Mark Pedraza

CIS 4130

Richard Holowczak

9-28-24

Proposal

The data set I will choose for this project is titled, "Flight Prices." Below is a description of this data set and where you can find it, the columns associated with the data set, the variable that will be predicted, and the machine learning model that will be used.

Description

This data set contains data about purchasable tickets from Expedia between April 2022 and October 2022. The airports these tickets go to and from are DFW, JFK, DEN, LGA, EWR, ATL, OAK, ORD, PHL, BOS, IAD, DTW, LAX, CLT, SFO, MIA. The data set is formatted as a CSV file and each row is a separate ticket. In total, there are 27 columns, and the size of the data set is 31.09 GB. You can find the data set along with descriptions for each column here: https://www.kaggle.com/datasets/dilwong/flightprices

Columns

- 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

Prediction

I will predict the *totalFare* column. To predict this, I will use the columns startingAirport, destinationAirport, travelDuration, isBasicEconomy, seatsRemaining, and totalTravelDistance. The machine learning model that will be used will be Linear Regression.

<u>Data Acquisition</u>

Below are the steps that I took to download the data set and copy it into a new bucket. For specific codes examples for relevant steps, refer to *Appendix A* below.

1. Download API token from Kaggle site. We can find this under settings in our profile for Kaggle.

2. Load into SSH in the instance for compute engine. Create the folder, ".kaggle/", to store the downloaded Kaggle token.

3. Upload the token using the "UPLOAD FILE" option on the top right of the SSH window.

4. Move that token into the folder and then make the folder secure.

5. Download pip. This will allow us to download Kaggle commands later.

6. Create a python dev environment.

7. Activate the dev environment.

8. Download Kaggle command-line tools.

9. Go to dataset on Kaggle and copy the API command. Then use it inside the instance SSH to download the data set.

10. Download unzip and unzip the file.

11. Create a bucket for this project. But before that, authenticate. Without it, bucket creation is stopped.

12. Copy the unzipped file into the new bucket and in the process make a folder titled "landing". This folder is where the unzipped file will be stored inside the bucket.

13. Make other folders in the bucket. I used the UI in the storage interface of the site instead of the command-line for this. The new folders will be titled cleaned, code, models and trusted. (*Figure 1*).

14. Now everything is done. We can check if the downloaded data is there through the buckets interface. For this project, the unzipped file is called itineraries.csv (*Figure 2*).

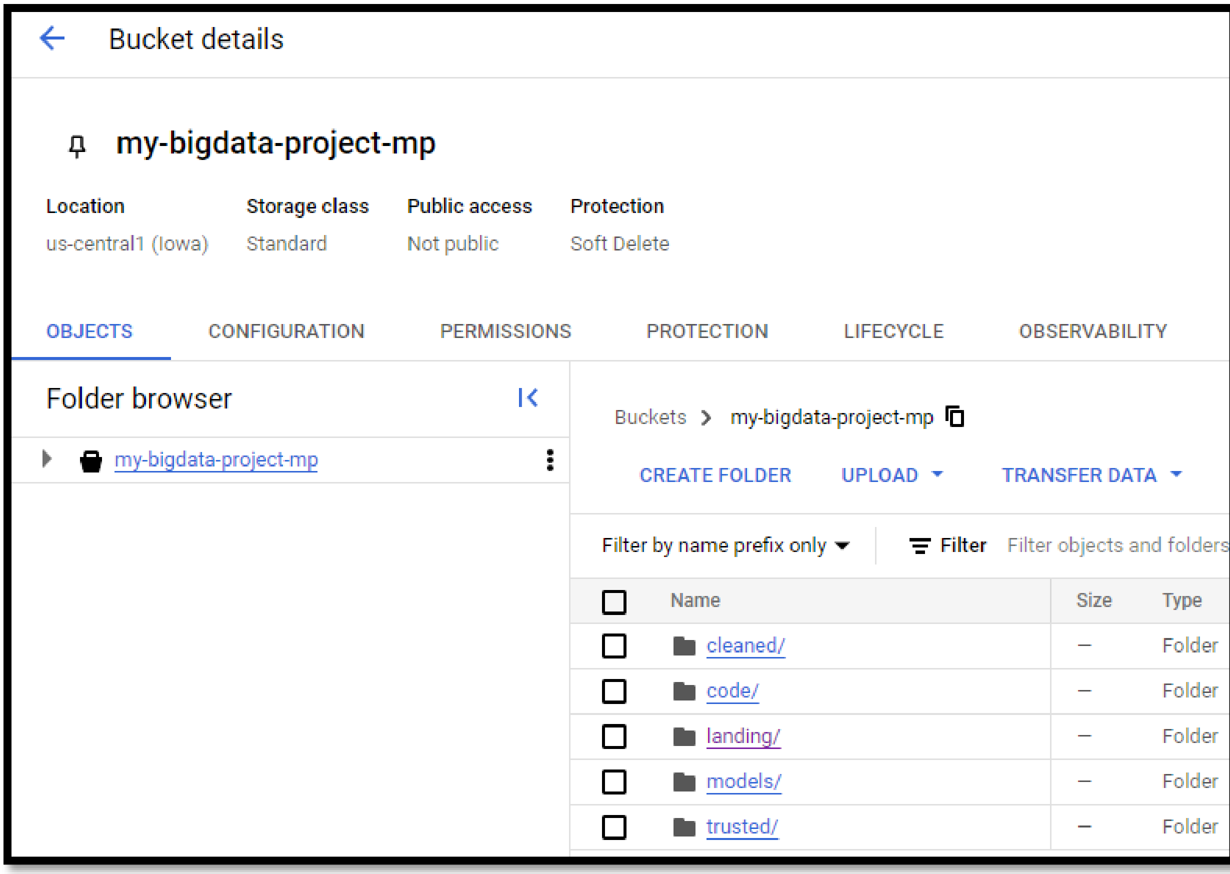


Figure 1- Folders in bucket

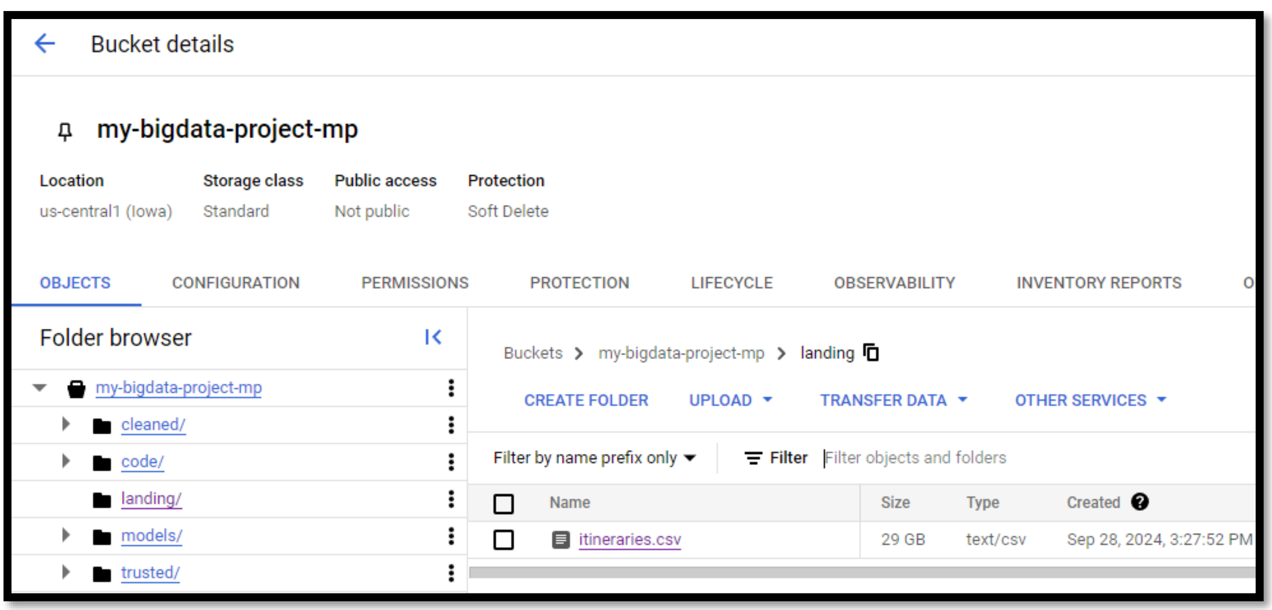


Figure 2- Downloaded data

Exploratory Data Analysis and Data Cleaning

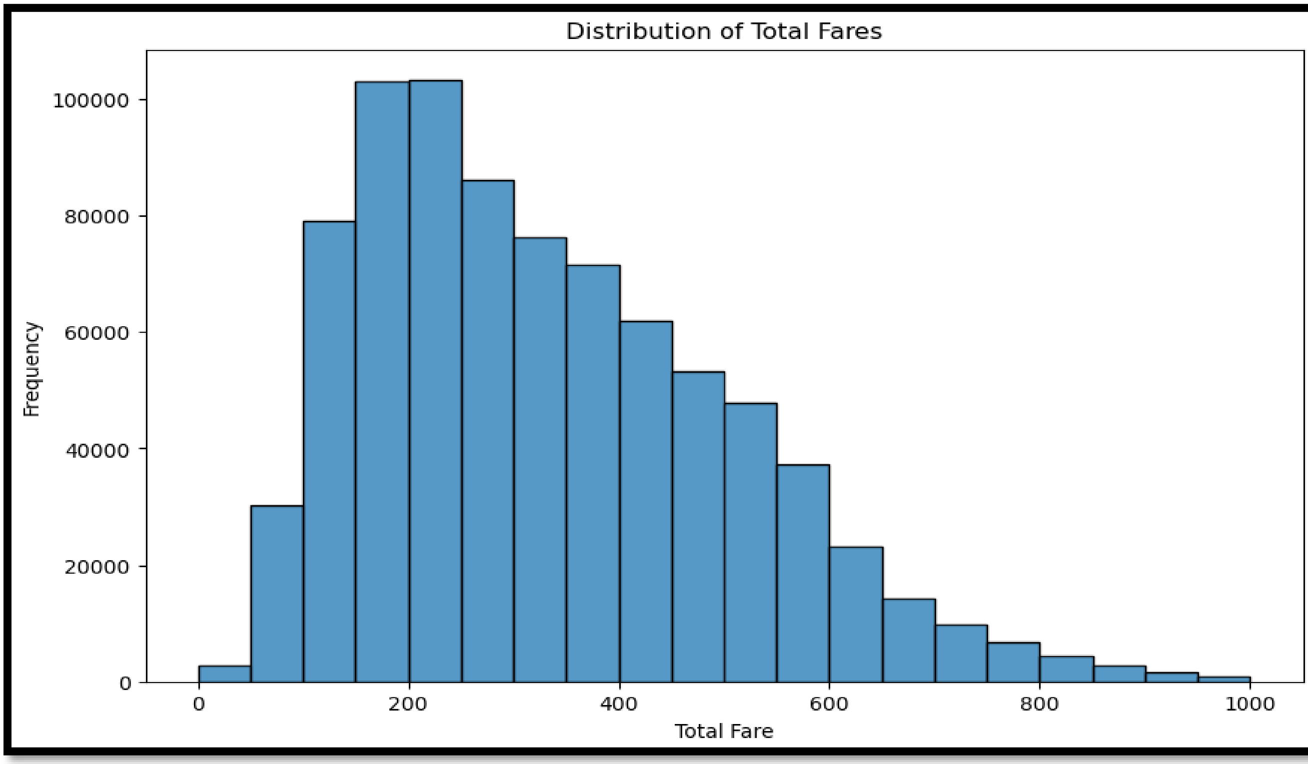
In this section, we will go over the data and try to understand it at a deeper level. We will do this through aggregating numeric columns, date columns, creating graphs such as histograms or scatter plots, and more. Note that no outliers have been removed for any graphs below.

