CIS 4130

Professor Richard Holowczak

Jenny Lin

jenny.lin10@baruchmail.cuny.edu

Milestone 1

Flight Prices Prediction

Description:
This dataset contains information about purchasable flight tickets available on Expedia between April 16, 2022, and October 5, 2022. The dataset focuses on flights to and from several major airports, including ATL, DFW, DEN, ORD, LAX, CLT,MIA, JFK, EWR, SFO, DTW, BOS, PHL, LGA, IAD, and OAK. The dataset has columns relating to information about each flight, including flight date, total fare, if flight is non-stop, refundable, etc.

URL/location: https://www.kaggle.com/datasets/dilwong/flightprices?resource=download

Data set attributes:
legId: Unique identifier for each flight.
searchDate: Date on which the entry was taken from Expedia.
flightDate: Date of the flight.
startingAirport: IATA code of the departure airport.
destinationAirport: IATA code of the arrival airport.
fareBasisCode: Fare basis code for the flight.
travelDuration: Total travel duration (hours and minutes).
elapsedDays: Number of elapsed days (usually 0).
isBasicEconomy: Boolean indicating if the ticket is for basic economy.
isRefundable: Boolean indicating if the ticket is refundable.
isNonStop: Boolean indicating if the flight is non-stop.
baseFare: Base price of the ticket (in USD).
totalFare: Total price of the ticket (including taxes and fees).
seatsRemaining: Number of remaining seats.
totalTravelDistance: Total travel distance in miles (may contain missing data).
segmentsDepartureTimeEpochSeconds: Departure time (Unix time) for each leg.
segmentsDepartureTimeRaw: Departure time in ISO 8601 format for each leg.
segmentsArrivalTimeEpochSeconds: Arrival time (Unix time) for each leg.
segmentsArrivalTimeRaw: Arrival time in ISO 8601 format for each leg.
segmentsArrivalAirportCode: Arrival airport code for each leg.
segmentsDepartureAirportCode: Departure airport code for each leg.
segmentsAirlineName: Name of the airline for each leg.
segmentsAirlineCode: Airline code for each leg.
segmentsEquipmentDescription: Description of the airplane for each leg.
segmentsDurationInSeconds: Duration of the flight (in seconds) for each leg.
segmentsDistance: Distance traveled (in miles) for each leg.

segmentsCabinCode: Cabin class for each leg (e.g., coach).

Predictive task:
The main task is to predict total fare of a flight (**totalFare** column) based on other flight attributes. The goal is to identify how features such as the duration, travel distance, ticket type (basic, economy, refundable), departure/arrival airports, duration, and number of remaining seats influence a flight price ticket.

Model Selection:
I will use Linear Regression to predict the totalFare variable. This model is suited to predict continuous numerical variables, which is appropriate for forecasting the total price of a ticket. Using this model, I will evaluate how dfiferent features contribute to price changes and how much these features contribute to price changes. I aim to determine these features using domain knowledge of flights, correlation analysis, and using data visualization techniques to explore how variables affect prices.

Milestone 2

Data Acquistion

In this project, I obtained my Kaggle API token to download the Flight Prices dataset onto my bucket my-bigdatatech-project-jl via the Linux Command line.

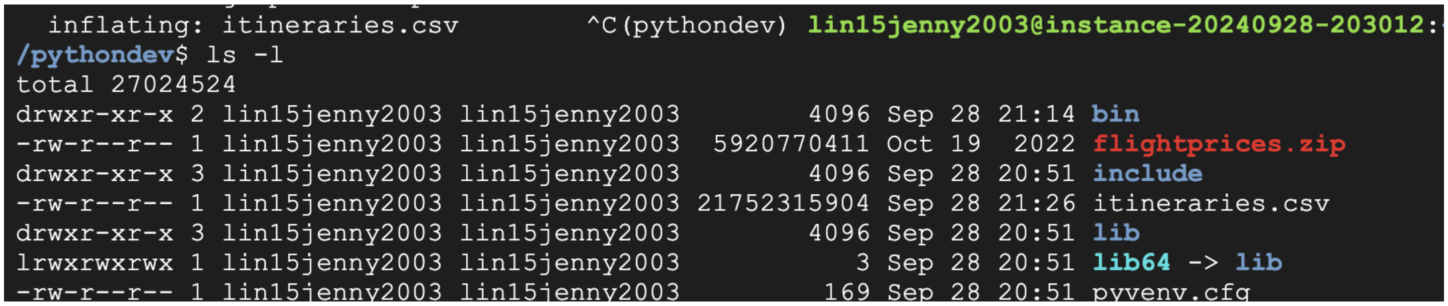
1. First, I created and started a compute engine VM, then connected to the Linux instance with secure shell, setted up the API token, and setted up Python environment.

2. Then, I ran this code (copied API command from dataset) to download the dataset:
kaggle datasets download -d dilwong/flightprices

3. Used this command to install zip utility:
Sudo apt install zip

4. Extract dataset:
Unzip flightprices.zip

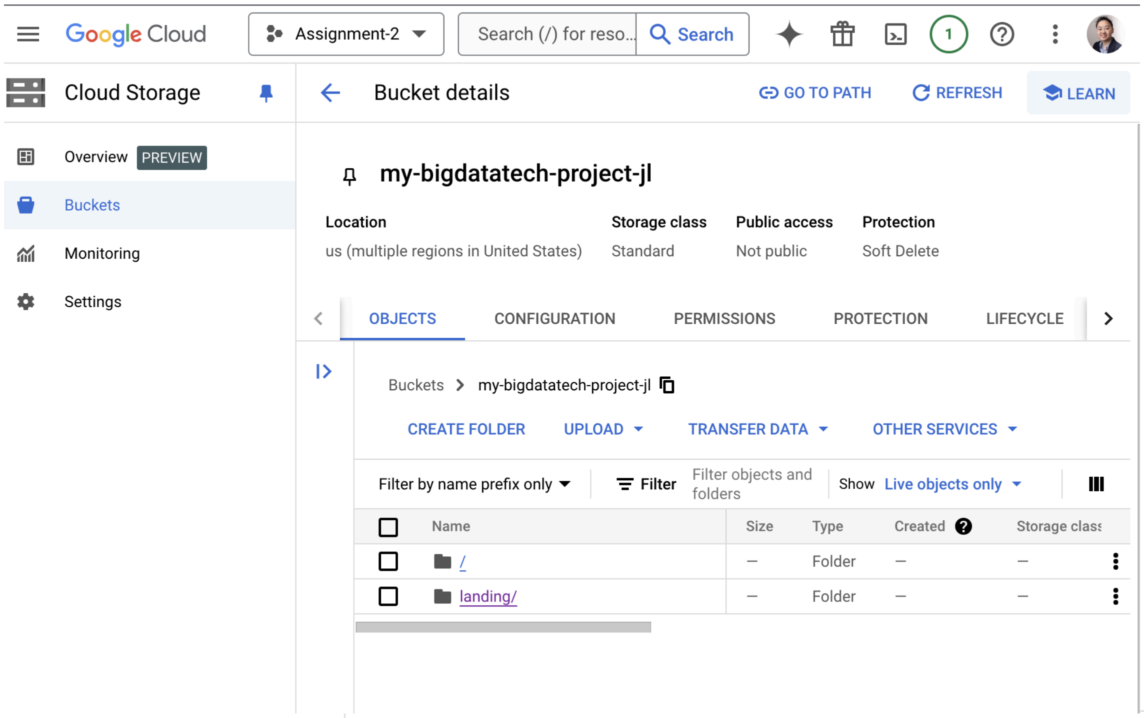
After unzipping the file with the command unzip flightprices.zip and using ls -l command to check our files, we can see the storage (in GB) of the zipped and unzipped datasets:



```
  inflating: itineraries.csv       ^C(pythondev) lin15jenny2003@instance-20240928-203012:
/pythondev$ ls -l
total 27024524
drwxr-xr-x 2 lin15jenny2003 lin15jenny2003        4096 Sep 28 21:14 bin
-rw-r--r-- 1 lin15jenny2003 lin15jenny2003  5920770411 Oct 19  2022 flightprices.zip
drwxr-xr-x 3 lin15jenny2003 lin15jenny2003        4096 Sep 28 20:51 include
-rw-r--r-- 1 lin15jenny2003 lin15jenny2003 21752315904 Sep 28 21:26 itineraries.csv
drwxr-xr-x 3 lin15jenny2003 lin15jenny2003        4096 Sep 28 20:51 lib
lrwxrwxrwx 1 lin15jenny2003 lin15jenny2003           3 Sep 28 20:51 lib64 -> lib
-rw-r--r-- 1 lin15jenny2003 lin15jenny2003         169 Sep 28 20:51 pyvenv.cfg
```

5. Then we will create a new google cloud storage bucket with the command gs://my-bigdatatech-project-jl –project=concise-ranger-435903-i6 –default-storage-class=STANDARD –location=us-central1 –uniform-bucket-level-access

6. Then create the itineraries.csv into the landing folder with the command gcloud storage cp itineraries.csv gs://my-bigdatatech-project-jl/landing/



Cloud Storage

Overview PREVIEW
Buckets
Monitoring
Settings

Bucket details — GO TO PATH | REFRESH | LEARN

my-bigdatatech-project-jl

Location	Storage class	Public access	Protection
us (multiple regions in United States)	Standard	Not public	Soft Delete

