departure_city**, providing reasonable estimates for incomplete data.

Next, the **departure_time** and **arrival_time** columns were sanitized to remove unusual symbols, ensuring consistency in the time data. Finally, the stops column was standardized by removing the text " **stop**" or " **stops**," leaving only numeric values. Missing or **NaN** values in this column were replaced with 0, indicating direct flights. These steps were essential for preparing the dataset for further analysis.

```
# Cleaning the 'price' column: remove 'SAR' and commas, convert to float
df['price'] = df['price'].replace({'SAR ': '', ',': ''}, regex=True).astype(float)
```

✓ 0.9s Python

```
# Check for missing prices
missing_prices = df['price'].isnull().sum()
print("Number of missing prices:", missing_prices)

# Fill missing prices using the median of similar flights
# For example, we can group by 'company_name' and 'departure_city'
median_prices = df.groupby(['company_name', 'departure_city'])['price'].median()

# Fill missing values in the 'price' column
for index, row in df[df['price'].isnull()].iterrows():
    # Get the median price for the group of similar flights
    median_price = median_prices.get((row['company_name'], row['departure_city']), np.nan)
    # Fill the missing price if median_price is available
    if pd.notnull(median_price):
        df.at[index, 'price'] = median_price

# Check if there are any missing prices left
missing_prices_after = df['price'].isnull().sum()
print("Number of missing prices after filling:", missing_prices_after)
```

✓ 0.1s Python

```python
# Clean the 'departure_time' and 'arrival_time' columns to remove weird symbols
df['departure_time'] = df['departure_time'].str.replace(r'[\u200B-\u200D\u202F\u00A0]', '', regex=True)
df['arrival_time'] = df['arrival_time'].str.replace(r'[\u2008-\u200D\u202F\u00A0]', '', regex=True)
```

✓ 0.4s Python

```python
# Ensure all values are treated as strings, then use regex to replace ' stop' or ' stops' with an empty string
df['stops'] = df['stops'].astype(str).str.replace(r'\s*stops?', '', regex=True).str.strip()

# Convert the cleaned 'stops' column to numeric
df['stops'] = pd.to_numeric(df['stops'], errors='coerce')

# Convert the numeric column to integers (if not NaN)
df['stops'] = df['stops'].fillna(0).astype(int)  # Replace NaN with 0 and convert to int
```

✓ 0.5s Python

Step 5: Data processing

In this step several critical data processing tasks were performed to enhance the dataset's usability. Flight durations were converted to total minutes for standardization. Duplicate rows were identified and removed to maintain data integrity. The "**Arrival Time**" and "**Departure Time**" columns were transformed into a 24-hour format, and unnecessary suffixes were removed for consistency. A new column, "**Flight Lands Next Day**," was added to indicate if a flight arrives the next day based on time comparisons, along with an "**Arrival Date**" column created from the "Departure Date" for accurate tracking. Key columns like "**Price**" and "**Flight Duration**" were converted to float for consistency. Finally, the cleaned dataset was saved to a new CSV file, significantly improving its integrity and readiness for analysis.

```python
# Convert flight durations to total minutes
def convert_duration_to_minutes(duration_str):
    if pd.isnull(duration_str) or duration_str.strip() == "":
        return np.nan  # Return NaN for null or empty strings

    # Using a more flexible regex pattern
    match = re.match(r'(?:(\d+)\s*hour(?:s)?|(\d+)\s*hr(?:s)?)?\s*(?:(\d+)\s*minute(?:s)?|(\d+)\s*min(?:s)?)?', duration_str.strip(), re.IGNORECASE)

    if match is None:
        print(f"Unmatched duration string: '{duration_str}'")  # Debugging output
        return np.nan  # If format is unexpected, return NaN

    # Handle the cases where only hours are present
    hours = int(match.group(1)) * 60 if match.group(1) else (int(match.group(2)) * 60 if match.group(2) else 0)

    # Handle minutes: if minutes are present, get them; otherwise, set to 0
    minutes = int(match.group(3)) if match.group(3) else (int(match.group(4)) if match.group(4) else 0)

    return hours + minutes  # Total minutes

# Example of converting the 'flight_duration' column
df['flight_duration'] = df['flight_duration'].apply(convert_duration_to_minutes)
```

✓ 1.0s Python

```python
duplicate_rows = df.duplicated()

# To view the duplicate rows
df[duplicate_rows]
```

✓ 0.1s Python

```python
# Count the number of duplicate rows
num_duplicates = df.duplicated().sum()

# Print the number of duplicates
print("Number of duplicate rows:", num_duplicates)
```

✓ 0.1s Python

```python
# Dropping duplicates (if any)
df = df.drop_duplicates(keep='first')
```

✓ 0.1s Python

```python
# Count the number of duplicate rows in the cleaned dataset
num_duplicates_cleaned = df.duplicated().sum()

# Print the number of duplicates in the cleaned dataset
print("Number of duplicate rows in the cleaned data:", num_duplicates_cleaned)
```