Code for Exploratory Data Analysis

I used Google DataProc, PySpark and Jupiter Notebook to complete this section. With my data being 30 GB in size, a single compute would not suffice. To solve this, DataProc with PySpark was needed with a couple workers in my cluster. The library 'HandySpark' was used to count nulls within all columns. Although it has more functions to allow for PySpark DataFrames to be treated like Pandas DataFrames, and thus easier data visualization, it tends to be buggy. Instead, I loaded the entire dataset into a PySpark DataFrame, then sampled 1% of that DataFrame into a Pandas DataFrame. 1% sounds small, but there are roughly 82 million records in this dataset, so 1% turned out to be 8,200,000 records which is a good sample. This way, I can use Pandas and Matplotlib to visualize the data. Below, there are a few graphs that we created. More graphs are located at the end of *Appendix B*.

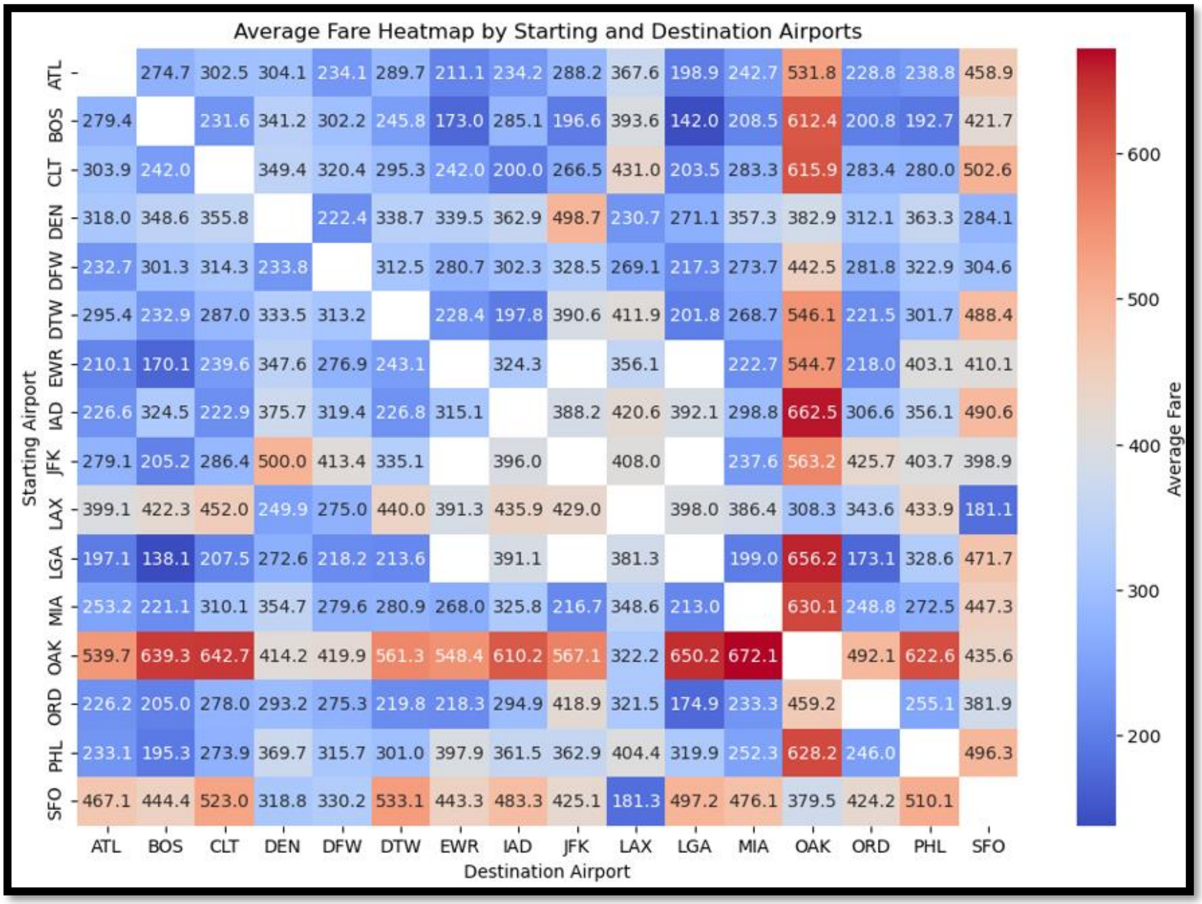
Visual 1 – Distribution of Total Fares

Below is a graph that showcases the distribution of “totalFares” (price of a plane ticket after taxes and fees.) Most tickets seem to cost around $200 then drop off in price from there. Outliers here are hard to notice since they are placed in the same bin at the end.



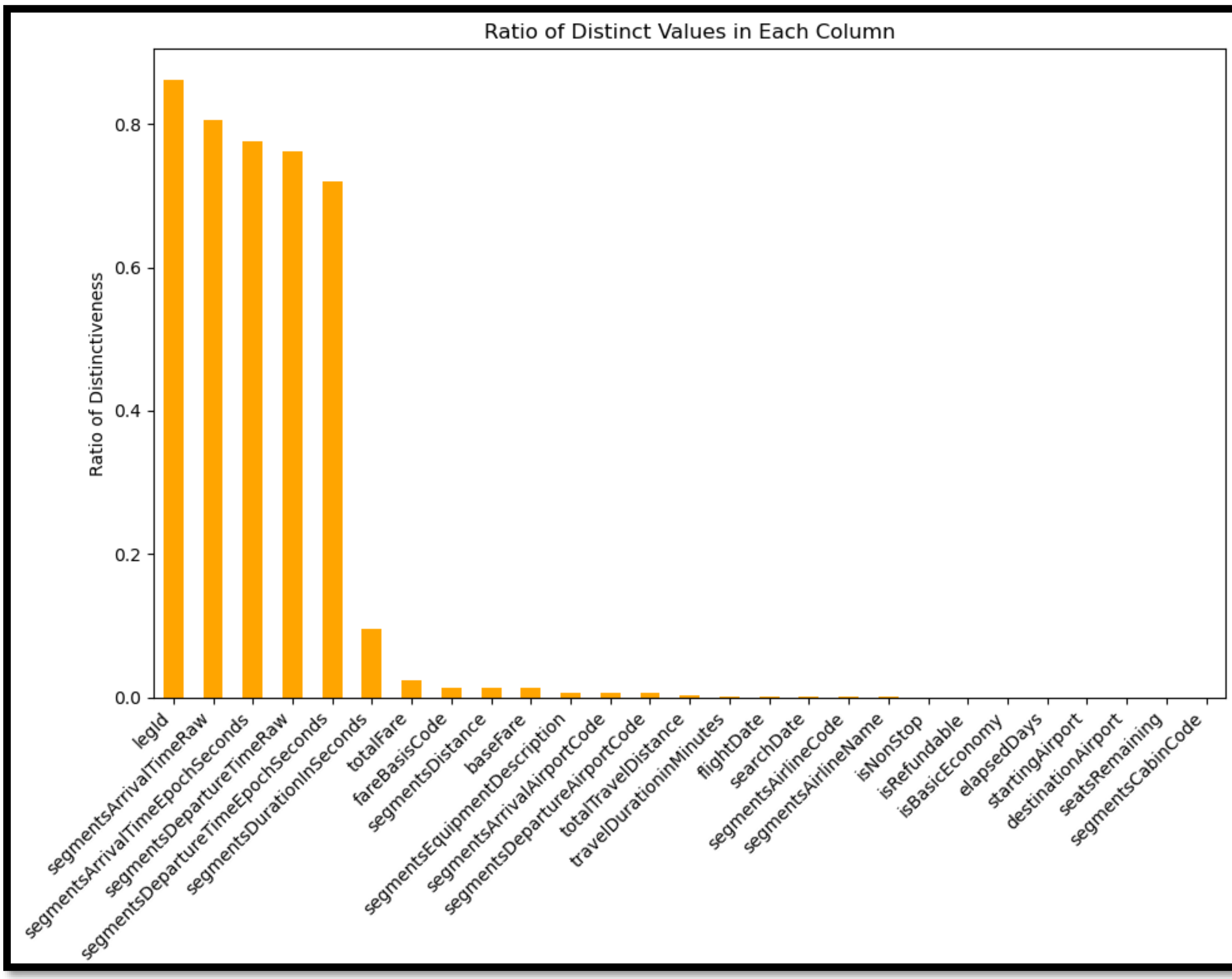
Visual 2 – Average Fare Heatmap by Starting and Destination Airports

Below, we have another visualization. This graph is a matrix that gives the average price of a ticket based on the tickets starting airport and ending airport. Here, you can see that starting at the IAD airport and ending at the OAK airport is particularly expensive! But it is important to note, this data is a sample from the total set and could contain outliers. So, on repeat executions of the code, this may potentially look different.



Visual 3 - Ratio of Distinct Values in Each Column

This graph shows the ratio of how "unique" each column is. Essentially, each unique value in a column is counted, then the sum of that is divided by the total amount of values for that column. This gives us a ratio of how many unique values are in a column. Since we are looking to create a machine learning model, knowing this will be useful. Columns that are too unique don't hold any predictive value, so they will either be discarded or transformed.