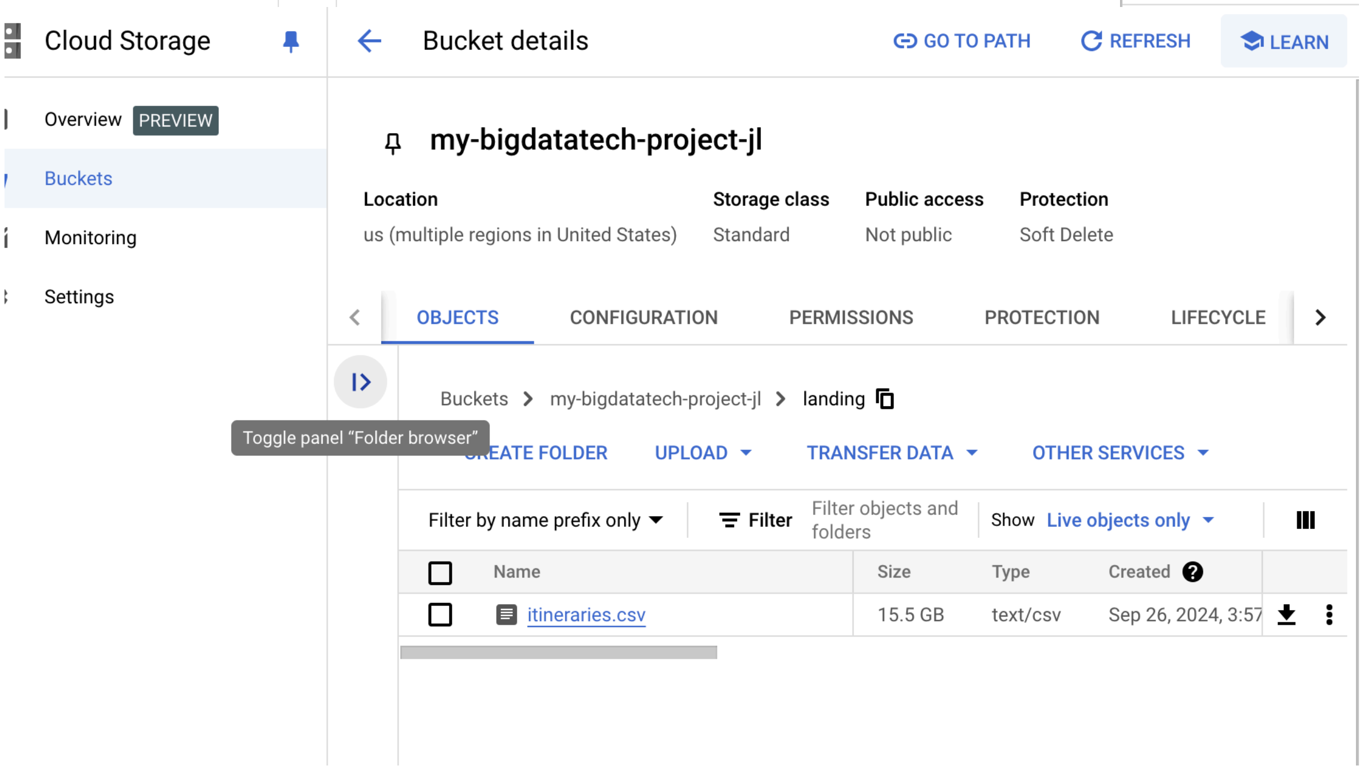
OBJECTS | CONFIGURATION | PERMISSIONS | PROTECTION | LIFECYCLE

Buckets > my-bigdatatech-project-jl

CREATE FOLDER | UPLOAD | TRANSFER DATA | OTHER SERVICES

Filter by name prefix only | Filter | Filter objects and folders | Show Live objects only

Name	Size	Type	Created	Storage class
/	—	Folder	—	—
landing/	—	Folder	—	—



Cloud Storage

Overview PREVIEW
Buckets
Monitoring
Settings

Bucket details — GO TO PATH | REFRESH | LEARN

my-bigdatatech-project-jl

Location	Storage class	Public access	Protection
us (multiple regions in United States)	Standard	Not public	Soft Delete

OBJECTS | CONFIGURATION | PERMISSIONS | PROTECTION | LIFECYCLE

Buckets > my-bigdatatech-project-jl > landing

Toggle panel “Folder browser”

CREATE FOLDER | UPLOAD | TRANSFER DATA | OTHER SERVICES

Filter by name prefix only | Filter | Filter objects and folders | Show Live objects only

Name	Size	Type	Created
itineraries.csv	15.5 GB	text/csv	Sep 26, 2024, 3:57

Milestone 3

Exploratory Data Analysis

In the EDA portion of my big data project, I used PySpark on a dataproc cluster to process and analyze my data and convert the PySpark to a Pandas data frame to create data visualizations and analysis. The dataset, loaded directly from GCS, contains 44,589,769 records. The initial basic exploration included obtaining the list of variables, examining missing values per column, and imputing the totalTravelDistance with the mean and imputing segmentsEquipmentDescription with the mode, and obtaining statistics for all numeric columns (max, min, avg, std. dev). After assessing the missing data, I noted the totalTravelDistance contained 3,718,047 missing values (~8.3%) and the segmentsEquipmentDescription had 883,152 missing values. For visualizations, I created a box plot with x-axis being segmentsEquipmentDescription and the y-axis being totalFare to analyze the relationship between total fare and the plane model. I noticed Embraer 175 with enhanced winglets || Airbus A320 || Embraer 175 has one of highest total fares (500). For the box plot with total travel distance vs total fare, there is a large spread of total fare with a total travel distance of 2337 miles. I created a histogram plot that shows how fares are distributed for one-way flights that shows flight prices from $180 - 590 USD where fares are widely dispersed with the most common fare being around $420. Lastly, a histogram analyzing how ticket prices are distributed across all flights where it is observed that multi-segment flights with United and Delta have the highest average total fare.

```
print(f"Number of records: {df.count()}")
```

```
Number of records: 44589769
```

```
# List of variables
variables = df.columns
print(variables)
```

```
['legId', 'searchDate', 'flightDate', 'startingAirport', 'destinationAirport',
'fareBasisCode', 'travelDuration', 'elapsedDays', 'isBasicEconomy',
'isRefundable', 'isNonStop', 'baseFare', 'totalFare', 'seatsRemaining',
'totalTravelDistance', 'segmentsDepartureTimeEpochSeconds',
'segmentsDepartureTimeRaw', 'segmentsArrivalTimeEpochSeconds',
'segmentsArrivalTimeRaw', 'segmentsArrivalAirportCode',
'segmentsDepartureAirportCode', 'segmentsAirlineName', 'segmentsAirlineCode',
'segmentsEquipmentDescription', 'segmentsDurationInSeconds',
'segmentsDistance', 'segmentsCabinCode']
```

```
# Number of missing values per column
missing_values = df.select([F.count(F.when(F.col(c).isNull(), c)).alias(c) for
c in df.columns])
missing_values.show()
```

```
Key Findings (other observations have either 1 or no null values):
```

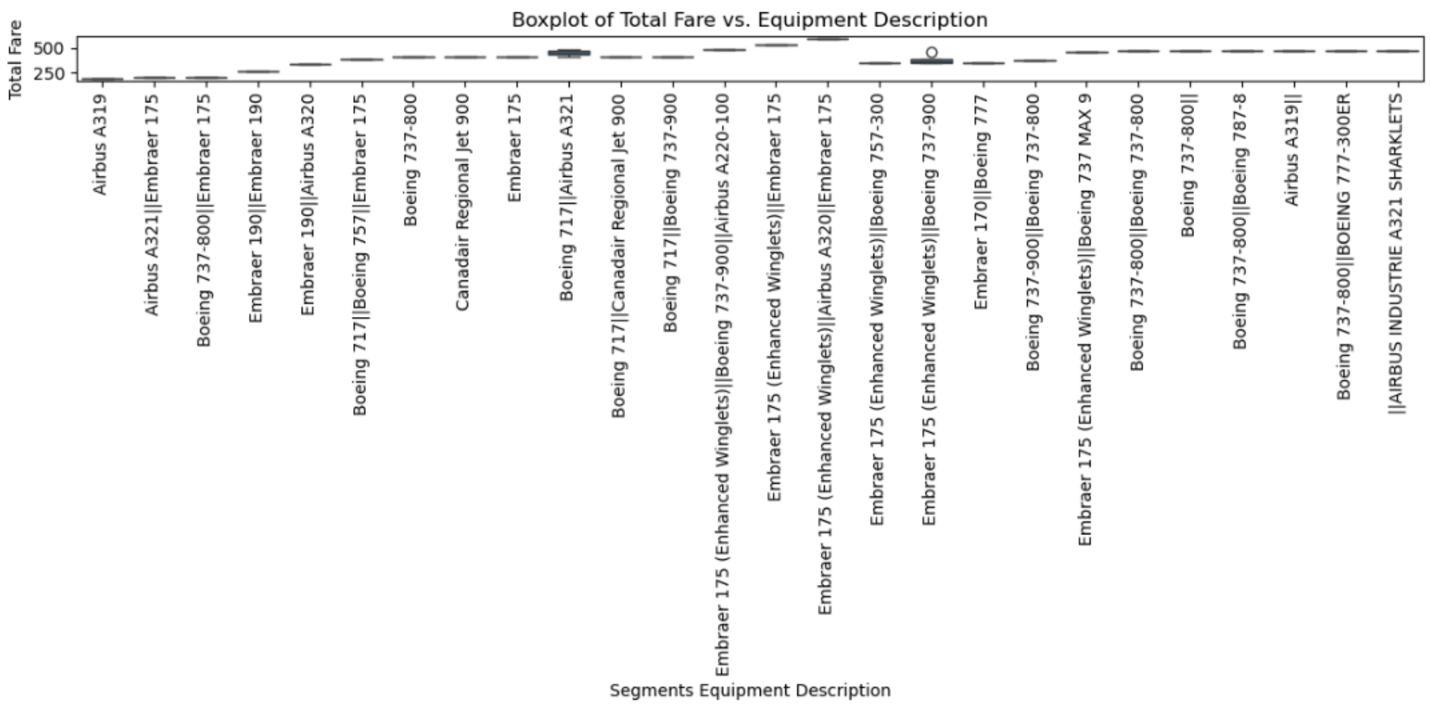
```
segmentsEquipmentDescription: 883,152 null values
totalTravelDistance: 3,718,047 null values
```

```python
# Get statistics with df.describe() function
# find min, max, avg, std dev for all numeric variables
numeric_stats = df.describe()
numeric_stats.show()
```

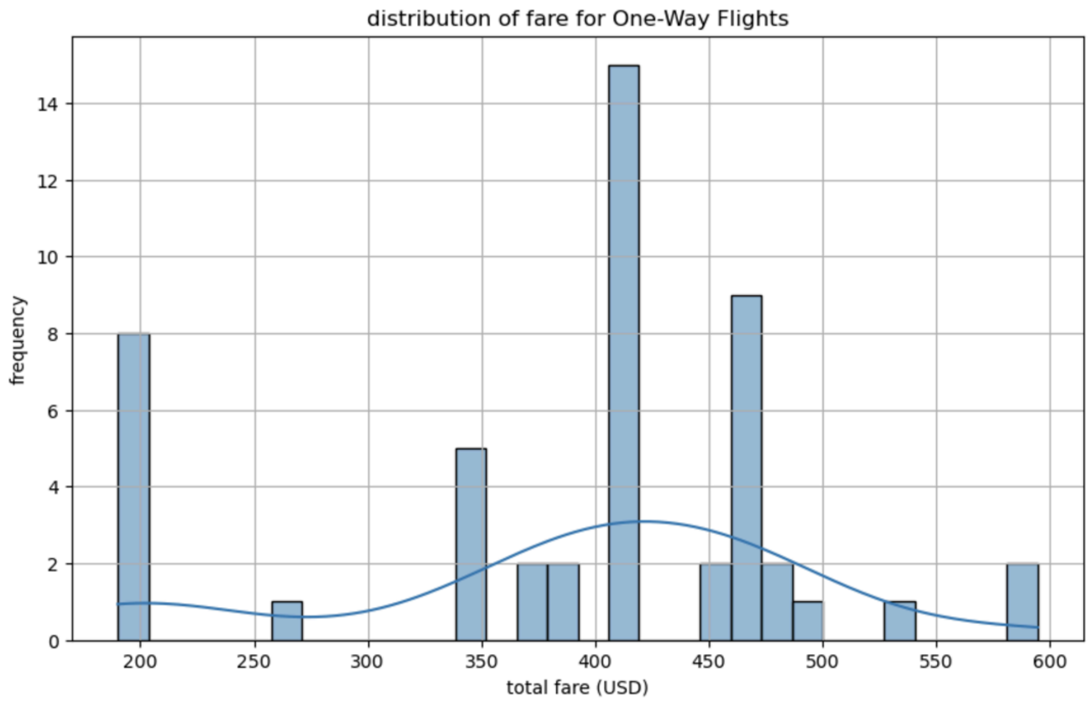
```
+-------+-------------------+-------------------+-------------------+-------------------+-------------------+
|summary|elapsedDays        |baseFare           |totalFare          |seatsRemaining     |totalTravelDistance|
+-------+-------------------+-------------------+-------------------+-------------------+-------------------+
|count  |44589768           |44589768           |44589768           |44589768           |44589769           |
|mean   |0.15119594253103089|320.02159316912497 |370.02160136958565 |5.720673294375516  |1571.8602124850659 |
|stddev |0.3582823364929339 |191.5858284192446  |204.5770820219753  |2.9507164413147393 |805.6285789791551  |
|min    |0                  |0.41               |23.97              |0                  |97                 |
|max    |2                  |7662.33            |8260.61            |10                 |4681               |
+-------+-------------------+-------------------+-------------------+-------------------+-------------------+
```

```python
# get min and max dates for date variables
columns = df.columns

date_columns = [c for c in columns if 'date' in c.lower()]

for date_col in date_columns:
    min_date = df.select(F.min(date_col)).first()[0]
    max_date = df.select(F.max(date_col)).first()[0]
    print(f"{date_col} - Min date: {min_date}, Max date: {max_date}")
```