✓ 0.1s Python

```python
# Load the dataset
df = pd.read_csv('FlightsDataCleanedPrimary.csv')

# Removing the '+1', '+2', etc., from the 'Arrival Time' column
df['Arrival Time'] = df['Arrival Time'].apply(lambda x: re.sub(r'\+\d+', '', x))


# Convert 'Departure Time' and 'Arrival Time' to 24-hour format
df['Departure Time'] = pd.to_datetime(df['Departure Time'], format='%I:%M%p', errors='coerce').dt.time
df['Arrival Time'] = pd.to_datetime(df['Arrival Time'], format='%I:%M%p', errors='coerce').dt.time

# Checking the first few rows to verify the changes
df[['Departure Time', 'Arrival Time']].head()
```

✓ 0.8s Python

```python
# Adding a new column 'Flight Lands Next Day' with binary values based on the condition
df['Flight Lands Next Day'] = df.apply(
    lambda row: 1 if row['Arrival Time'] and row['Departure Time'] and row['Arrival Time'].hour < row['Departure Time'].hour else 0,
    axis=1
)

# Checking the updated dataset with the new column
df[['Departure Time', 'Arrival Time', 'Flight Lands Next Day']].head()
```

✓ 3.0s Python

```python
# Adding a new column 'Flight Lands Next Day' with binary values based on the condition
df['Flight Lands Next Day'] = df.apply(
    lambda row: 1 if row['Arrival Time'] and row['Departure Time'] and row['Arrival Time'].hour < row['Departure Time'].hour else 0,
    axis=1
)

# Checking the updated dataset with the new column
df[['Departure Time', 'Arrival Time', 'Flight Lands Next Day']].head()
```

✓ 3.0s Python

```python
# Change the data types of the specified columns in the Primary dataset
df['Price'] = df['Price'].astype('float64')
df['Flight Duration'] = df['Flight Duration'].astype('float64')

# Verify the changes
df.dtypes
```

✓ 0.0s Python

```python
# Save the cleaned data to a new CSV file
df.to_csv('FlightsDataCleaned.csv', index=False)
```

✓ 1.7s Python

Step 6: Save the cleaned data to a new CSV file

We the updated dataset was saved to a new CSV file named **Primary_Data_updated.csv**. This step was crucial to ensure that all changes made during the data processing and cleaning stages were preserved for future analysis. By saving the dataset, users can easily access the cleaned data, which is now free from duplicates and properly formatted.

```python
# Saving the updated dataset to a new CSV file after the changes have been made
updated_file_path = 'Primary_Data_updated.csv'
df.to_csv(updated_file_path, index=False)

updated_file_path
```

2.Secondary data

Step 1: Import Required Libraries

To begin the data processing and cleaning phase, essential libraries were imported to facilitate various tasks. The pandas library was imported for data manipulation and analysis, providing powerful data structures like DataFrames. The numpy library was included for numerical operations, offering support for large, multi-dimensional arrays and matrices. Additionally, the re library was imported for handling regular expressions, which are useful for string manipulation and pattern matching. These libraries were crucial for efficiently managing and processing the dataset.

```python
import pandas as pd
import numpy as np
import re
```

Python

Step 2: Load the Dataset

The next step involved loading the dataset into a pandas DataFrame for further processing. This was necessary to bring the data into a structured format that allows for efficient manipulation and analysis. The dataset was loaded from a CSV file using the read_csv function from the pandas library. This function reads the data into a DataFrame, which is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

```python
# Load the dataset
df = pd.read_csv('flights.csv')
```

[3] Python

Step 3: Handling Missing Values

After the dataset was loaded, we identified missing values. Missing data can be problematic because it can distort analyses or lead to inaccurate models, so it's important to address these issues early. We used the isnull().sum() function to check each column and see how many missing values were present. This gave us a clear view of where data might be incomplete, setting the stage for deciding how to deal with these gaps. The missing values in the Route column were removed, as they were deemed irrelevant for our analysis.

```python
missing_data = df.isnull().sum()
print(missing_data)
```

[4] Python

```
Searched Date              0
Departure Date             0
Arrival Date               0
Flight Lands Next Day      0
Departure Airport          0
Arrival Airport            0
Number Of Stops            0
Route                  38397
Airline                    0
Cabin                      0
Price                      0
dtype: int64
```

Step 4: Data Cleaning

Data cleaning involved converting prices from USD to SAR, splitting the 'Departure Date' and 'Time', and addressing missing values to ensure consistency and accuracy in analysis. The Price column was cleaned by removing dollar signs and commas to allow proper numerical operations, and then converted to a numeric format using pandas' replace and astype functions. Prices were converted to SAR using a predefined exchange rate for consistent financial comparison across records. To maintain focus on a specific segment of interest, we filtered the DataFrame to include only rows where the 'Cabin' is 'Economy', as this is the class we are primarily analyzing. The 'Departure Airport' and 'Arrival Airport' columns were renamed to 'Departure City' and 'Arrival City' for clarity and to match terminology in the primary dataset. A 'Flight Duration' column was added to facilitate comparison with the primary dataset, ensuring consistency in key metrics. Additionally, the 'Departure Time' and 'Arrival Time' columns were reformatted to 24-hour format for standardization and easier analysis of time-based variables.