Visual 4 – General Statistics

There are a few other graphs, but this section would be bloated if they were all included. I do, however, want to include the out of the pandas describe function, which shows many statistics on both numeric and date columns. You will notice for certain columns such as "totalTravelDistance" or "totalFare", that there are extreme values for the max when compared where most of the data is. This is also noticeable when we look at the mean. So, there are quite a few outliers that will be removed when the data is finally cleaned.

	searchDate	flightDate
count	822423	822423
mean	2022-07-13 17:22:05.208560640	2022-08-09 14:41:18.493913600
min	2022-04-16 00:00:00	2022-04-17 00:00:00
25%	2022-06-04 00:00:00	2022-07-01 00:00:00
50%	2022-07-15 00:00:00	2022-08-14 00:00:00
75%	2022-08-23 00:00:00	2022-09-18 00:00:00
max	2022-10-05 00:00:00	2022-11-19 00:00:00

	totalTravelDistance	travelDurationinMinutes
count	761616.000000	822423.000000
mean	1609.485398	427.829745
min	89.000000	0.000000
25%	878.000000	262.000000
50%	1463.000000	409.000000
75%	2415.000000	566.000000
max	7252.000000	1436.000000
std	857.471086	224.242696

	elapsedDays	baseFare	totalFare	seatsRemaining
count	822423.000000	822423.000000	822423.000000	822423.000000
mean	0.149152	292.659001	340.384564	5.971941
min	0.000000	0.010000	19.590000	0.000000
25%	0.000000	159.000000	197.100000	4.000000
50%	0.000000	260.470000	305.580000	7.000000
75%	0.000000	398.140000	452.100000	9.000000
max	2.000000	7000.000000	7548.600000	10.000000
std	0.356266	182.864964	195.689682	2.881598

Summary of data and predictions for feature engineering

In short, the data is mostly clean, but data cleaning was still needed. Of the 27 columns, 5 are numeric. However, there is one named “travelDuration”, that although a string, could be converted to a more concise unit of time with some tweaking, and be turned numeric as well. There are also only 2 columns with missing data. These are “travelDuration”, and “segmentsEquipmentDescription”. All records with a null value will be dropped. In the future the nulls may be treated differently. For “travelDuration” (once converted to numeric) we could turn nulls into the mean of the column. For “segmentsEquipmentDescription”, nulls may be replaced with the most frequent value. This would allow us to still use these records.

For feature engineering, I will likely have to drop many columns as they will be difficult to encode. Specifically, the column’s titled “segments” have multiple data points inside of each record. For example, segmentsDistance can have “541 || 1473”, “None || None”, “399 || 1549”, and many other combinations. These types of columns appear very hard to feed into a ML model. So, for now, they will be dropped.

For the source code of the exploratory data analysis file and for the cleaning file, they will be stored in *Appendix B*, and *Appendix C* respectively.

Feature Engineering and Modeling

Below is a screenshot of an excel sheet that displays the columns in the dataset we will be using, its data type, and the type of feature engineering treatment it will undergo. For most machine learning capabilities, I utilized the PySpark machine learning library.

Columns	Data Type	Feature Engineering Treatment		
searchDateMonth	integer	Scalar	VectorAssembler	
searchDateDay	integer	Scalar	VectorAssembler	
searchDateIsWeekend	double			
flightDateMonth	integer	Scalar		
flightDateDay	integer	Scalar		
flightDateIsWeekend	double			
startingAirport	string	StringIndexer	One-Hot Encoder	
destinationAirport	string	StringIndexer	One-Hot Encoder	
fareBasisCode	string	StringIndexer	One-Hot Encoder	**VectorAssembler**
isBasicEconomy	boolean	Binarizer		Features
isRefundable	boolean	Binarizer		
isNonStop	boolean	Binarizer		
elapsedDays	integer	Scalar	VectorAssembler	
seatsRemaining	integer	Scalar	VectorAssembler	
totalTravelDistance	integer	Scalar	VectorAssembler	
travelDurationMinutes	integer	Scalar	VectorAssembler	
segmentsArrivalAirportCode	string	StringIndexer	One-Hot Encoder	
segmentsDepartureAirportCode	string	StringIndexer	One-Hot Encoder	
segmentsAirlineName	string	StringIndexer	One-Hot Encoder	
segmentsAirlineCode	string	StringIndexer	One-Hot Encoder	
segmentsEquipmentDescription	string	StringIndexer	One-Hot Encoder	
segmentsDistance	string	StringIndexer	One-Hot Encoder	
segmentsCabinCode	string	StringIndexer	One-Hot Encoder	
searchDateDayOfWeek	integer	Scalar	VectorAssembler	
flightDateDayOfWeek	integer	Scalar	VectorAssembler	

Summary of feature engineering and modeling

The main steps of the feature engineering script and modeling are as follows.

1. Get the data and drop highly unique columns (“legId”) as well as highly correlated columns (“baseFare”)
2. Transform column “travelDuration” into “travelDurationMinutes”. It transforms the column into a single unit of measurement which is minutes.
3. Extract features from the data columns “searchDate” and “flightDate”. For each date column, we will extract the month, day, and a boolean is_weekend, as features.
4. We will create a pipeline that includes everything from points 5 to 9.
5. Index appropriate columns.
6. Encode the indexed columns.
7. Create an assembler for all numerical columns, then scale the vectors outputted.
8. Create and assembler for all remaining vectors and scaled features. The output column will be “features”.
9. Create the linear regression model. Our label column will be “totalFare”.
10. Put everything from 5-9 in a pipeline.
11. Use the pipeline on our data.
12. Create a random train/test split from the data.
13. Instantiate an evaluator and a grid.
14. Create a cross validator with our pipeline, grid, and evaluator. We set the fold to 6.
15. Fit our cross validator to our training data and derive metrics from all the models.
16. Get the best model and evaluate it to find the RMSE and R-squared.
17. Finally, save our model to /model in our bucket and our data to /trusted in our bucket.

For feature engineering, there were a couple challenges when starting. Initially, I wanted to drop many of the columns labeled “segment”, as they appeared to be too unique between one another, thus providing no predictive power. However, after expanding on the exploratory data analysis, I found that many of these columns did not hold many unique values. You can see this in *Visual 4* of the “Exploratory Data Analysis and Data Cleaning” section. Of the 12 segments columns, only 4-5 have high uniqueness, like the identifier legid. So, I could still use the other columns for our model. However, I may still be able to use those 4-5 columns if we split them up, then do our feature treatment. Also, with the dataset being so large, a small sample of the set has to be used for it to take a reasonable amount of time. Only 0.5% of the data was used and it still took nearly 50 minutes to run the script. The training took most of that time, with the pipeline creation coming in second.

All the code for the feature engineering and model creation can be found in *Appendix D*.

The average metrics for all the models that were trained is roughly 58.72. The code is below.

```
# Show the average performance over the six folds
print(f"Average metric {all_models.avgMetrics}")
```

```
Average metric [58.7163687900069]
```

The RSME and R-squared of the best model among all models trained were 57.7 and 0.88 respectively. The code is below along with a side by side of the total fare and the model's prediction.

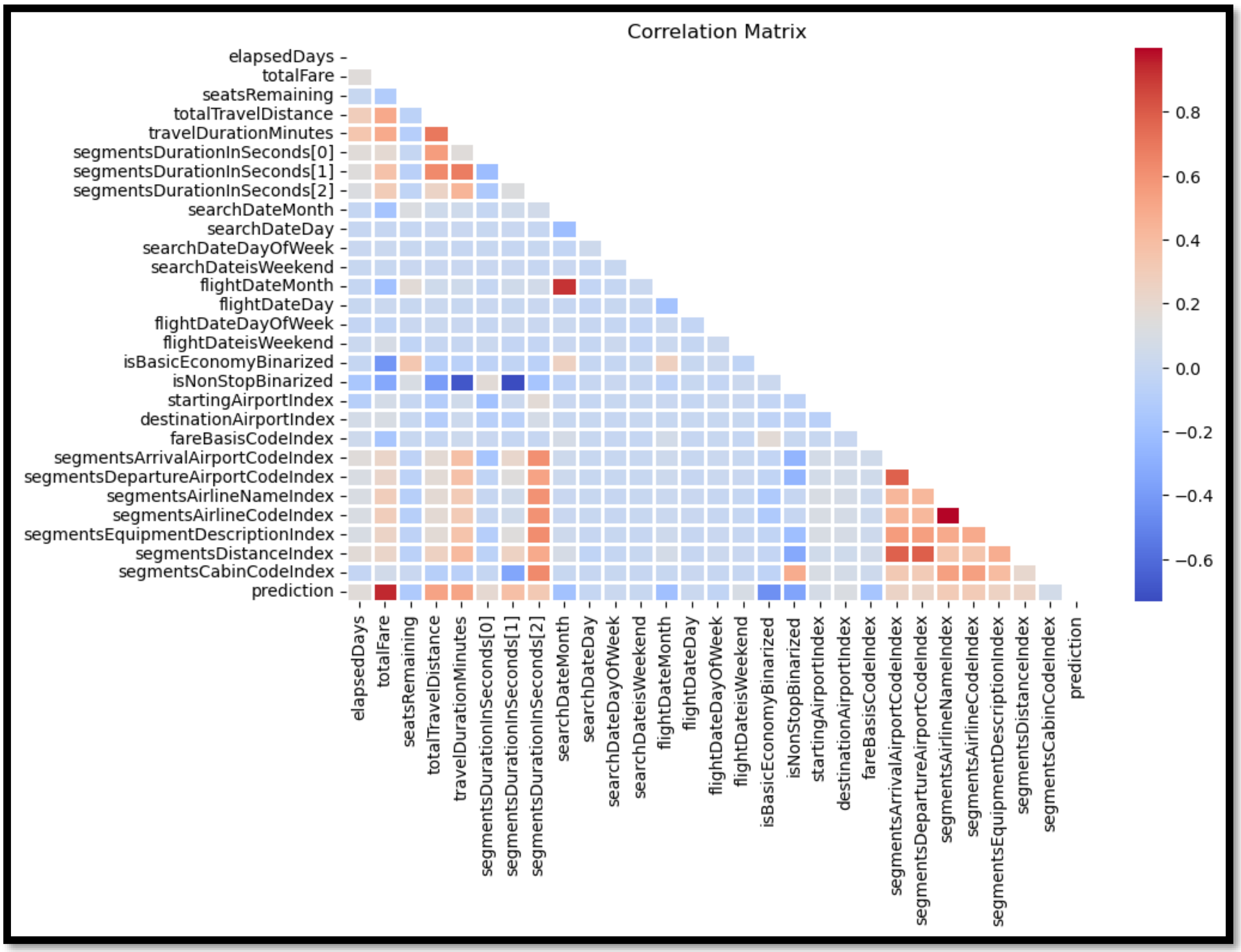
```
# Get the best model from all of the models trained
bestModel = all_models.bestModel

# Use the model 'bestModel' to predict the test set
test_results = bestModel.transform(testData)

# Show the predicted totalFare
test_results.select('totalFare', 'prediction').show(truncate=False)

# Calculate RMSE and R2
rmse = evaluator.evaluate(test_results, {evaluator.metricName:'rmse'})
r2 =evaluator.evaluate(test_results,{evaluator.metricName:'r2'})
print(f"RMSE: {rmse}  R-squared:{r2}")
```

```
RMSE: 55.019262513041525  R-squared:0.8873606113076086
+---------+------------------+
|totalFare|prediction        |
+---------+------------------+
|457.6    |335.6070943229785 |
|331.6    |236.161544762181  |
|181.6    |165.59969797758777|
|463.7    |504.63027023209065|
|228.6    |213.32221487097178|
|241.6    |233.97993287584342|
|281.6    |445.0608382548361 |
|128.6    |129.632597412518  |
|212.6    |227.7614131802248 |
|588.6    |478.7053221167554 |
|317.6    |340.5538972065891 |
|571.6    |600.2815471526486 |
|178.6    |129.72774665694737|
|198.6    |230.55948983520022|
|378.6    |393.5438723331017 |
|208.6    |230.3580659377818 |
|497.6    |445.7897225970332 |
|767.6    |781.7053092850845 |
|597.2    |433.913655429527  |
|317.6    |310.4454567379491 |
+---------+------------------+
```

Model Evaluation and Data Visualization

This section will go over some visualizations that were created on the newly trained model and the data the model used. This data includes all features used for the model. Model hyperparameters are also included here. The source code for this section is in *Appendix E*.