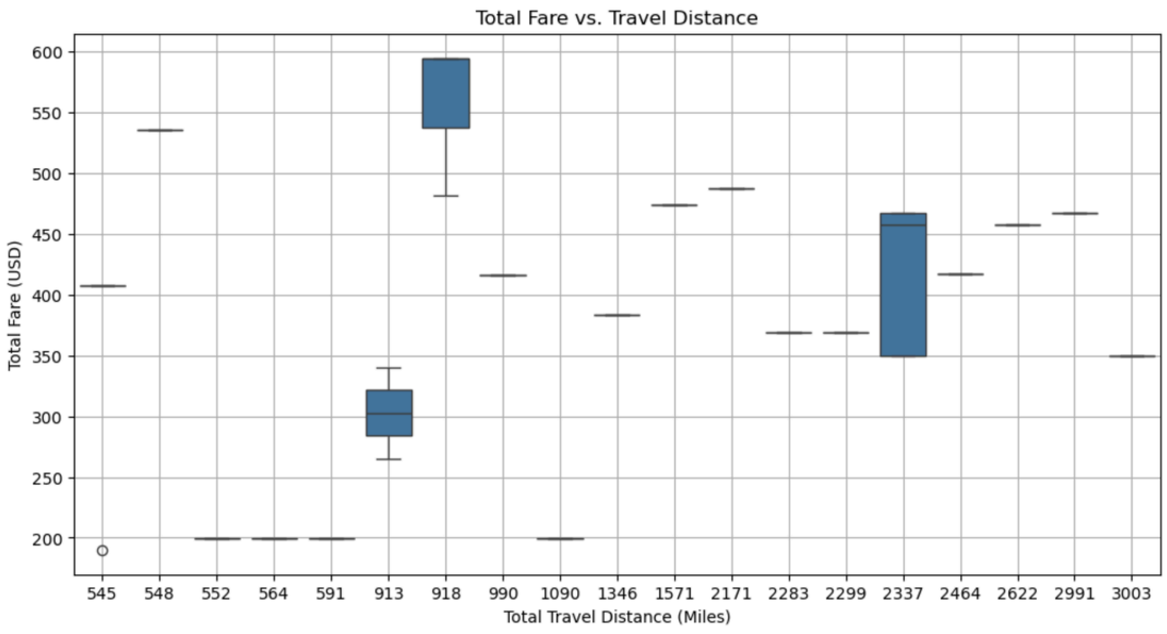
```
searchDate - Min date: 2022-04-16, Max date: 2022-07-22
flightDate - Min date: 2022-04-17, Max date: 2022-09-19
```



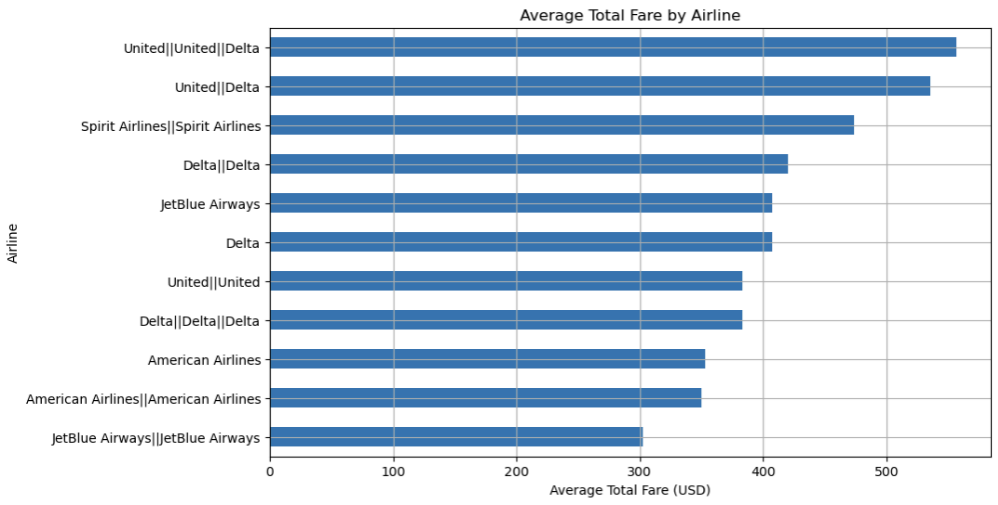
The boxplot shows the variation in total fare prices based on different equipment (aircraft types), highlighting how specific aircraft are associated with higher/lower fees.



The histogram shows the distribution of total fare for one-way flights, which demonstrates common fare ranges and fare outliers.



The boxplot illustrates the variation in total fare across different travel distances, highlighting the range, median, and outliers for each distance bracket.



The bar graph displays the average total fare (USD) for each airline, highlighting differences in ticket pricing across carriers and multi-leg flights.

Data Cleaning

The data is very clean, with all 27 columns having 1 or 0 null values except totalTravelDistance and segmentsEquipmentDescription. These 2 variables had a significant amount of null values, with segmentsEquipmentDescription having 883,152 null values and totalTravelDistance having 3,718,047 null values, but I decided to impute the totalTravelDistance as it is an important feature in predicting the flight price. After imputing the mean, I calculated the percentage of missing values and it was missing about 8%, so I imputed the rest with the median. I also dropped the segmentsDepartureTimeEpochSeconds and the segmentsArrivalTimeEpochSeconds columns since they were not human-readable formats and not relevant to predicting the flight price. The Expedia dataset contains search date from 4-16-2022 to 07-22-2022 and flight date from 04-17-2022 to 09-19-2022. The cleaned data is saved to the /cleaned folder as a parquet. I believe a problem I'll have in feature engineering is creating new features from columns such as segmentsDurationInSeconds and segmentsDistance, since they contain multiple entries for multi-leg flights that are separated by II and these features will need be split into individual values to analyze each leg separately.

Milestone 4

Model Building

I used PySpark to load, preprocess, and model flight data from a Parquet file. First, I loaded my data from the /cleaned folder using the spark.read.parquet() method. I then started the feature engineering process. Some steps I performed were converting categorical string columns, startingAirport, destinationAirport, fareBasisCode, and segmentsCabinCode to numerical indices using StringIndexer. I assembled the features into a feature vector using VectorAssembler and used StandardScaler to ensure all features had similar scales. I did some feature engineering with datetimes such as getting day of the week as a number, converting date columns to actual date data types, creating variables to check if search/flight date is on a weekend or holiday, seeing when booking was last minute or booked in advance, and checking if the flight is overnight or a daytime flight. I also converted boolean columns isBasicEconomy, isRefundable, isNonStop to integers so they are usable in my linear regression model. These transformations allowed me to create a structured dataset ready for modeling. The dataset was split into training and testing sets using an 80/20 ratio, implemented with a Linear Regression model to predict totalFare of flights, and evaluated using Root Mean Squared Error (RMSE). The results were written to a text file and the processed data was saved to the /trusted folder while the trained model was saved to the /models folder. Some challenges I encountered was ensuring data was properly scaled and not skewed. I also had to brainstorm different relevant features that were appropriate for flight data. Trying to identify how to transform features was initially confusing as I had to convert variables to a different data type in order to ensure values are consistent. I also had trouble initially reading my cleaned dataset, as it was stored as a parquet file instead of a csv and required manual specification of the schema to load it. I fixed this issue by restarting my kernel, creating a new dataproc cluster, and including the entire path of the parquet file. I also had issues with data types of columns not being supported by the model (string data types) that I had to convert to train my model. I also faced null values in my columns which I filled with 0s since the VectorAssembler cannot process. Summary of my outputs: I was able to create a new feature called seatAvailabilityCategory with criteria of low, medium, and high based on the number of seats remaining. I created other new variables: bookingProximity, flightTiming, season, isHoliday, flightDate_weekend, flightdate_yearmonth, flightDate_dayofweek, and flightDate_dayofweekname. After applying indexing, assembling, and scaling transformations, the data is stored in /trusted/processed_data as a parquet file. The trained linear regression model is saved in the /models/linear_regression_model directory, and the evaluation of the model after training (using the RMSE) is written to the model_evaluation.txt in the /results folder.

Source Code and Library References:

- Pyspark for data processing, feature engineering, and model training
- Os for managing file paths

Model: Predict the variable totalFare

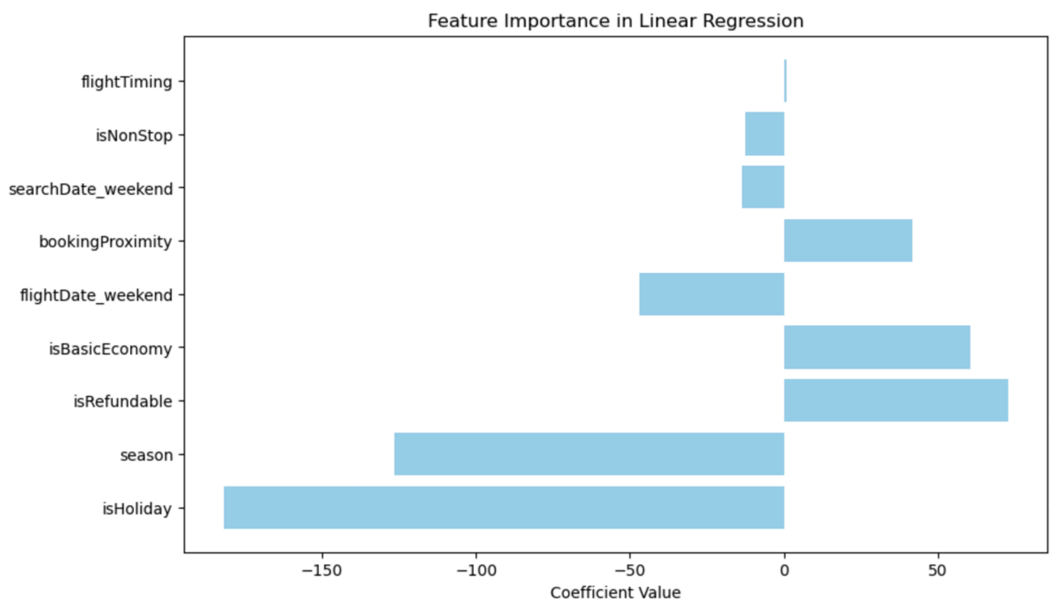
Features

Column	Data Type	Variable Type	Feature Engineering Treatment
startingAirport	String	Categorical	StringIndexer
destinationAirport	String	Categorical	StringIndexer
fareBasisCode	String	Categorical	StringIndexer
travelDuration	String	Categorical	VectorAssembler, StandardScaler
elapsedDays	Integer	Continuous	VectorAssembler
isBasicEconomy	Integer	Continuous	VectorAssembler
isRefundable	Integer	Continuous	VectorAssembler
isNonStop	Integer	Continuous	VectorAssembler
baseFare	Float	Continuous	VectorAssembler
totalFare	Float	Continuous	Label
seatsRemaining	Integer	Continuous	VectorAssembler
totalTravelDistance	Integer	Continuous	VectorAssembler
segmentsDurationInSeconds	Integer	Continuous	VectorAssembler
segmentsDistance	Integer	Continuous	VectorAssembler
segmentsCabinCode	String	Categorical	StringIndexer