Convert the currency from $ to SAR

[9]:
```
# Define the conversion rate from USD to SAR
usd_to_sar_rate = 3.75

# Remove dollar signs and commas from the 'Price' column and convert it to numeric
df['Price'] = df['Price'].replace('[\$,]', '', regex=True).astype(float)

# Convert prices from USD to SAR
df['Price'] = df['Price'] * usd_to_sar_rate

# Display the first few rows to verify the changes
print(df[['Price']].head())

print("Prices have been successfully converted from USD to SAR.")
```

```
    Price
0  311.25
1  375.00
2  292.50
3  375.00
4  555.00
Prices have been successfully converted from USD to SAR.
```

Changing Arrival Time, Departure Time data type format to 24 hour

[23]:
```
# Replacing '1900-01-01' with NaN Departure Time
df['Arrival Time'] = df['Arrival Time'].replace('1900-01-01', pd.NA)
df['Departure Time'] = df['Departure Time'].replace('1900-01-01', pd.NA)

# Converting the 'Arrival Time' to a datetime object, only considering the time
df['Arrival Time'] = pd.to_datetime(df['Arrival Time'], errors='coerce').dt.time
df['Departure Time'] = pd.to_datetime(df['Departure Time'], errors='coerce').dt.time

# Checking the updated 'Arrival Time' column
# Displaying the first few rows of only the 'Departure Time' and 'Arrival Time' columns
df[['Departure Time', 'Arrival Time','Flight Lands Next Day']].head()


# The dataset is updated in place without creating a new file
```

[23]:

	Departure Time	Arrival Time	Flight Lands Next Day
0	23:48:00	15:03:00	1
1	09:34:00	19:43:00	0
2	23:48:00	15:03:00	1
3	08:30:00	19:37:00	0
4	05:00:00	15:20:00	0

Add 'Flight Duration' Coulmn

```
import pandas as pd
```

```
# Read the CSV files into dataframes
df = pd.read_csv('flightsCleanedSecondary.csv')

# Convert the "Departure Time" and "Arrival Time" columns to datetime, including date and time
df['Departure Time'] = pd.to_datetime(df['Departure Time'], format='%Y-%m-%d %H:%M:%S')
df['Arrival Time'] = pd.to_datetime(df['Arrival Time'], format='%Y-%m-%d %H:%M:%S')

# Calculate the Flight Duration in minutes and rename the column to 'Flight Duration'
df['Flight Duration'] = (df['Arrival Time'] - df['Departure Time']).dt.total_seconds() / 60

# Adjust for flights landing the next day
df.loc[df['Flight Duration'] < 0, 'Flight Duration'] += 24 * 60

# The df now contains the updated "Flight Duration" column in memory, without saving to a new file.
print("Dataset updated with 'Flight Duration' column.")
```

```
Dataset updated with 'Flight Duration' column.
```

Step 5: Data processing

Data processing involved identifying and handling duplicate rows, as well as converting date and time columns to appropriate formats. Initially, duplicate rows were identified using the duplicated() function from the pandas library. The number of duplicate rows was counted and printed to assess the extent of duplication. Any duplicates were then dropped using the drop_duplicates() method, ensuring that only the first occurrence was kept. The cleaned dataset was checked again to confirm the removal of duplicates. Next, the Departure Date and Arrival Date columns were converted to datetime format to retain both date and time information. New columns for Departure Time and Arrival Time were created by extracting the time part from the datetime columns. The original date columns were then updated to keep only the date part. This was done using the to_datetime and dt accessor methods from the pandas library.

[11]:
```
# Count the number of duplicate rows
num_duplicates = df.duplicated().sum()

# Print the number of duplicates
print("Number of duplicate rows:", num_duplicates)
```

```
Number of duplicate rows: 155186
```

[12]:
```
# Dropping duplicates (if any)
df = df.drop_duplicates(keep='first')
```

In this step, we removed all duplicate rows from the dataset, but retained the first occurrence of each duplicate. The keep='first' argument ensures that the first instance of any duplicate is kept, while subsequent duplicates are removed.

[13]:
```
# Count the number of duplicate rows in the cleaned dataset
num_duplicates_cleaned = df.duplicated().sum()

# Print the number of duplicates in the cleaned dataset
print("Number of duplicate rows in the cleaned data:", num_duplicates_cleaned)
```

```
Number of duplicate rows in the cleaned data: 0
```

Step 6: Save the cleaned data to a new CSV file

We the updated dataset was saved to a new CSV file named **FINALSecondaryDataCleaned.csv**. This step was crucial to ensure that all changes made during the data processing and cleaning stages were preserved for future analysis. By saving the dataset, users can easily access the cleaned data, which is now free from duplicates and properly formatted.

[24]:
```
# Saving the updated dataset to a new CSV file after the changes have been made
updated_file_path = 'FINALSecondaryDataCleaned..csv'
df.to_csv(updated_file_path, index=False)

updated_file_path
```

Link to cleaned and processed datasets: Structured Data Files