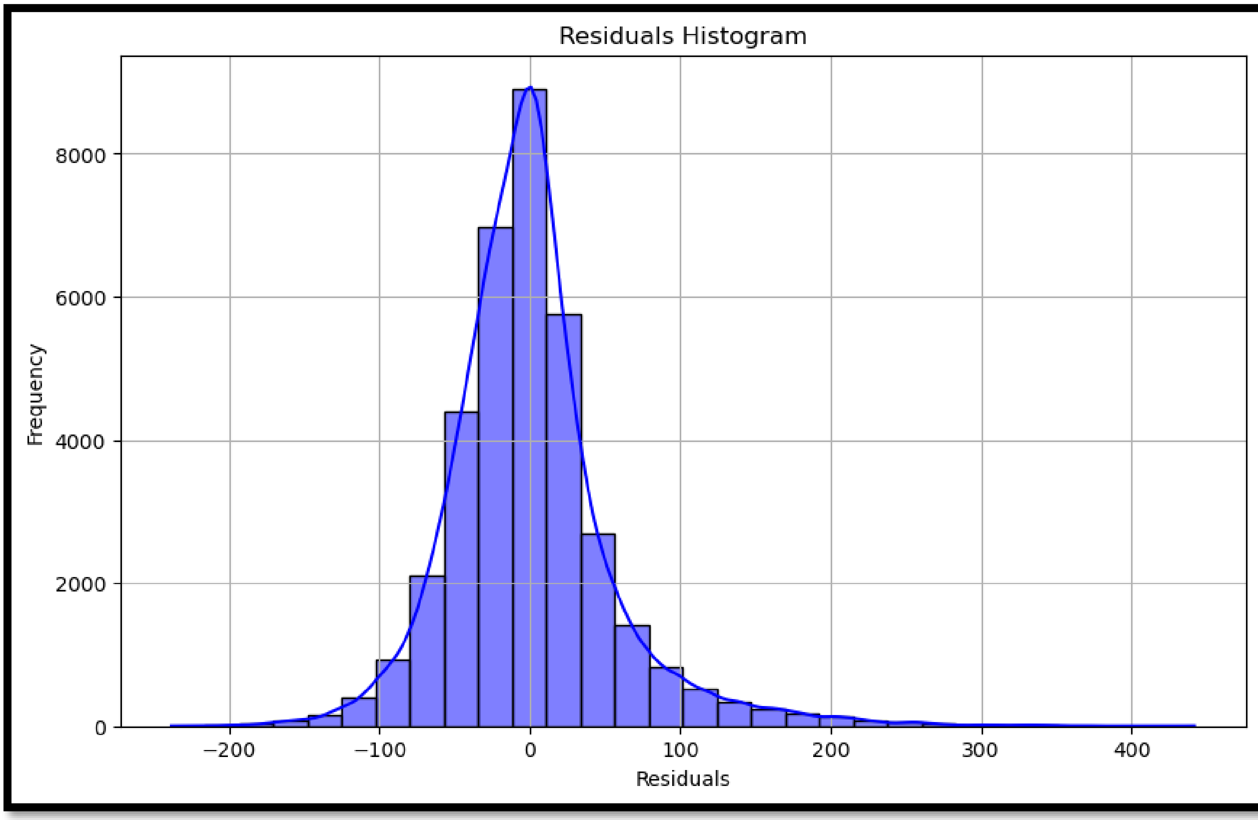
Visual 1 – Correlation of Columns, Post Feature Engineering and Model Creation.



This is a correlation matrix of all our columns post feature engineering and model creation. All columns are present here besides “baseFare” and “isRefundableBinarized”.

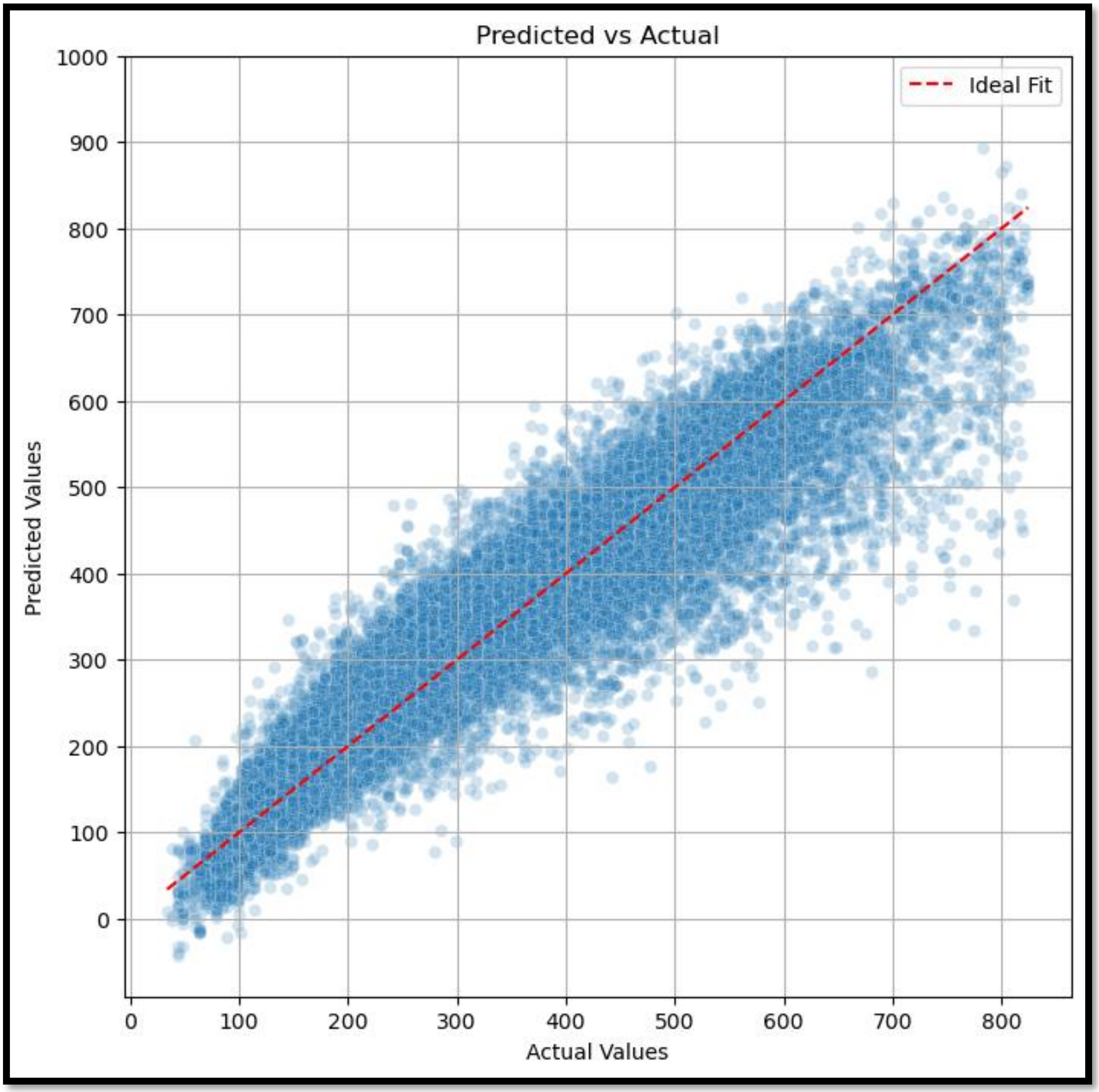
For the most part, all the features that were included in the model were not correlated at all. But there were still a few correlated variables that slipped through the cracks, as this information could only be found out *right before* we would create our model (At least for the most part). This means that the model could actually be tuned further by dropping some of these features. But we will continue visualizations despite this for now.

Visual 2 – Residuals Histogram.



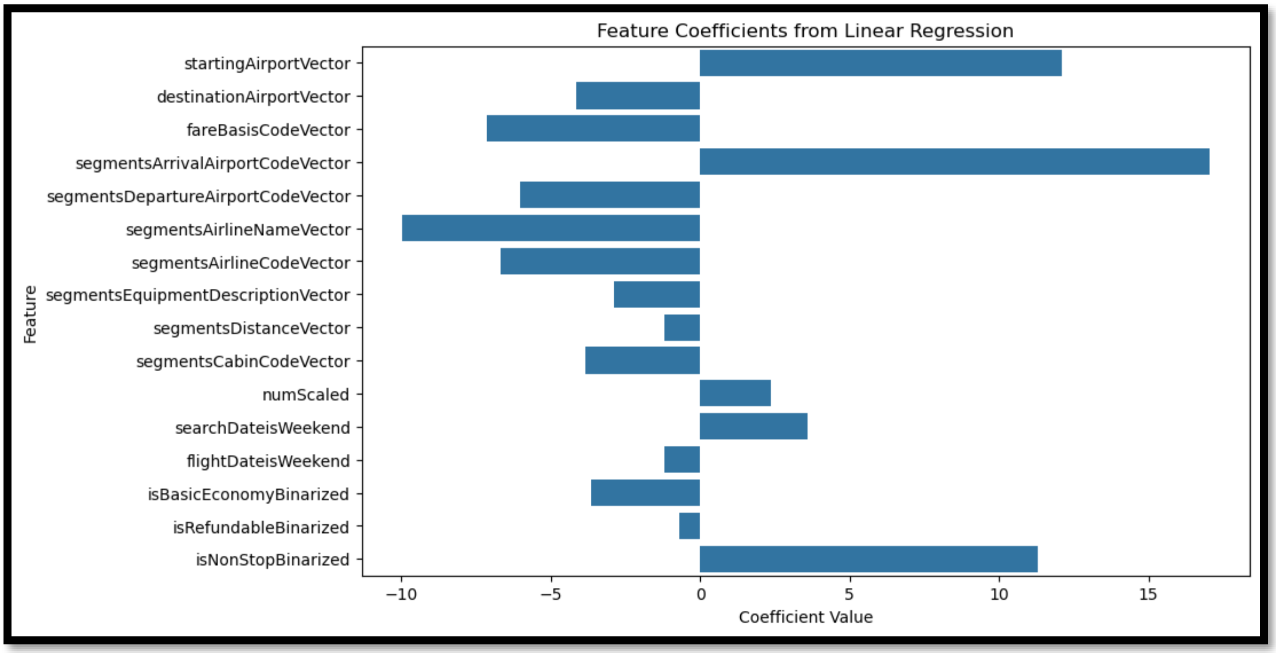
Here is a histogram of the residuals. Residuals are the difference between the actual and predicted values and should follow a uniform distribution when plotted. As the graph shows, our model contains this distribution. This essentially tells us that the model generated is “valid”. That meaning, our model has generated an acceptable random error, so we can continue to use it confidently.