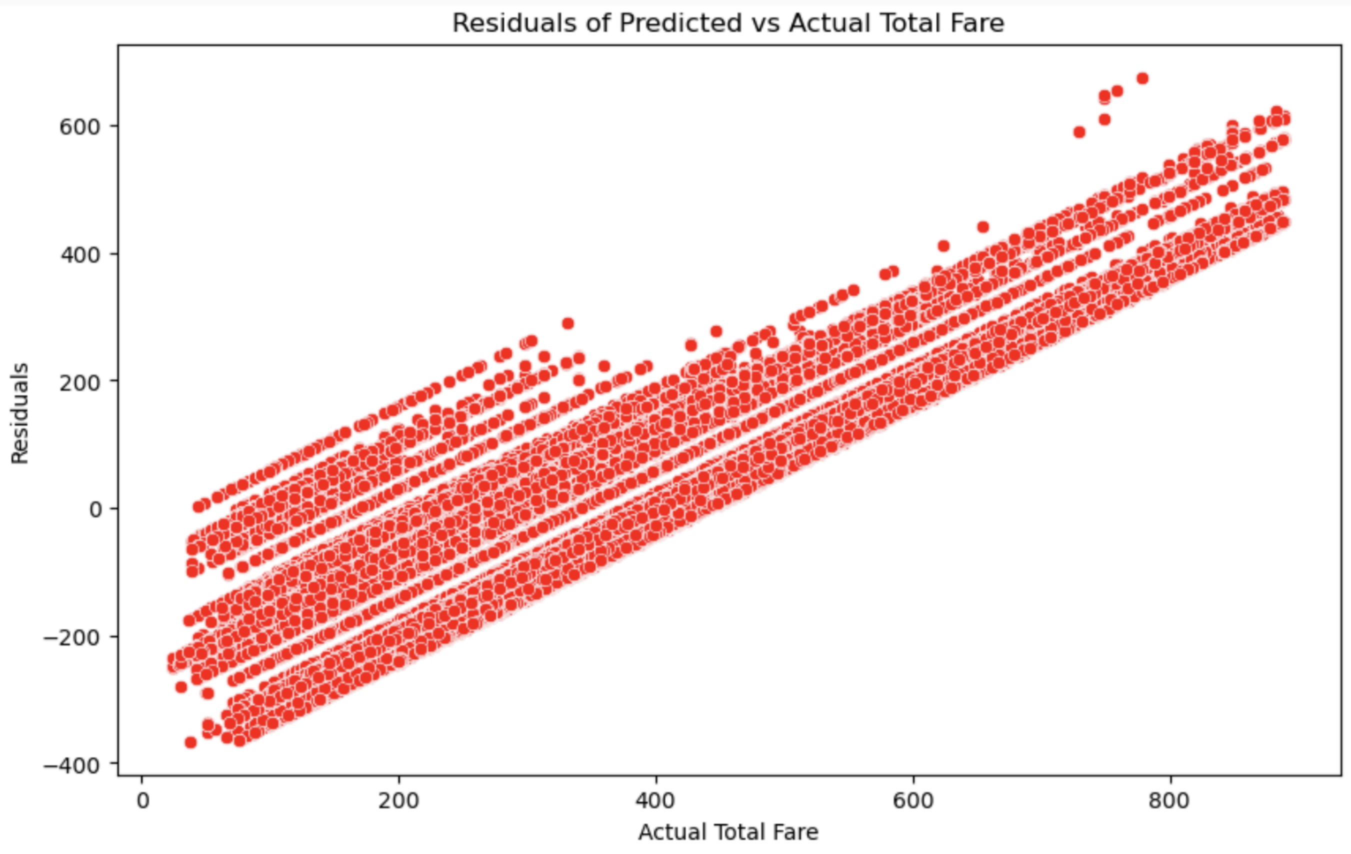
Label

totalFare	Integer	Continuous

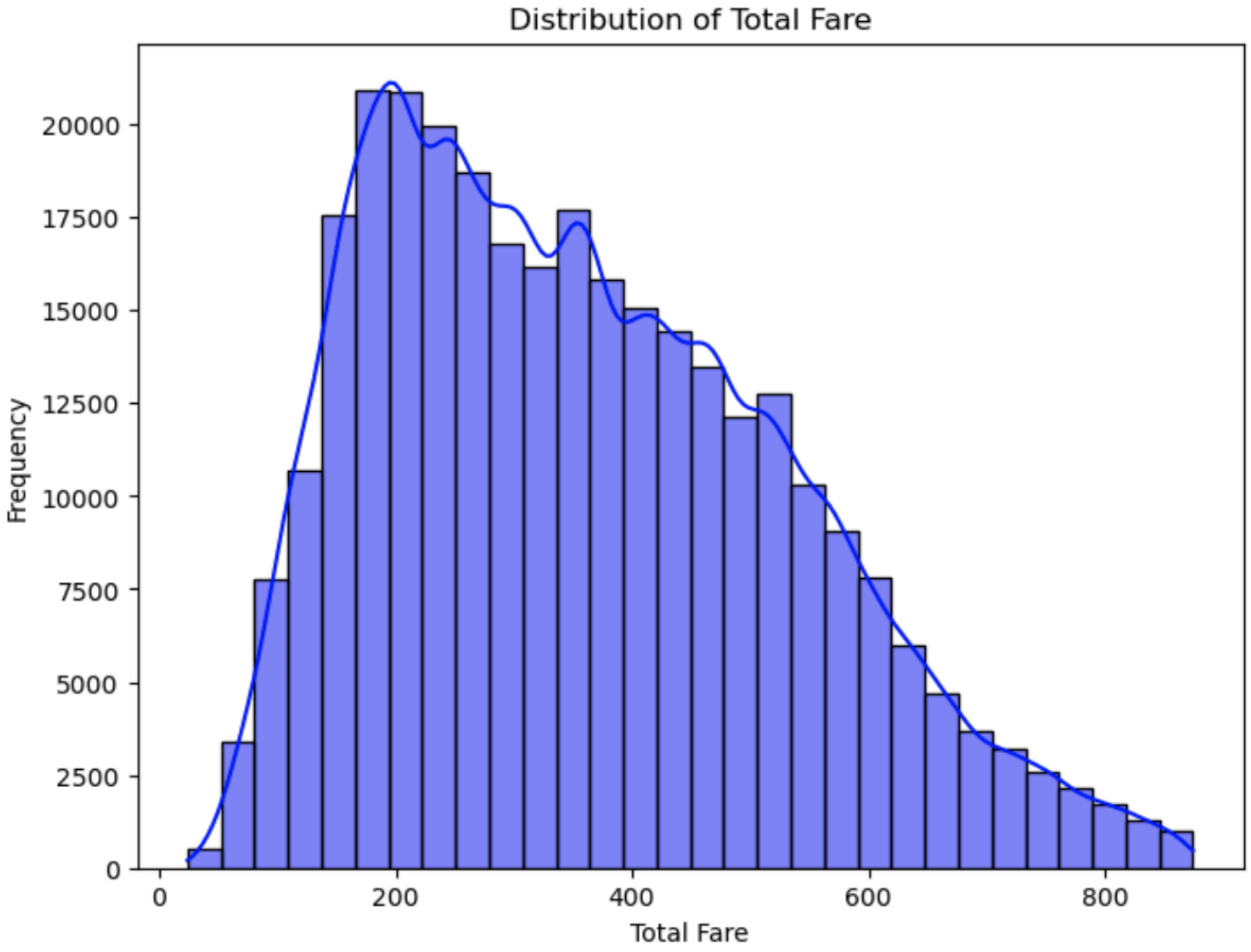
Data Visualization



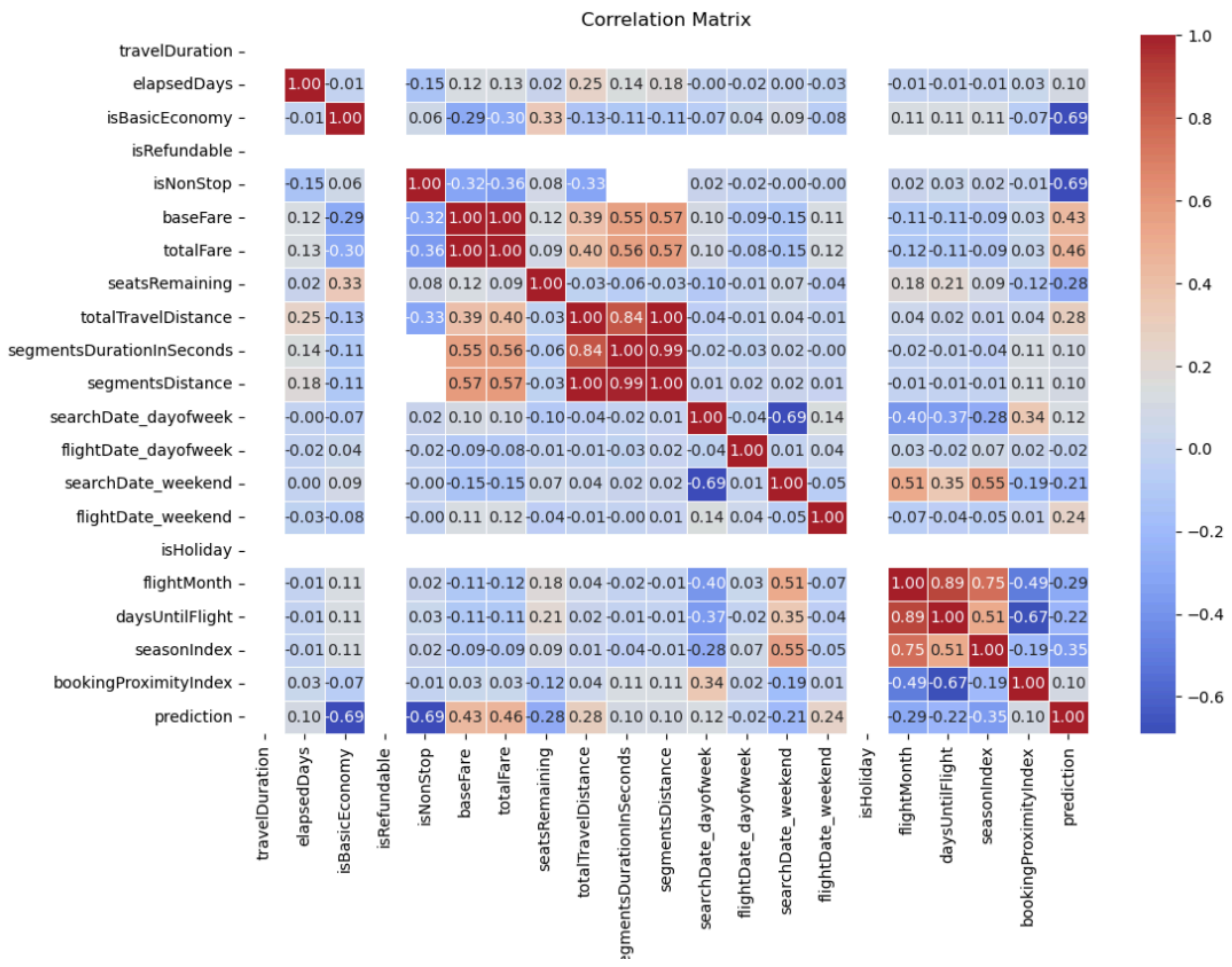
This bar chart shows the importance of the features used in training the linear regression model. The features isRefundable, isBasicEconomy, and bookingProximity have the highest coefficient values, meaning that they have the most impact on predicting the total fare.



This plot shows the residuals (the difference between actual and predicted total fare) against the actual total fare values. The positive upward trend in the residuals shows that the model's prediction error increases as the total fare increases, which may mean that the model performs is less accurate for high fares. This could also mean that the model needs further analysis for higher fare predictions.



The histogram displays the distribution of total fares. The x-axis represents the total fare range, and the y-axis represents the frequency. The distribution is right-skewed, indicating that most fares are concentrated in the lower range. The highest frequency of total fares is observed below 500, with the peak occurring around 250-500. The average total fare (mean) is 359. By filtering out the outliers using IQR method, our total fare falls within the range of 24-889.



Analyzing the correlation matrix above, it seems the target variable totalFare is not strongly correlated with the training features. The highest positive correlation with totalFare is segments distance variable with a 0.57 correlation and segments duration in seconds with a 0.56 correlation. The correlation matrix shows weak negative correlations between totalFare and isNonStop (-0.36) and isBasicEconomy (-0.29), suggesting that non-stop and basic economy flights have slightly lower fares. Since a lot of features are notstrongly correlated, these features have a limited impact on totalFare.

The most important features that contribute to totalFare are bookingProximity, isBasicEconomy, segmentsDuration, segmentsDistance, totalTravelDistance, and isRefundable according to the visualizations such as the correlation matrix and the bar graph.

Milestone 6

Summary and Conclusion

In this project, I developed and executed a data processing pipeline that aimed to analyze a flight prices dataset. The pipeline was designed to clean, transform, and analyze the data. This led to creating visualizations in the end such as a histogram to visualize distribution of total fare, a correlation matrix of most relevant features, and feature importance in linear regression. The data pipeline included the following steps: data importation from Kaggle using an API token, exploratory data analysis (EDA), data cleaning, feature engineering, splitting the dataset into training and testing sets. building the model (linear regression), fitting the model on the training data, evaluating the model with RMSE and R-squared, and tuning the hyperarameters using cross-validation to improve the model. The final model was able to predict the total fare of a ticket based on relevant features such as flight timings, seasonality, refundability, and booking proximity. The model has a RMSE of 156.33 and a R-squared of 0.2004. From these results, the model captures some variance in flight prices, but can be improved to account for larger amount of variance in flight prices. Some ideas include adding more relevant features or advanced models that can capture non-linear relationships. For further improvements, I would explore other regression techniques like Ridge or Lasso regression, or Random Forest which can handle complex patterns in my dataset.

Milestone 7

Github

Appendix A: Data Aquisition

This portion contains the process of acquiring the data from Kaggle

```
Mkdir  .kaggle
Kaggle.json
Mv kaggle.json .kaggle/
Chmod 600 .kaggle/kaggle.json
# set up python env
Sydo apt -y install python3-pip python3.11-venv
Python3 -m venv pythondev
Cd pythondev
Source bin/activage
Pip3 install kaggle
# download data
Kaggle datasets download -d dilwong/flightprices
# install utilities
Sudo apt install zip
Unzip flightprices.zip
# verify storage size of zipped and unzipped datasets
Ls -l
# create GCP bucket
Gcloud storage buckets create gs://my-bigdatatech-project-jl
–project=concise-ranger-435903-i6 –default-storage-class=STANDARD
–location=us-central1 –uniform-bucket-level-access
# move the itineraries.csv file into the landing folder
gcloud storage cp itineraries.csv gs://my-bigdatatech-project-jl/landing/
```

Appendix B: Data Exploratory Analysis

This portion contains the process of analyzing and visualizing the dataset

```
[ ]: from pyspark.sql import SparkSession

[ ]: # Create a Spark session with increased executor memory
spark = SparkSession.builder \
.appName("MyApp") \
.config("spark.executor.memory", "8g") \
.config("spark.driver.memory", "8g") \
.getOrCreate()

[ ]: df =
spark.read.csv("gs://my-bigdatatech-project-jl/landing/itineraries.csv",␣
↪header=True, inferSchema=True)

[ ]: print(f"Number of records: {df.count()}")
[ ]: # List of variables
variables = df.columns
print(variables)

[ ]: from pyspark.sql import functions as F
# Number of missing values per column
missing_values = df.select([F.count(F.when(F.col(c).isNull(), c)).alias(c) for␣
↪c in df.columns])
# Show result
missing_values.show()

[ ]: from pyspark.sql.functions import col, when
def count_nulls(df, segmentsEquipmentDescription):
"""
Function to count the number of null values in a given column of a␣
↪DataFrame.
:param df: The PySpark DataFrame
:param column_name: The column for which to count null values
:return: The count of null values in the specified column
"""
return df.select(when(col(segmentsEquipmentDescription).isNull(), 1).
↪alias(segmentsEquipmentDescription)).groupBy().
↪sum(segmentsEquipmentDescription).collect()[0][0]
# Example usage
null_count = count_nulls(df, "segmentsEquipmentDescription")
print(f"Number of null values in segmentsEquipmentDescription: {null_count}")

[ ]: def count_nulls(df, totalTravelDistance):
"""
Function to count the number of null values in a given column of a␣
↪DataFrame.
:param df: The PySpark DataFrame
:param column_name: The column for which to count null values
:return: The count of null values in the specified column
"""
return df.select(when(col(totalTravelDistance).isNull(), 1).
↪alias(totalTravelDistance)).groupBy().sum(totalTravelDistance).
↪collect()[0][0]
null_count = count_nulls(df, "totalTravelDistance")
3
print(f"Number of null values in totalTravelDistance: {null_count}")

[ ]: # Get the first 10 rows of the segmentsEquipmentDescription column
df.select("segmentsEquipmentDescription").show(10)

[ ]: # Get statistics with df.describe() function
# find min, max, avg, std dev for all numeric variables
numeric_stats = df.describe()
numeric_stats.show()
```

```
[ ]: from pyspark.sql import functions as F
# get min and max dates for date variables
columns = df.columns
date_columns = [c for c in columns if 'date' in c.lower()]
for date_col in date_columns:
min_date = df.select(F.min(date_col)).first()[0]
max_date = df.select(F.max(date_col)).first()[0]
print(f"{date_col} - Min date: {min_date}, Max date: {max_date}")
```

```
[ ]: # Filter the Spark DataFrame (example condition)
filtered_df = df.filter(df.totalFare.isNotNull() & df.
↪segmentsEquipmentDescription.isNotNull())
# Limit to 10 rows
limited_df = filtered_df.limit(50)
# Convert to Pandas DataFrame
pandas_df = limited_df.toPandas()
```

```
[ ]: import seaborn as sns
import matplotlib.pyplot as plt
# Set the size of the plot
plt.figure(figsize=(12, 6))
# Create a boxplot
sns.boxplot(x='segmentsEquipmentDescription', y='totalFare', data=pandas_df)
# Set title and labels
plt.title('Boxplot of Total Fare vs. Equipment Description')
plt.xlabel('Segments Equipment Description')
plt.ylabel('Total Fare')
6
plt.yticks(fontsize=10)
plt.xticks(fontsize=10)
# Rotate x-axis labels for better readability (if needed)
plt.xticks(rotation=90)
# Show the plot
plt.tight_layout()
plt.show()
plt.tight_layout()
```

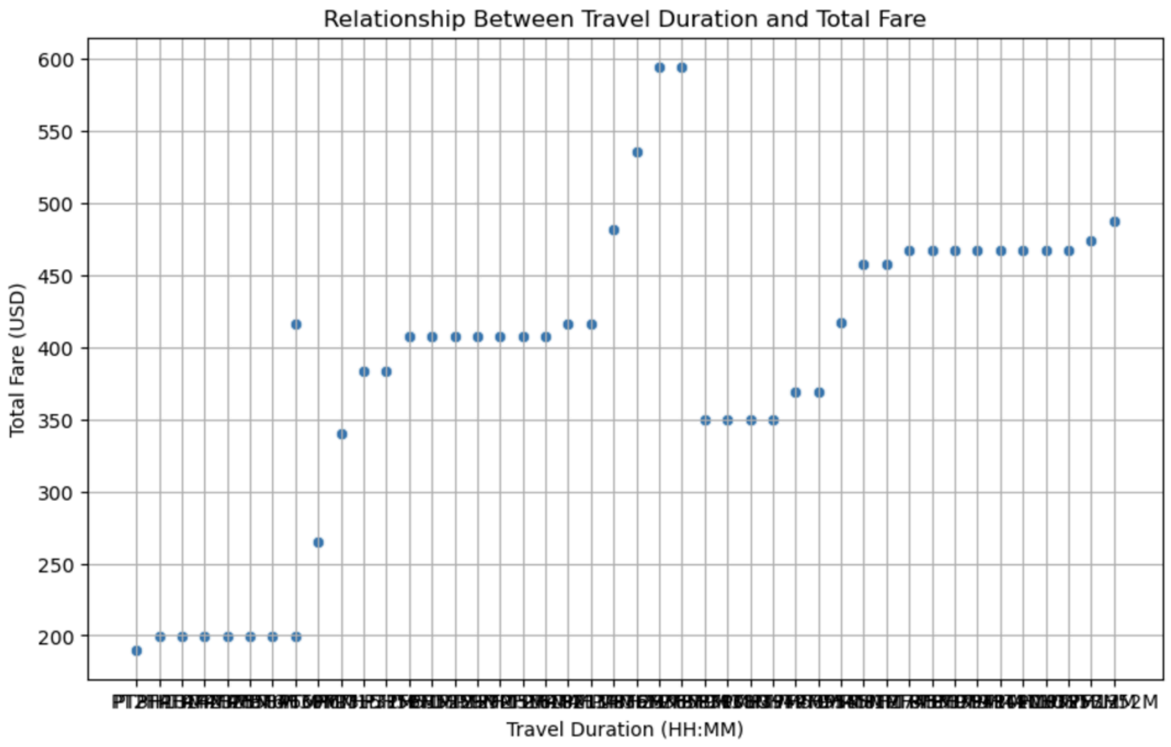
```
[ ]: # get distribution of total fare for one-way flights
plt.figure(figsize=(10, 6))
sns.histplot(pandas_df['totalFare'], bins=30, kde=True)
plt.title('distribution of fare for One-Way Flights')
plt.xlabel('total fare (USD)')
plt.ylabel('frequency')
plt.grid()
plt.show()
```

```
[ ]: # scatterplot to see if seats remaining and total fare have a strong
correlation
plt.figure(figsize=(10, 6))
```