Visual 3 – Scatter plot of Predicted vs Actual.



In this scatter plot, we have the actual (x) vs the predicted (y) values. If our model predicted perfectly, then all points would rest on the red line. So, if the predicted value is 100, then the actual value should be 100. But obviously, our model isn't perfect, and there are some outliers. Although for the most part, this model is good at predicting the label column ("totalFare"). Most of the points follow the direction of the "ideal fit" line, and many are close to it.

Visual 4 – Bar Chart of Features and their Coefficients.



This is a bar chart of the feature columns along with their coefficients. The order of magnitude for each coefficient dictates how much of an effect it has on our linear regression model. For example, segmentsArrivalAirportCodeVector has a significant impact on our linear regression model, while isRefundableBinarized seems to have a much smaller impact. For the numerical columns, they are currently all grouped together in the numScaled vector. Unfortunately, it can be difficult to connect the coefficients of those numerical features to their coefficients. It may still be possible, but it would require more time. I also expected the numScaled to have a higher coefficient, but surprisingly it does not.

Visual 5 – Linear Regression Model Hyperparameters

```
Best Model Parameters:
aggregationDepth: 2
elasticNetParam: 0.0
epsilon: 1.35
featuresCol: features
fitIntercept: True
labelCol: totalFare
loss: squaredError
maxBlockSizeInMB: 0.0
maxIter: 100
predictionCol: prediction
regParam: 0.0
solver: auto
standardization: True
tol: 1e-06
```

Here are the hyperparameters of our model. Remember that these parameters weren't specifically created by me, they were created from our cross validator which is in *Appendix D*. With that in mind, these are the hyperparameters that gathered the best results. Although this is not a graph like other visualizations, it helps us fully see what our model looks like, beyond what it used to train with.

Conclusion

This project took quite a length of time, and the report is now technically above 60 pages, (although excluding the Appendixes which is mostly code, it's down to about 20 pages). So, let's go through a quick summary of what we did here to refresh our minds, and to conclude this project.

We started by searching for a dataset that can be utilized for machine learning. After some searching, we found a good data set about flights with quite a few useful features, including the prices of flights. With this data, we can use the features to predict flight prices. Of which, a linear regression algorithm would be best suited for, as the label column (flight prices) would be continuous. So, using Google Cloud Storage, we extracted our data through the Linux command line of a compute instance. Then, we loaded that data into a bucket and did some exploratory data analysis to understand it. Afterwards, we use that newfound knowledge to clean and transform the data, so it's suited to be fed into a linear regression model. With our clean and transformed data, we created a cross validator that trained a lot of models with different hyperparameters to find the best one. Once it did, we extracted the best model and evaluated it. It ended up having good metrics, with an RMSE of 55 and R^2 of 0.89. Finally, we did some visualizations with this new model to see other metrics about it and to also understand why it's getting those metrics. Broadly speaking, this is everything that we did. So, what conclusions can we draw about our data now that this project is complete?

To start, flight prices are heavily affected by starting airport and segment airports. If a flight is nonstop, it also heavily impacts flight prices. Interestingly though, numerical features *as a whole* do not contribute much to flight prices. Some numerical features include (but are not limited to) elapsedDays, seatsRemaining, travelDurationMinutes, searchDateDayOfWeek, etc. Although, this does *not* mean that numerical features don't have a large impact, it's just difficult to see the impact of individual numeric features, as they all were essentially "grouped" together. And, as a whole, they do not contribute much.

In the end, we have a well trained and tuned model. Technically, there could still be more work done to further fine tune this, but a line must be drawn somewhere. As it stands, this is more than enough to understand the entire process of creating a pipeline and using it to train a machine learning model. This project was nothing short of fun, and it helped me understand and enjoy the effort it takes to make a good model. Below is the GitHub link for this project.

GitHub URL for this project: [https://github.com/markpedraza/flight-prices-ML-project]

Appendix A

These are screenshots of the code used in the respective steps for the Data Acquisition page (So "2)" in this appendix showcases code for step 2 in the Data Acquisition page).

2) – Kaggle folder creation

```
markpedraza645@instance-1:~$ mkdir .kaggle
```

4.a) – move token to folder

```
markpedraza645@instance-1:~$ mv kaggle.json .kaggle/
```

4.b) – secure folder

```
markpedraza645@instance-1:~$ chmod 600 .kaggle/kaggle.json
```

5) - install pip

```
markpedraza645@instance-1:~$ sudo apt -y install python3-pip python3.11-venv
```

6) – create python dev environment

```
markpedraza645@instance-1:~$ python3 -m venv pythondev
```

7) – activate python environment

```
markpedraza645@instance-1:~$ cd pythondev
markpedraza645@instance-1:~/pythondev$ source bin/activate
(pythondev) markpedraza645@instance-1:~/pythondev$
```

8) – install Kaggle commands

```
(pythondev) markpedraza645@instance-1:~/pythondev$ pip3 install kaggle
```

9) – download flight prices dataset

```
(pythondev) markpedraza645@instance-1:~/pythondev$ kaggle datasets download -d dilwong/flightprices
```

10.a) - install zip

```
(pythondev) markpedraza645@instance-1:~/pythondev$ sudo apt install zip
```

10.b) - unzip the file

```
(pythondev) markpedraza645@instance-1:~/pythondev$ unzip flightprices.zip
```

11.a) – create a new bucket

```
(pythondev) markpedraza645@instance-1:~/pythondev$ gcloud storage buckets create
 gs://my-bigdata-project-mp --project=cis4130-flight-prices-project --default-
storage-class=STANDARD --location=us-central1 --uniform-bucket-level-access
Creating gs://my-bigdata-project-mp/...
```

11.b) – authenticate login

```
(pythondev) markpedraza645@instance-1:~/pythondev$ gcloud auth login

You are running on a Google Compute Engine virtual machine.
It is recommended that you use service accounts for authentication.

You can run:

  $ gcloud config set account `ACCOUNT`

to switch accounts if necessary.

Your credentials may be visible to others with access to this
virtual machine. Are you sure you want to authenticate with
your personal account?

Do you want to continue (Y/n)?  y
```

12) – copy unzipped file to new bucket

```
(pythondev) markpedraza645@instance-1:~/pythondev$ gcloud storage cp itineraries
.csv gs://my-bigdata-project-mp/landing/
WARNING: Parallel composite upload was turned ON to get the best performance on
uploading large objects. If you would like to opt-out and instead
perform a normal upload, run:
`gcloud config set storage/parallel_composite_upload_enabled False`
If you would like to disable this warning, run:
`gcloud config set storage/parallel_composite_upload_enabled True`
Note that with parallel composite uploads, your object might be
uploaded as a composite object
(https://cloud.google.com/storage/docs/composite-objects), which means
that any user who downloads your object will need to use crc32c
checksums to verify data integrity. gcloud storage is capable of
computing crc32c checksums, but this might pose a problem for other
clients.

Copying file://itineraries.csv to gs://my-bigdata-project-mp/landing/itineraries
.csv
  Completed files 32/1 | 29.0GiB/29.0GiB | 158.3MiB/s

Average throughput: 149.1MiB/s
```

Appendix B

Source code for Exploratory Data Analysis file.

```
#!/usr/bin/env python

# coding: utf-8


# # Exploratory Data Analysis


# In[1]:


# This package will be used for data visualization. It must first be installed then imported. Uncomment to install.

#!pip install handyspark


# In[2]:


import numpy as np

from matplotlib import pyplot as plt

import pandas as pd

import seaborn as sns

np.bool = np.bool_


# In[3]:


from handyspark import *


# # Creating the Spark DataFrame iterneraries.csv and doing simple functions
```

```
# In[4]:


# Url for flight data
url = "gs://my-bigdata-project-mp/landing/itineraries.csv"

# Create a Spark DataFrame from our csv
df = spark.read.csv(url, header=True, inferSchema=True)


# In[5]:


# Check the columns and their data types
df.printSchema()


# In[6]:


# Count how many records are here
num_records = df.count()
print("This dataset contians {} records.".format(num_records))


# # Using HandySpark to count nulls is all columns


# In[7]:


# Create a HandySpark DataFrame from our current DataFrame
hpdf = HandyFrame(df)
```

In[8]:

```
# Count nulls in each column
hpdf.isnull()
```

#

Transform the "travelDuration" column into "travelDurationInMinutes"

#

- We will do this before the Data visualization step to help up derive infomation about how travel duration affects other variables.

In[9]:

```
# We will extract the hour and minutes from travelDuration and combine them into a new column called travelDurationinMinutes

#We will use these functions to extract the numbers from the strings in travelDuration
from pyspark.sql.functions import regexp_extract, col, when, expr

# Define regex patterns to capture hours and minutes
hours_pattern = "PT(\\d+)H"       # Captures the digits before 'H' in the "PT#H" format
minutes_pattern = "H(\\d+)M"      # Captures the digits before 'M' in the "#M" format after 'H'
only_minutes_pattern = "PT(\\d+)M" # For cases with only minutes (e.g., "PT20M")

# Extract hours and minutes, converting to integers
df_extracted = df \
    .withColumn("hours", regexp_extract(col("travelDuration"), hours_pattern, 1).cast("int")) \
    .withColumn("minutes", when(col("travelDuration").rlike(only_minutes_pattern),
```

```
                    regexp_extract(col("travelDuration"), only_minutes_pattern, 1))
            .otherwise(regexp_extract(col("travelDuration"), minutes_pattern, 1)).cast("int"))

# Calculate total minutes
df_with_total_minutes = df_extracted.withColumn(
    "travelDurationinMinutes",
    expr("coalesce(hours, 0) * 60 + coalesce(minutes, 0)")
)

# Get these new columns into our df, then drop the two unnecessary columns
df = df_with_total_minutes

# Compare travelDuration and travelDurationInMinutes to make sure the values are correct
df = df.drop("hours","minutes")
df.select("travelDuration","travelDurationinMinutes").sample(withReplacement=False,
fraction=0.05).show(truncate=False)

# Finally, drop travelDuration as it is no longer useful
df = df.drop("travelDuration")

# # Create Pandas DataFrame from a sample and do some data visualization

# In[10]:

#Take a sample from our Spark DataFrame and transform it into a Pandas DataFrame.

sample_percentage = 0.01 # 1% of the dataset represents around 820,000 rows.
```

```
sample_spark_df = df.sample(withReplacement=False, fraction=sample_percentage)

pdf = sample_spark_df.toPandas()

pdf.tail(3)


# In[11]:


numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']


pdf_numeric = pdf.select_dtypes(include=numerics)

pdf_numeric.head(10)


# In[12]:


#This function will be used for visualization in the next cell


#Function that looks at a pandas dataframe and checks how much of a column is "distinct".

#Then it creates a ratio of how unique each value is in that column is and creates a new dataframe from that data.

#

#ARG1: Pandas dataframe that you want to check

def distinct_ratio(pandas_df, sort=False):


    columns = ["name","distinctCount","totalCount","ratioDistinctTotal"]

    data = []
```

```
    for col_name in pdf.columns:
        distinct = pdf[col_name].nunique()#Number of distinct values in the column
        total = len(pdf[col_name])#Total number of values in the column
        ratio = round(distinct/total, 4) #A numer that is close to 1 is useless for ML. Vice versa for better.

        data.append([col_name,distinct,total,ratio])#Append everything we have to data that we will later add to a new dataframe



    distinct_df = pd.DataFrame(data, columns=columns) #create a new dataframe with our generated data
    distinct_df = distinct_df.set_index(columns[0]) # set "name" as the column

    #sorts dataframe by the ratio if requested
    if sort:
        distinct_df.sort_values('ratioDistinctTotal', ascending=False, inplace=True)

    return distinct_df

# # Visualizing Data

# In[ ]:

# Visualization of how "distinct" each column is.

# Essentially shows how useful each column is for ML. A value close to 1 is bad, as more values are unqiue and useless.
```

```
#Get distinct data for this graph
distinct_df = distinct_ratio(pdf, sort=True)

#Change size of graph
plt.figure(figsize=(10, 8))

distinct_df["ratioDistinctTotal"].plot(kind="bar", color="orange")

# Add labels and title
plt.title("Ratio of Distinct Values in Each Column")
plt.ylabel("Ratio of Distinctiveness")

# Tilt the x-axis labels
plt.xticks(rotation=45, ha='right')
plt.tight_layout()  # Adjust layout to prevent label clipping

# Show the graph
plt.show()

# In[14]:

# Fare Distribution

bins = np.arange(0, 1001, 50)

plt.figure(figsize=(10, 6))
```

```
sns.histplot(pdf['totalFare'], bins=bins)
plt.title('Distribution of Total Fares')
plt.xlabel('Total Fare')
plt.ylabel('Frequency')
plt.show()


# In[15]:


# Travel Duration vs. Total Fare


plt.figure(figsize=(10, 6))
sns.scatterplot(x='travelDurationinMinutes', y='totalFare', data=pdf, hue='isNonStop', alpha=0.4)
plt.title('Travel Duration vs. Total Fare')
plt.xlabel('Travel Duration (Minutes)')
plt.ylabel('Total Fare')
plt.legend(title='Non-Stop Flight', loc='upper left')
plt.show()


# In[16]:


#  Fares by Starting and Destination Airports


fare_matrix = pdf.pivot_table(values='totalFare', index='startingAirport', columns='destinationAirport', aggfunc='mean')
plt.figure(figsize=(12, 8))
sns.heatmap(fare_matrix, annot=True, fmt=".1f", cmap='coolwarm', cbar_kws={'label': 'Average Fare'})
plt.title('Average Fare Heatmap by Starting and Destination Airports')
```

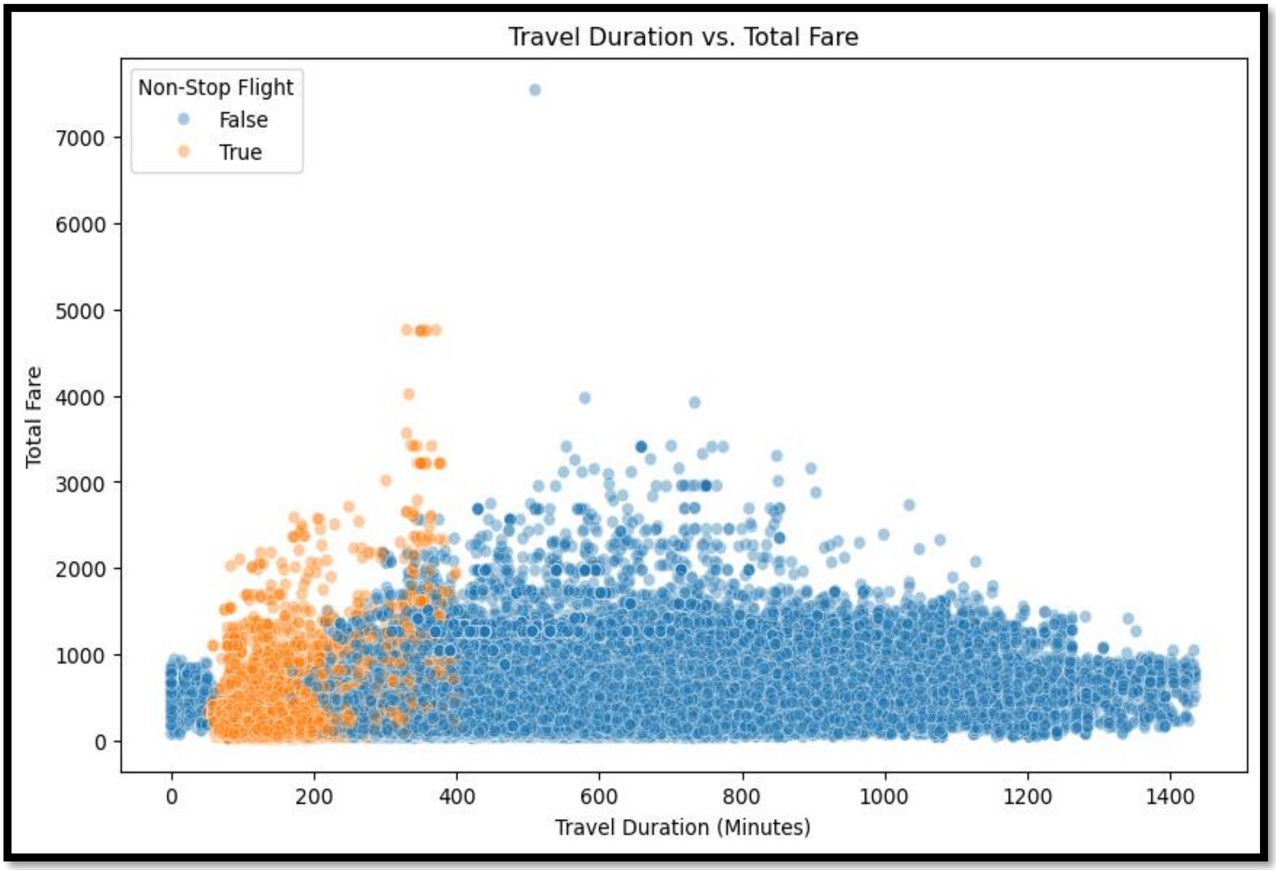
```
plt.xlabel('Destination Airport')
plt.ylabel('Starting Airport')
plt.show()


# In[17]:


# Total Travel Distance vs. Total Fare


plt.figure(figsize=(10, 6))
sns.regplot(x='totalTravelDistance', y='totalFare', data=pdf, scatter_kws={'alpha': 0.2})
plt.title('Total Travel Distance vs. Total Fare')
plt.xlabel('Total Travel Distance (Miles)')
plt.ylabel('Total Fare')
plt.show()


# In[18]:


# Travel Duration Distribution


plt.figure(figsize=(10, 6))
sns.violinplot(x='travelDurationinMinutes', data=pdf)
plt.title('Distribution of Travel Duration')
plt.xlabel('Travel Duration')
plt.show()


# In[19]:
```

```
# Correlation Matrix

plt.figure(figsize=(8, 6))
correlation_matrix = pdf_numeric.corr()
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm', cbar=True)
plt.title('Correlation Matrix')
plt.show()

# # Visualizing Data: Checking for Outliers

# In[20]:

#We will call describe before any changes
pdf_numeric.describe()

# In[21]:

#elapsedDays, baseFare, totalFare, seatsRemaining, totalTravelDistance, 
travelDurationinMinutes

pdf_numeric.boxplot(column=['elapsedDays'])
plt.show()

# In[22]:

pdf_numeric.boxplot(column=['baseFare','totalFare'])
plt.show()
```

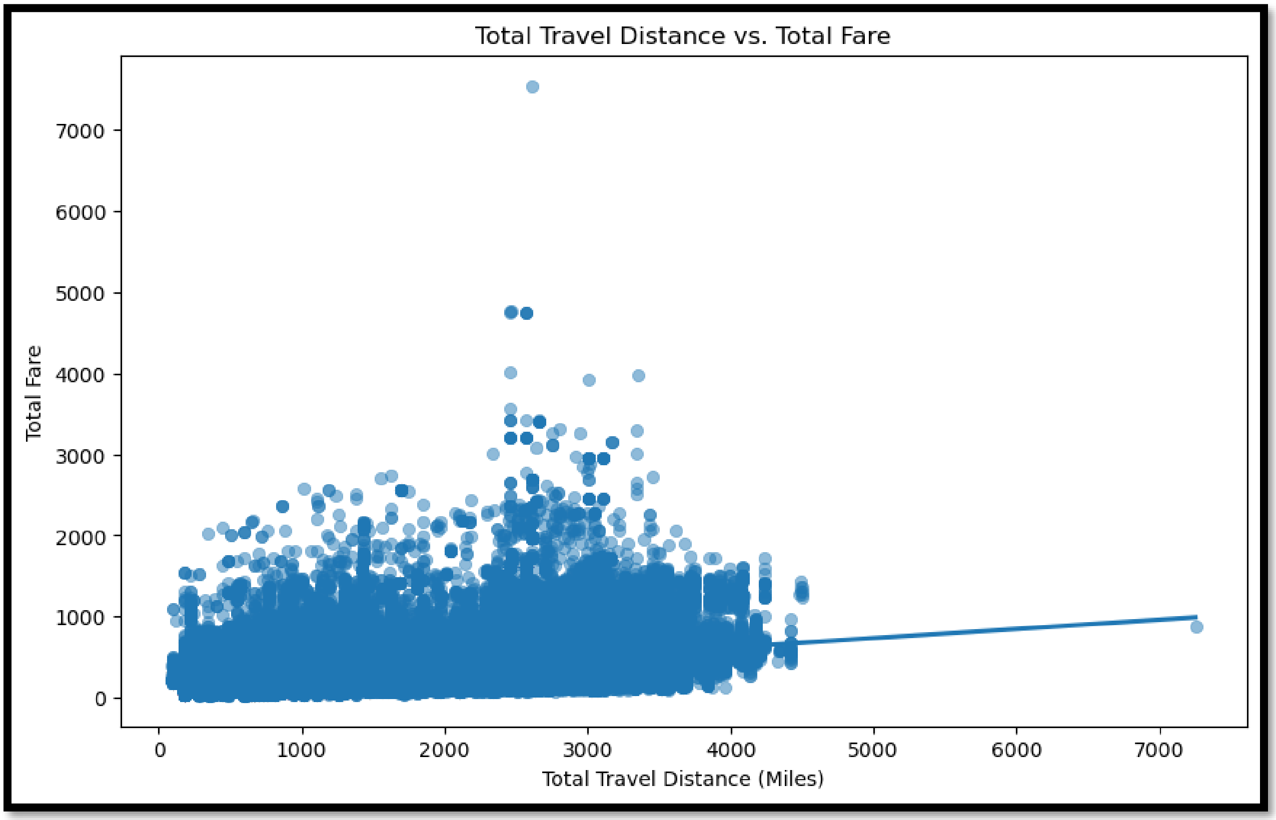
```
# In[23]:


pdf_numeric.boxplot(column=['seatsRemaining'])
plt.show()


# In[24]:


pdf_numeric.boxplot(column=['totalTravelDistance'])
plt.show()


# In[25]:


pdf_numeric.boxplot(column=['travelDurationinMinutes'])
plt.show()


# # Other Descripters


# In[26]:


# Convert both date columns to datetime instead of object so Pandas can find the min and max dates once we describe


pdf['searchDate'] = pd.to_datetime(pdf['searchDate'], format='%Y-%m-%d')
pdf['flightDate'] = pd.to_datetime(pdf['flightDate'], format='%Y-%m-%d')


# In[27]:
```

```
pdf.describe()


# In[28]:


pdf.dtypes


# In[ ]:
```