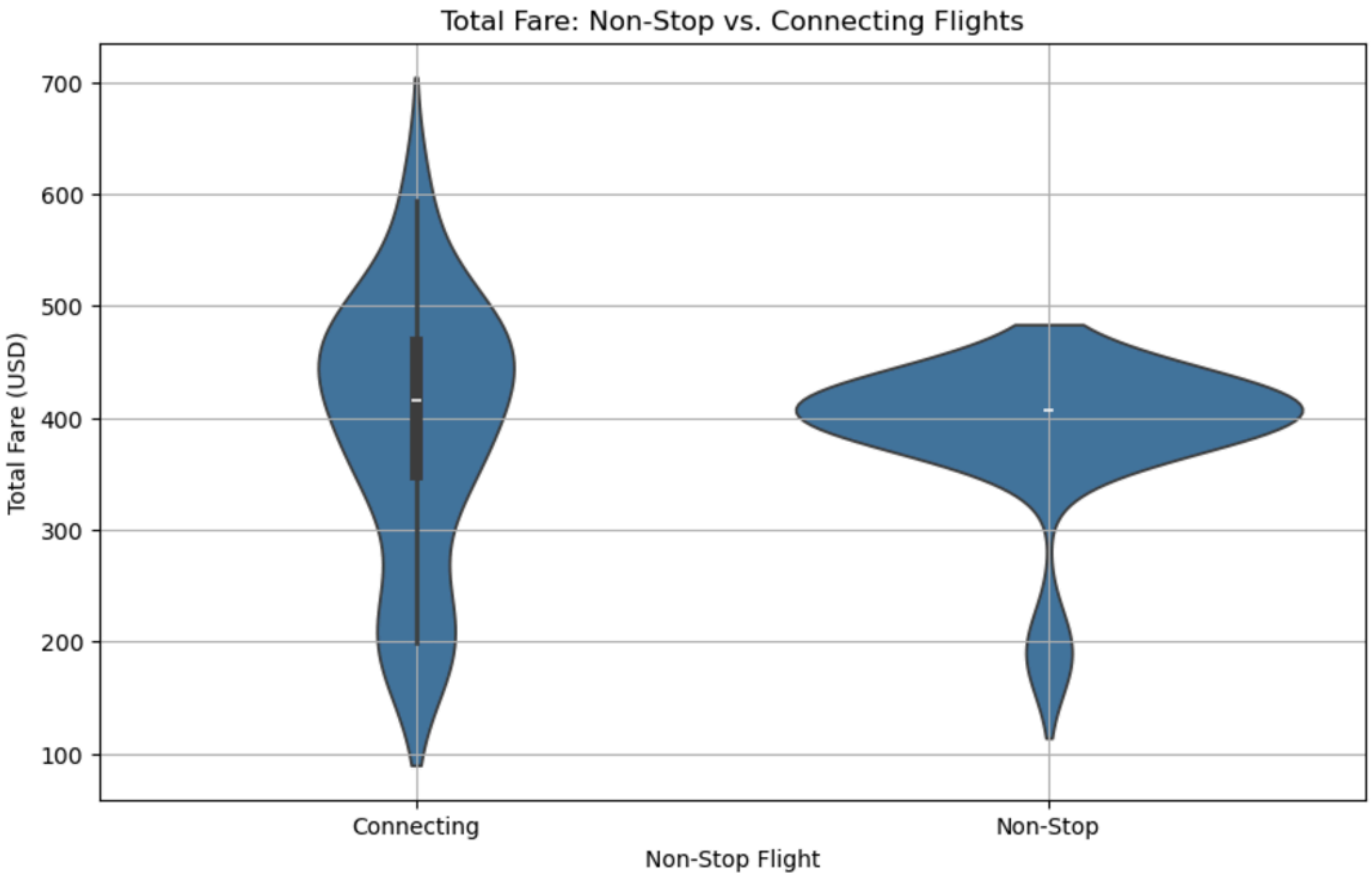
```
sns.scatterplot(x='seatsRemaining', y='totalFare', data=pandas_df)
plt.title('Seats Remaining vs. Total Fare')
plt.xlabel('Seats Remaining')
plt.ylabel('Total Fare (USD)')
plt.grid()
plt.show()
```

```
[ ]: # travel distance vs total fare
plt.figure(figsize=(12, 6))
sns.boxplot(x='totalTravelDistance', y='totalFare', data=pandas_df)
plt.title('Total Fare vs. Travel Distance')
plt.xlabel('Total Travel Distance (Miles)')
plt.ylabel('Total Fare (USD)')
plt.grid()
plt.show()
```

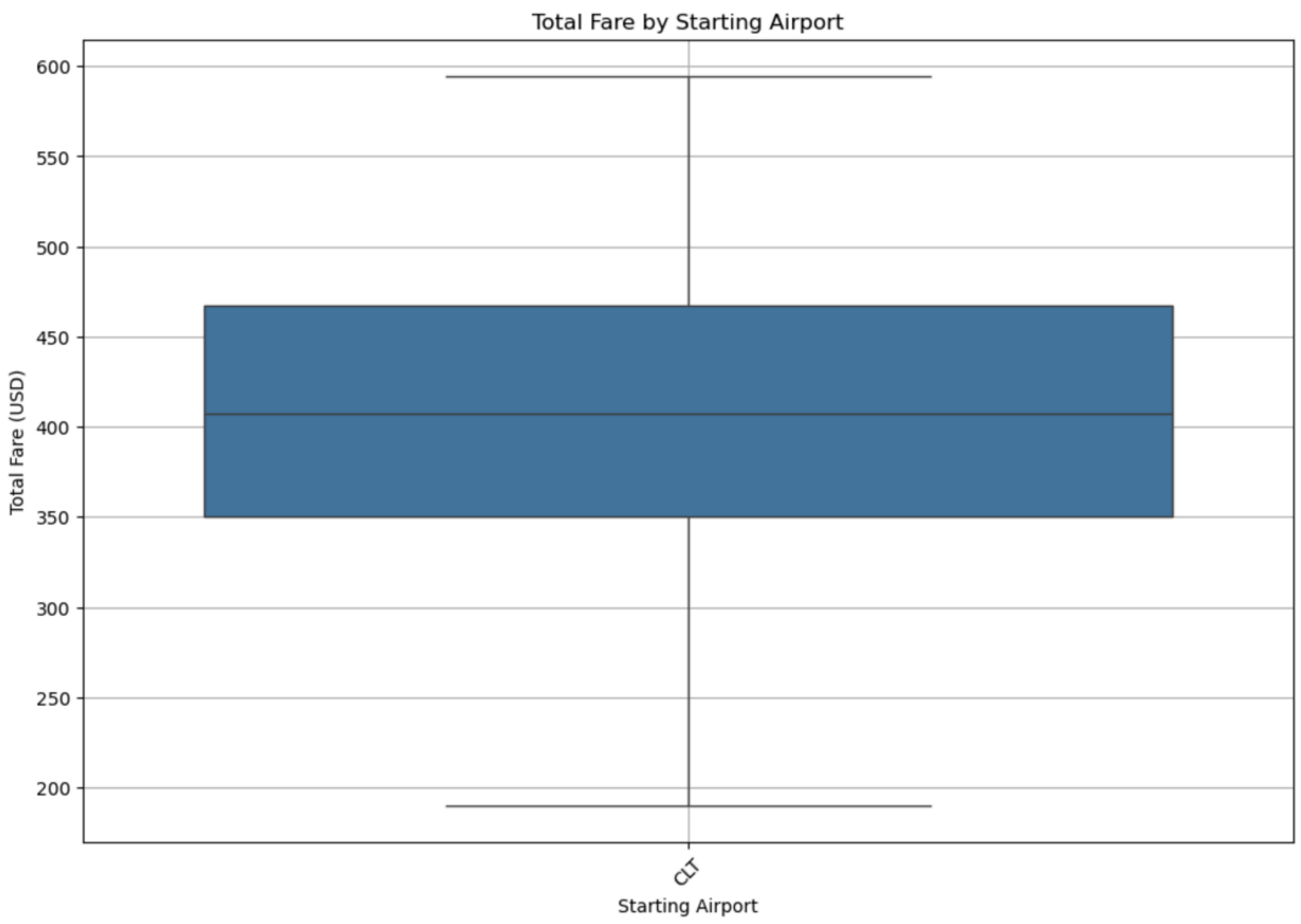
```
[ ]: # travel duration vs total fare
plt.figure(figsize=(10, 6))
sns.scatterplot(x='travelDuration', y='totalFare', data=pandas_df)
plt.title('Relationship Between Travel Duration and Total Fare')
plt.xlabel('Travel Duration (HH:MM)')
plt.ylabel('Total Fare (USD)')
plt.grid()
plt.show()
```



```
[ ]: # fare comparison of non-stop vs connecting flights
plt.figure(figsize=(10, 6))
sns.violinplot(x='isNonStop', y='totalFare', data=pandas_df)
plt.title('Total Fare: Non-Stop vs. Connecting Flights')
plt.xlabel('Non-Stop Flight')
plt.ylabel('Total Fare (USD)')
plt.xticks([0, 1], ['Connecting', 'Non-Stop'])
plt.grid()
plt.show()
```