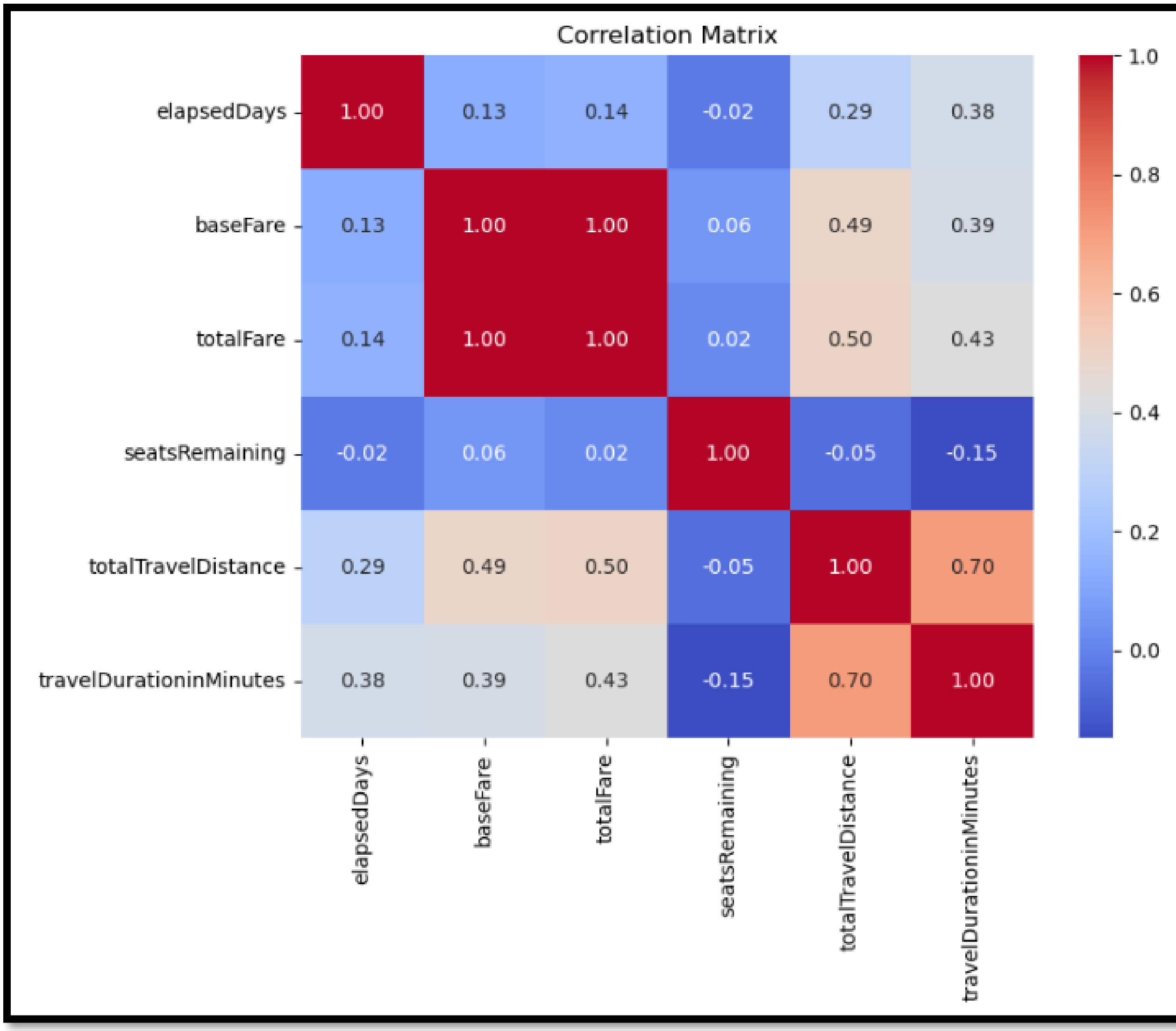
Graph 1: Travel Duration vs Total Fare



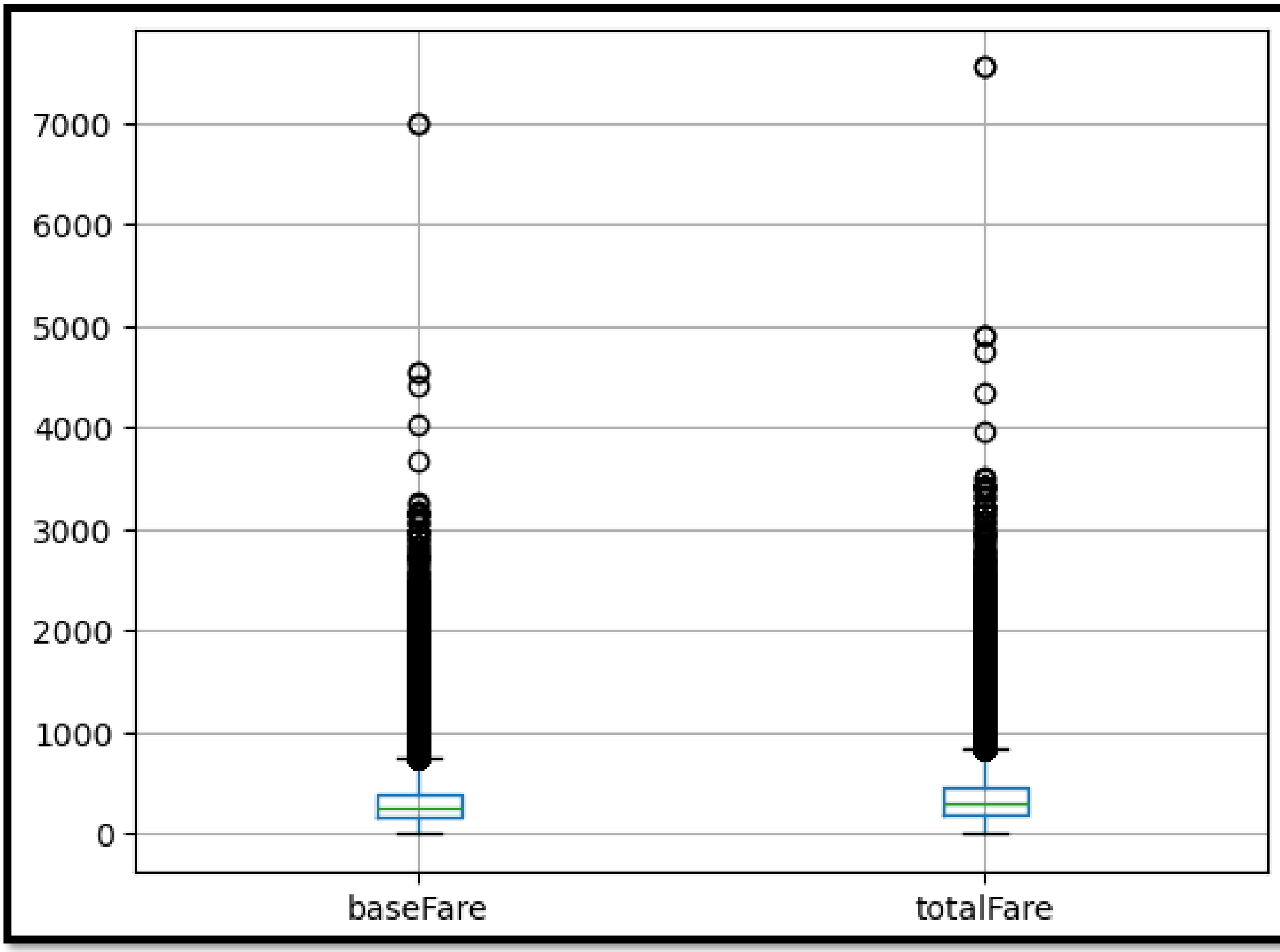
Graph 2: Toal Travel Distance vs. Total Fare



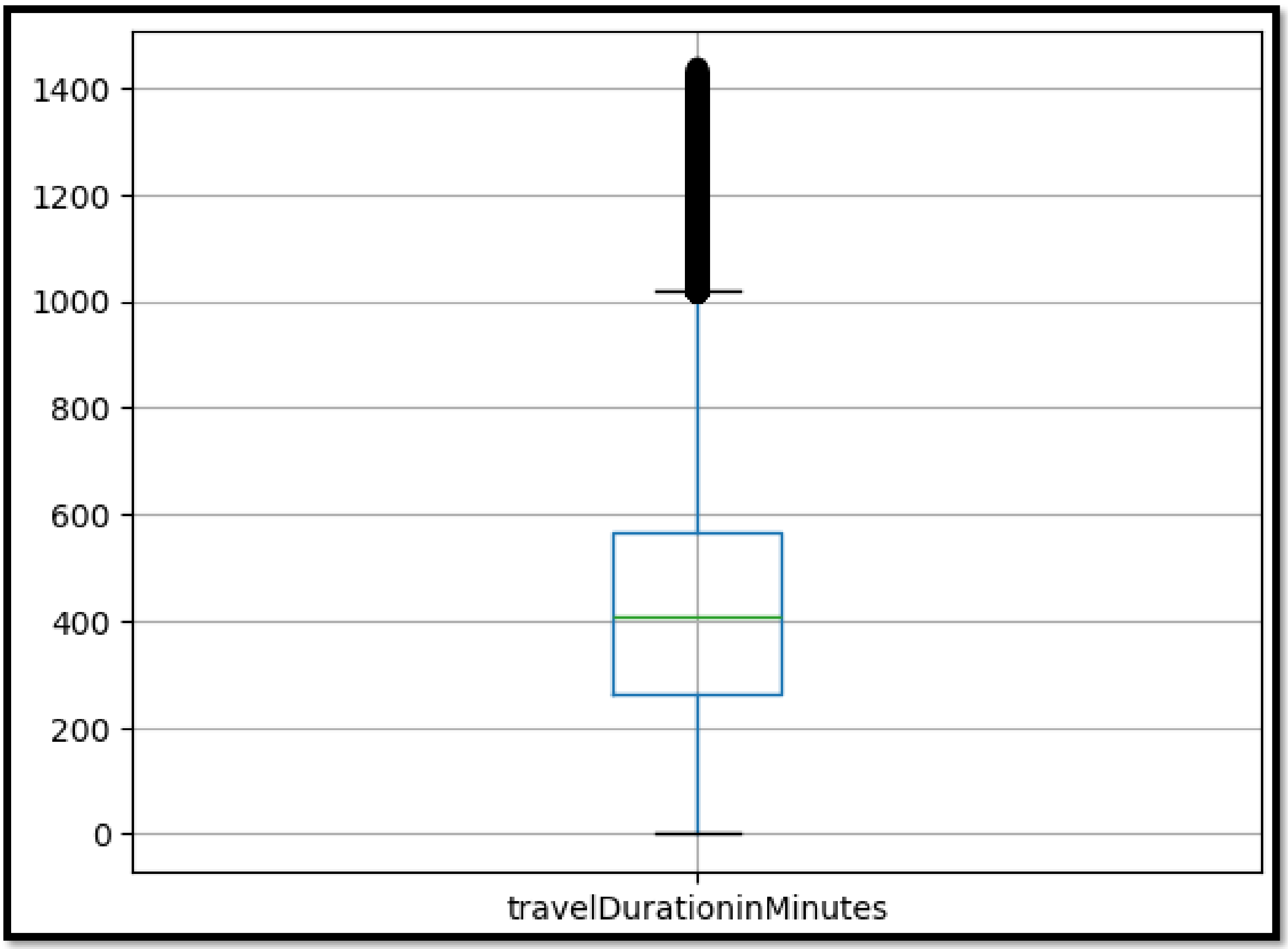
Graph 3: Correlation Matrix



Graph 4: Boxplot of "BaseFare" and "TotalFare"