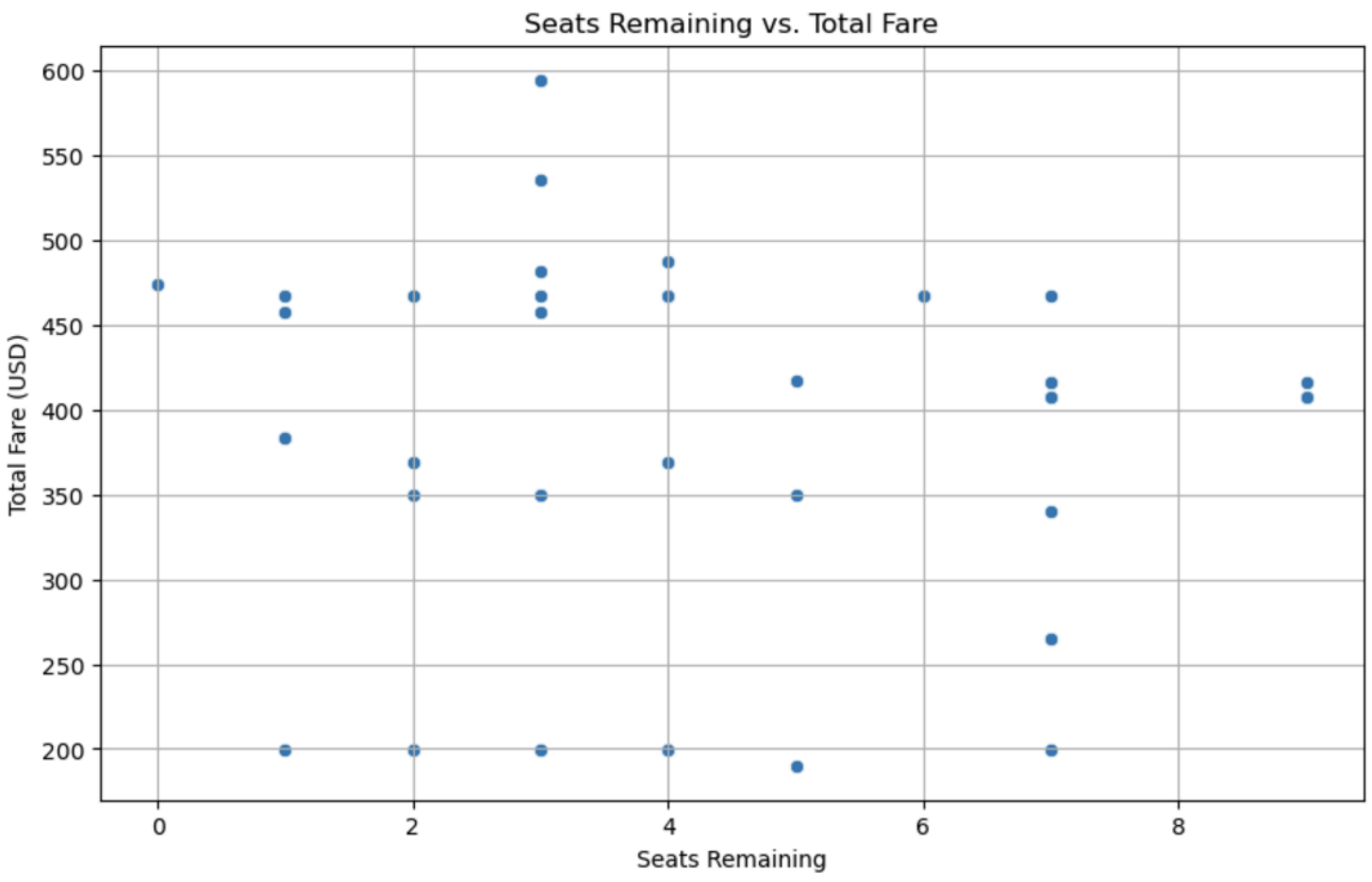


```
[ ]: # analyze fare variations between diff routes
plt.figure(figsize=(12, 8))
sns.boxplot(x='startingAirport', y='totalFare', data=df)
plt.title('Total Fare by Starting Airport')
plt.xlabel('Starting Airport')
plt.ylabel('Total Fare (USD)')
plt.xticks(rotation=45)
plt.grid()
plt.show()
```



```
[ ]: # average total fare by airline
average_fare_by_airline =
df.groupby('segmentsAirlineName')['totalFare'].mean().
↪sort_values()
plt.figure(figsize=(10, 6))
13
average_fare_by_airline.plot(kind='barh')
plt.title('Average Total Fare by Airline')
plt.xlabel('Average Total Fare (USD)')
plt.ylabel('Airline')
plt.grid()
plt.show()
```

```
[ ]: # use a histogram to analyze how ticket prices are distributed across all␣
↪flights
plt.figure(figsize=(10, 6))
sns.histplot(df['totalFare'], bins=30, kde=True)
plt.title('Distribution of Total Fare for One-Way Flights')
plt.xlabel('Total Fare (USD)')
plt.ylabel('Frequency')
plt.grid()
plt.show()
[ ]: # see if there's a correlation between num of seats remaining and total
fare
plt.figure(figsize=(10, 6))
sns.scatterplot(x='seatsRemaining', y='totalFare', data=df)
plt.title('Seats Remaining vs. Total Fare')
plt.xlabel('Seats Remaining')
plt.ylabel('Total Fare (USD)')
plt.grid()
plt.show()
```



Appendix C: Data Cleaning

This portion describes our data and what methods were used to clean it

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.functions import col, when, mean, expr

spark = SparkSession.builder \
    .appName("Data Cleaning") \
    .getOrCreate()

# load data
```

```
data_path = "gs://my-bigdatatech-project-jl/landing/itineraries.csv"
df_cleaned = spark.read.csv(data_path, header=True, inferSchema=True)

# group by all columns and count occurrences
duplicates = df_cleaned.groupBy(df_cleaned.columns).count().filter("count > 1")
duplicates.show()

# count nulls in a column
def count_nulls(df, column_name):
    return df.select(
        when(col(column_name).isNull(), 1).alias(column_name)
    ).groupBy().sum(column_name).collect()[0][0]

null_count = count_nulls(df_cleaned, "segmentsEquipmentDescription")
print(f"Number of null values in segmentsEquipmentDescription: {null_count}")

# impute missing values
# impute `totalTravelDistance` with mean
mean_value = df_cleaned.select(mean("totalTravelDistance")).collect()[0][0]
df_cleaned = df_cleaned.fillna({"totalTravelDistance": mean_value})

# impute `segmentsEquipmentDescription` with mode
mode_value = df_cleaned.groupBy("segmentsEquipmentDescription").count() \
    .orderBy("count", ascending=False).first()[0]
df_cleaned = df_cleaned.fillna({"segmentsEquipmentDescription": mode_value})

df_cleaned.select("totalTravelDistance", "segmentsEquipmentDescription").show()

# total rows
total_count = df_cleaned.count()

# calculate the percentage of missing values for each column
missing_percentages = {
    col: (df_cleaned.filter(F.col(col).isNull()).count() / total_count) * 100
    for col in df_cleaned.columns
}

# print the missing percentages
print("Missing percentages per column:")
for col, percent in missing_percentages.items():
    print(f"{col}: {percent:.2f}%")

# obtain the median value for `totalTravelDistance`
median_totalTravelDistance = df_cleaned.approxQuantile("totalTravelDistance",
[0.5], 0.01)[0]

# impute missing values with the median
df_cleaned = df_cleaned.fillna({"totalTravelDistance":
median_totalTravelDistance})
```

```
# check nulls in `totalTravelDistance`
null_totalTravelDistance =
df_cleaned.filter(df_cleaned.totalTravelDistance.isNull()).count()
null_percentage = (null_totalTravelDistance / total_count) * 100
print(f"Percentage of null values in totalTravelDistance:
{null_percentage:.2f}%")

# drop columns not in human-legible formats
df_cleaned = df_cleaned.drop('segmentsDepartureTimeEpochSeconds',
'segmentsArrivalTimeEpochSeconds')

cleaned_data_path =
"gs://my-bigdatatech-project-jl/cleaned/Data_Cleaned.parquet"
df_cleaned.write.parquet(cleaned_data_path)

spark.stop()
```

Appendix D: Feature Engineering and Modeling

This code shows the feature engineering and modeling process

```
# In[1]:

from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator
import os

spark.conf.set("spark.sql.debug.maxToStringFields", 1000)

# Create a Spark session
spark = SparkSession.builder \
    .appName("ML Pipeline") \
    .config("spark.executor.memory", "8g") \
    .config("spark.executor.cores", "4") \
    .config("spark.executor.instances", "4") \
    .config("spark.sql.shuffle.partitions", "100") \
    .getOrCreate()

# Google Cloud Storage path
gcs_path = 'gs://my-bigdatatech-project-jl/cleaned/Data_Cleaned.parquet'

# Read the Parquet file from GCS
df_spark = spark.read.parquet(gcs_path)
```

```
# Show the DataFrame schema and the first few rows
df_spark.printSchema()
df_spark.show()


# In[2]:

filtered_df = df_spark.filter(df_spark.totalFare.isNotNull() &
df_spark.segmentsEquipmentDescription.isNotNull())

limited_df = filtered_df.limit(50)

pandas_df = limited_df.toPandas()


# In[3]:


df_spark.dtypes


# In[4]:


from pyspark.sql.functions import to_date

# convert date columns to actual date data type
df_spark = df_spark.withColumn('searchDate', to_date(df_spark.searchDate,
'yyyy-MM-dd'))
df_spark = df_spark.withColumn('flightDate', to_date(df_spark.flightDate,
'yyyy-MM-dd'))


# In[5]:


from pyspark.sql.functions import date_format
from pyspark.sql.functions import col

# get year-month variable
df_spark = df_spark.withColumn("searchdate_yearmonth",
date_format(col("searchDate"), "yyyy-MM"))
df_spark = df_spark.withColumn('flightdate_yearmonth',
date_format(col("flightDate"), "yyyy-MM"))


# In[6]:


from pyspark.sql.functions import dayofweek
```

```
# get day of week as a number
df_spark = df_spark.withColumn("searchDate_dayofweek",
dayofweek(col("searchDate")) )
df_spark = df_spark.withColumn("flightDate_dayofweek",
dayofweek(col("flightDate")) )


# In[7]:


# get day of week as name e.g. Monday
df_spark = df_spark.withColumn("searchDate_dayofweekname",
date_format(col("searchDate"), "E"))
df_spark = df_spark.withColumn("flightDate_dayofweekname",
date_format(col("flightDate"), "E"))


# In[8]:


from pyspark.sql.functions import when

# check if search or flight date falls on a weekend
df_spark = df_spark.withColumn("searchDate_weekend",
when(df_spark.searchDate_dayofweek == 1,
1.0).when(df_spark.searchDate_dayofweek == 7, 1.0).otherwise(0))
# check if search or flight date falls on a weekend
df_spark = df_spark.withColumn("flightDate_weekend",
when(df_spark.flightDate_dayofweek == 1,
1.0).when(df_spark.flightDate_dayofweek == 7, 1.0).otherwise(0))


# In[9]:


from pyspark.sql.functions import col, when

# Define a list of holiday dates
holidays = [
    '2024-01-01',  # New Year's Day
    '2024-07-04',  # Independence Day
    '2024-12-25',  # Christmas Day
    '2024-11-28' # Thanksgiving Day
]

# Add a column to indicate if the flightDate is a holiday
df_spark = df_spark.withColumn(
    "isHoliday",
    when(col("flightDate").cast("string").isin(holidays), 1).otherwise(0)
)


# In[10]:


# Holiday Indicator
df_spark = df_spark.withColumn("isHoliday",
when(col("flightDate").cast("string").isin(holidays), 1).otherwise(0))


# In[11]:


from pyspark.sql.functions import month

# extract the month from flightDate and create a new column flightMonth
df_spark = df_spark.withColumn("flightMonth", month(col("flightDate")))

# create the season column based on the month
df_spark = df_spark.withColumn(
    "season",
    when(col("flightMonth").isin(6, 7, 8), "Summer")
    .when(col("flightMonth").isin(12, 1, 2), "Winter")
    .when(col("flightMonth").isin(3, 4, 5), "Spring")
    .otherwise("Fall")
)


# In[12]:


from pyspark.sql.functions import lit, datediff

# 5. Flight Timing
df_spark = df_spark.withColumn("flightTiming",
when((col("segmentsDepartureTimeRaw") >= lit("18:00:00")) &
                                            (col("segmentsArrivalTimeRaw") <=
lit("06:00:00")), "Overnight")
                                       .otherwise("Daytime"))

# 6. Proximity of Booking
df_spark = df_spark.withColumn("daysUntilFlight", datediff("flightDate",
"searchDate")) \
           .withColumn("bookingProximity", when(col("daysUntilFlight") <= 1,
"Last Minute")
                                           .when((col("daysUntilFlight") > 1) &
(col("daysUntilFlight") <= 7), "Within a Week")
                                           .otherwise("Planned in Advance"))
```

```
# In[13]:


# check transformations
df_spark.select("flightTiming", "daysUntilFlight", "bookingProximity").show(10)


# In[14]:


from pyspark.sql.functions import col

# Converting boolean columns to integer - 0 or 1 - in PySpark
df_spark = df_spark.withColumn('isBasicEconomy',
col('isBasicEconomy').cast('int'))
df_spark = df_spark.withColumn('isRefundable', col('isRefundable').cast('int'))
df_spark = df_spark.withColumn('isNonStop', col('isNonStop').cast('int'))


# In[15]:


df_spark.select("isNonStop").show(10)


# In[16]:


# define seat availability volumes
df_spark = df_spark.withColumn(
    'seatAvailabilityCategory',
    when(col('seatsRemaining') <= 20, 'low')
    .when((col('seatsRemaining') > 20) & (col('seatsRemaining') <= 100),
'medium')
    .otherwise('high')
)

df_spark.select('seatsRemaining', 'seatAvailabilityCategory').show()


# In[17]:


from pyspark.sql.functions import col

# correct columns with right data types

df_spark = df_spark.withColumn("travelDuration",
col("travelDuration").cast("float")) \
                   .withColumn("segmentsDurationInSeconds",
col("segmentsDurationInSeconds").cast("int")) \
                   .withColumn("segmentsDistance",
col("segmentsDistance").cast("int"))


# In[18]:


from pyspark.sql.functions import col

# ensure columns are numeric
df_spark = df_spark.withColumn("elapsedDays", col("elapsedDays").cast("int")) \
           .withColumn("isBasicEconomy", col("isBasicEconomy").cast("int")) \
           .withColumn("isRefundable", col("isRefundable").cast("int")) \
           .withColumn("isNonStop", col("isNonStop").cast("int")) \
           .withColumn("baseFare", col("baseFare").cast("float")) \
           .withColumn("seatsRemaining", col("seatsRemaining").cast("int")) \
           .withColumn("totalTravelDistance",
col("totalTravelDistance").cast("int")) \
           .withColumn("segmentsDurationInSeconds",
col("segmentsDurationInSeconds").cast("int")) \
           .withColumn("segmentsDistance", col("segmentsDistance").cast("int"))


# In[19]:


df_spark.dtypes


# In[20]:


from pyspark.ml.feature import StringIndexer, VectorAssembler, StandardScaler,
OneHotEncoder
from pyspark.ml import Pipeline
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.sql.types import StructType, StructField, StringType, DateType,
IntegerType, FloatType, LongType


# In[21]:


schema = StructType([
    StructField("legId", StringType(), True),
    StructField("searchDate", DateType(), True),
    StructField("flightDate", DateType(), True),
    StructField("startingAirport", StringType(), True),
    StructField("destinationAirport", StringType(), True),
    StructField("fareBasisCode", StringType(), True),
    StructField("travelDuration", StringType(), True),
    StructField("elapsedDays", IntegerType(), True),
    StructField("isBasicEconomy", IntegerType(), True),
    StructField("isRefundable", IntegerType(), True),
    StructField("isNonStop", IntegerType(), True),
    StructField("baseFare", FloatType(), True),
    StructField("totalFare", FloatType(), True),
    StructField("seatsRemaining", IntegerType(), True),
    StructField("totalTravelDistance", IntegerType(), True),
    StructField("segmentsDepartureTimeEpochSeconds", LongType(), True),
    StructField("segmentsArrivalTimeEpochSeconds", LongType(), True),
    StructField("segmentsDurationInSeconds", IntegerType(), True),
    StructField("segmentsDistance", IntegerType(), True),
    StructField("segmentsCabinCode", StringType(), True)
])


# In[29]:


from pyspark.sql.functions import isnan

sampled_df = df_spark.sample(fraction=0.01, seed=42)

# split data into training and testing sets
train_data, test_data = sampled_df.randomSplit([0.7, 0.3], seed=42)

# remove nan values
train_data = train_data.filter(~isnan("totalFare"))
test_data = test_data.filter(~isnan("totalFare"))

# features for the model
feature_columns = ['isRefundable', 'isBasicEconomy', 'isNonStop',
'searchDate_weekend',
                   'flightDate_weekend', 'isHoliday', 'season', 'flightTiming',
'bookingProximity']

feature_columns = [
    col for col in feature_columns if train_data.select(col).distinct().count()
> 1
]

numeric_features = [col for col in feature_columns if
train_data.schema[col].dataType.typeName() != 'string']

# index and encode categorical features
```

```
indexers = [
    StringIndexer(inputCol=col, outputCol=col + "Index") for col in
feature_columns if col not in numeric_features
]
encoders = [
    OneHotEncoder(inputCol=col + "Index", outputCol=col + "Vector") for col in
feature_columns if col not in numeric_features
]

# assembling features into a vector
assembler = VectorAssembler(
    inputCols=[col + "Vector" for col in feature_columns if col not in
numeric_features] + numeric_features,
    outputCol="features"
)

# linear Regression model
lr = LinearRegression(labelCol="totalFare")

# create Pipeline
pipeline = Pipeline(stages=indexers + encoders + [assembler, lr])

paramGrid = (
    ParamGridBuilder()
    .addGrid(lr.regParam, [0.1, 0.01, 0.001])
    .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
    .addGrid(lr.maxIter, [10, 50, 100])
    .build()
)

evaluator = RegressionEvaluator(labelCol="totalFare",
predictionCol="prediction", metricName="rmse")

crossval = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=paramGrid,
    evaluator=evaluator,
    numFolds=3,
    parallelism=2
)

cv_model = crossval.fit(train_data)

# make predictions
predictions = cv_model.transform(test_data)

predictions = predictions.filter(~isnan("prediction") & ~isnan("totalFare"))

# evaluate the model
rmse = evaluator.evaluate(predictions)
```

```
r2 = evaluator.evaluate(predictions, {evaluator.metricName: 'r2'})

print(f"RMSE: {rmse}")
print(f"R-squared: {r2}")


predictions.select("flightDate", "totalFare", "prediction").show(10,
truncate=False)


# In[33]:


# transform the training data
transformed_train_data = model.transform(train_data)

# save the transformed data to the /trusted folder
transformed_train_data.write.parquet("gs://my-bigdatatech-project-jl/trusted/pr
ocessed_data", mode='overwrite')

print('transformed training data saved')

# save the LR model to the /models folder
model.save("gs://my-bigdatatech-project-jl/models/linear_regression_model")

print('LR model saved')


# In[ ]:


# Stop Spark session
spark.stop()
```

Appendix E: Data Visualization

Visualizing correlations between target variable and features to determine the relationship between target variable and features.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Create a Spark session
spark = SparkSession.builder \
    .appName("ML Pipeline") \
    .config("spark.executor.memory", "8g") \
    .config("spark.executor.cores", "4") \
    .config("spark.executor.instances", "4") \
    .config("spark.sql.shuffle.partitions", "100") \
    .getOrCreate()

spark.conf.set("spark.sql.debug.maxToStringFields", 1000)


# function to save plot to GCS
def save_fig(plt, img_name, img_type="png"):
    print(f"Saving {img_name}...")
    img_data = io.BytesIO()
    plt.savefig(img_data, format=img_type, bbox_inches='tight')
    img_data.seek(0)
    storage_client = storage.Client()
    bucket = storage_client.bucket("my-bigdatatech-project-jl")
    blob = bucket.blob(f"visualizations/{img_name}.{img_type}")
    blob.upload_from_file(img_data, content_type=f"image/{img_type}")
    print(f"{img_name} successfully saved to GCS!")

# data used for model
gcs_path = 'gs://my-bigdatatech-project-jl/trusted/processed_data/*'

# load data into pyspark df
df_spark = spark.read.parquet(gcs_path)

df_spark.printSchema()

from pyspark.ml.tuning import CrossValidatorModel

model_path = "gs://my-bigdatatech-project-jl/models/linear_regression_model"

cv_model = CrossValidatorModel.load(model_path)

# load pipelinemodel into variable
best_pipeline = cv_model.bestModel

# extract model
lr_model = best_pipeline.stages[-1]

print("Linear Regression model extracted successfully.")

from pyspark.sql.functions import col

# Outlier detection for 'totalFare' using IQR
quantiles = df_spark.approxQuantile("totalFare", [0.25, 0.75], 0.01)
Q1, Q3 = quantiles
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
```

```
df_spark = df_spark.filter((col("totalFare") >= lower_bound) &
(col("totalFare") <= upper_bound))

df_spark.select("totalFare").describe().show()

# totalFare
plt.figure(figsize=(8, 6))
sns.histplot(train_data.toPandas()['totalFare'], kde=True, color='blue',
bins=30)
plt.title("Distribution of Total Fare")
plt.xlabel("Total Fare")
plt.ylabel("Frequency")
plt.show()

# normal distribution with highest frequency total fare being <=500

# Load the saved predictions from the Parquet file
predictions =
spark.read.parquet("gs://my-bigdatatech-project-jl/predictions/predictions_data
")

# Show the predictions
predictions.show()

# In[37]:


# Residuals plot of predicted vs actual total fare
predictions_data = predictions.select("totalFare", "prediction").toPandas()
predictions_data['residuals'] = predictions_data['totalFare'] -
predictions_data['prediction']

plt.clf()
plt.figure(figsize=(10, 6))
sns.scatterplot(x=predictions_data['totalFare'],
y=predictions_data['residuals'], color='red')
plt.title("Residuals of Predicted vs Actual Total Fare")
plt.xlabel("Actual Total Fare")
plt.ylabel("Residuals")

plt.show()


# In[49]:

import numpy as np

# Sampling 1000 rows from the dataframe
small_df_spark = df_spark.limit(1000)
```

```
# use only numeric columns
small_df_pd = small_df_spark.toPandas()
numeric_cols = small_df_pd.select_dtypes(include=[np.number]).columns
corr_df_numeric = small_df_pd[numeric_cols].corr()

# create corr matrix
corr_matrix = corr_df_numeric.corr()

plt.figure(figsize=(12, 8))
sns.heatmap(corr_df_numeric, annot=True, cmap='coolwarm', fmt='.2f',
linewidths=0.5)
plt.title("Correlation Matrix")
save_fig(plt, "correlation_matrix")
plt.close()

# In[44]:


import matplotlib.pyplot as plt
import numpy as np

lr_model_from_pipeline = cv_model.bestModel.stages[-1]

coefficients = lr_model_from_pipeline.coefficients.toArray()


feature_names = ['isRefundable', 'isBasicEconomy', 'isNonStop',
'searchDate_weekend',
                 'flightDate_weekend', 'isHoliday', 'season', 'flightTiming',
'bookingProximity']

feature_importance = list(zip(feature_names, coefficients))

feature_importance.sort(key=lambda x: abs(x[1]), reverse=True)

sorted_feature_names, sorted_coefficients = zip(*feature_importance)

plt.figure(figsize=(10, 6))
plt.barh(sorted_feature_names, sorted_coefficients, color='skyblue')
plt.xlabel('Coefficient Value')
plt.title('Feature Importance in Linear Regression')
plt.show()

# get hyperparameters from LR model
print("Best Model Parameters:")
for param, value in lr_model.extractParamMap().items():
    print(f"{param.name}: {value}")

# Stop Spark session
spark.stop()
```