Graph 5: Boxplot of "travelDurationMinutes"



Appendix C

Source code for Data Cleaning file.

```
#!/usr/bin/env python

# coding: utf-8


# # Get data into a PySpark DataFrame


# In[1]:


# Url for flight data

url = "gs://my-bigdata-project-mp/landing/itineraries.csv"


# Load data into a PySpark DataFrame

clean_df = spark.read.csv(url, header=True, inferSchema=True)


# In[2]:


# Display our inital Schema before all changes.

clean_df.printSchema()


# # Remove NULLS

#

# #### From the Exploratory Data Analysis, we know that the only columns with NULLS
are...

# - totalTravelDistance

# - segmentsEquipmentDescription
```

```
# In[4]:


# Count records BEFORE dropping nulls
clean_df.count()


# In[5]:


# Count records AFTER dropping nulls in totalTravelDistance
clean_df = clean_df.filter("totalTravelDistance is not NULL")
clean_df.count()


# In[6]:


# Count records AFTER dropping nulls in segmentsEquipmentDescription
clean_df = clean_df.filter("segmentsEquipmentDescription is not NULL")
clean_df.count()


# # Remove Outliers


# In[7]:


#Although this function could be part of the feature engineering process,
# we do it here to remove outliers before we start feature engineering.
```

```
# We will extract the hour and minutes from travelDuration and combine them into a new column called travelDurationMinutes


#We will use these functions to extract the numbers from the strings in travelDuration

from pyspark.sql.functions import regexp_extract, col, when, expr


def transform_travel_duration(spark_df):


    # Define regex patterns to capture hours and minutes

    hours_pattern = "PT(\\d+)H"       # Captures the digits before 'H' in the "PT#H" format

    minutes_pattern = "H(\\d+)M"      # Captures the digits before 'M' in the "#M" format after 'H'

    only_minutes_pattern = "PT(\\d+)M" # For cases with only minutes (e.g., "PT20M")


    # Extract hours and minutes, converting to integers

    df_extracted = spark_df \

        .withColumn("hours", regexp_extract(col("travelDuration"), hours_pattern, 1).cast("int")) \

        .withColumn("minutes", when(col("travelDuration").rlike(only_minutes_pattern),

                            regexp_extract(col("travelDuration"), only_minutes_pattern, 1))

                .otherwise(regexp_extract(col("travelDuration"), minutes_pattern, 1)).cast("int"))


    # Calculate total minutes

    df_with_total_minutes = df_extracted.withColumn(

        "travelDurationMinutes",

        expr("coalesce(hours, 0) * 60 + coalesce(minutes, 0)")

    )
```

```
    # Get these new columns into our df, then drop the two unnecessary columns
    spark_df = df_with_total_minutes

    # Compare travelDuration and travelDurationMinutes to make sure the values are correct
    spark_df = spark_df.drop("hours","minutes")

    # Finally, drop travelDuration as it is no longer useful
    spark_df = spark_df.drop("travelDuration")

    return spark_df

# In[8]:

#FUNCTION 1
# we will make a function that takes a pyspark df, column name, min, max, as arguments
#
# it modifies the pyspark dataframe to enforce the min and max values in the given column.
#   - specifically, it will then remove any value equal to or above max, and any value equal to or below min

def set_min_max_col(spark_df, col_name: str, min: float, max: float):
    new_df = spark_df.where((col(col_name) <= max) & (col(col_name) >= min))
    return new_df

#FUNCTION 2
```

Uses the previous function multiple times

We come up with min max manually with new found knowledge from the Exploratory Data Analysis script.

```
def trim_outliers(spark_df):
    spark_df = set_min_max_col(spark_df, 'elapsedDays', 0, 1.2)
    spark_df = set_min_max_col(spark_df, "baseFare", 0, 740)
    spark_df = set_min_max_col(spark_df, "totalFare", 0, 825)
    spark_df = set_min_max_col(spark_df, "travelDurationMinutes", 0, 1000) # remember to call transform_travel_duration() first.
    spark_df = set_min_max_col(spark_df, "totalTravelDistance",0,4700)
    spark_df = set_min_max_col(spark_df, "seatsRemaining",0,20)
    return spark_df


# In[9]:


#Turn travel duration into a column we can measure. Also allows us to remove its outliers.
clean_df = transform_travel_duration(clean_df)


# In[10]:


#Transform all columns with outliers
clean_df = trim_outliers(clean_df)


# In[11]:


#Count records after forcing a min-max on all numeric columns
```

```
clean_df.count()

# # Display our final schema and write to /cleaned in our google bucket

# In[12]:

# Here is the final schema for the clean DataFrame
clean_df.printSchema()

# In[13]:

# We will now write this back to /cleaned

url = "gs://my-bigdata-project-mp/cleaned"

clean_df.write.parquet(path=url, mode="overwrite")
```

Appendix D

Source code for Feature Engineering and Data Modeling file.

```
#!/usr/bin/env python

# coding: utf-8


# # Model Creation Script


# In[1]:


spark


# # Get clean data


# In[2]:


# Url for flight data

url = "gs://my-bigdata-project-mp/cleaned"


# Load data into a PySpark DataFrame

df = spark.read.parquet(url)


# In[3]:


# Display our inital Schema before all changes.

df.printSchema()
```

In[4]:

```
# Drop legId since it will not be useful for ML as it is an identifer.
df.drop("legId")

# Drop baseFare because of its high correlation to totalFare.
df.drop("baseFare")
```

Import libraries for ML

In[5]:

```
from pyspark.sql.functions import *
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler, StandardScaler, Binarizer
from pyspark.ml import Pipeline
# Import the logistic regression model
from pyspark.ml.regression import LinearRegression, GeneralizedLinearRegression
# Import the evaluation module
from pyspark.ml.evaluation import *
# Import the model tuning module
from pyspark.ml.tuning import *
# We will use these functions to extract the numbers from the strings in travelDuration
from pyspark.sql.functions import regexp_extract, col, when, expr, split, rand
```

In[6]:

```
# Sample a portion of the dataset. Using all roughly 76 million records, with all features, will take too long.

SEED = 645

sample_percentage = 0.05 # 1% (or 0.1) of the dataset represents around 760,000 rows.

df = df.sample(withReplacement=False, fraction=sample_percentage, seed=SEED)


# # Create transform functions

#    - Function to seperate segments

#    - Function to extract features from date columns

#    - Function to binarize certain features


# In[8]:


# Function to seperate a segment column into new columns.

# Three new columns will be created, and split values will enter them.

# ARG1: Pyspark dataframe you want to transform

# ARG2: the name of the column you want to seperate

# ARG3: How many columns you want to create

#   EX- If you expect 3 values in a segment colum, set split to 3 so all values can be captured

#       If set too high, there will be columns with all 0's

#       If set too low, then some data will not be captured.


def seperate_segment(df, column_name, splits=2):


    # Will keep track of how many segments are made then return them as a list
```

```
    name_list_of_split_segments = []

    # Split the column using '||' and extract elements
    split_col = split(col(column_name), r'\|\|')  # Split on '||'

    # Add three new columns for the first three parts of the split, replacing nulls with 0 and
casting result as integers
    for i in range(splits):
        df = df.withColumn(
            f"{column_name}[{i}]",
            when(split_col.getItem(i).isNull(), lit(0))
            .otherwise(split_col.getItem(i))
            .cast("int"))  # Cast to integer

        # Add name of new split to list
        name_list_of_split_segments.append(f"{column_name}[{i}]")

    return df, name_list_of_split_segments


# In[9]:


# Function to extract features from a date column (Month,Day,DayOfWeek,isWeekend)
# ARG1: Pyspark dataframe you want to transform
# ARG2: Name of date column you want to extract features from

def date_feature_extraction(df, col_name: str):
```

```
    df = df.withColumn(col_name+"Month", month(col(col_name)))

    df = df.withColumn(col_name+"Day", day(col(col_name)))

    df = df.withColumn(col_name+"DayOfWeek", dayofweek(col(col_name)))

    df = df.withColumn(col_name+"isWeekend", when(df[col_name+"DayOfWeek"] == 1,
1.0).\

                    when(df[col_name+"DayOfWeek"] == 7, 1.0).otherwise(0))

    return df


# In[10]:


# Function that turns a boolean into either a 0 (False) or 1 (True)

# ARG1: Pyspark dataframe you want to transform

# ARG2: Name of boolean column you want to transform

def boolean_binarizer(df, col_name: str):

    df = df.withColumn(col_name+"Binarized", when(df[col_name] == True, 1.0).\

                    otherwise(0))

    return df


# # Transform the dataframe with newly created functions


# In[11]:


# Seperate certain columns which could allow for better model performance when using
seperated columns to train.

# COLUMNS TO POSSIBLY SEPERATE

# - segmentsArrivalTimeRaw

# - segmentsArrivalTimeEpochSeconds
```

- segmentsDepartureTimeRaw

- segmentsDepartureTimeEpochSeconds

- segmentsDurationInSeconds

#

The only column may be worth seperating is "segmentsDurationInSeconds", which technically could be correlated

to both arrival and departure time. This cant be proven visually since those columns are all different in format,

but it may be possible to derive arrival and departure time if we have duration. To truly prove this may require

reformatting the data quite a bit, but for now, we will only seperate one column.

df, new_split_segments = seperate_segment(df, "segmentsDurationInSeconds", splits=2)

split_segments = split_segments + new_split_segments

Visualize if columns are separated correctly

#

#df.select("segmentsDurationInSeconds","segmentsDurationInSeconds[0]",

"segmentsDurationInSeconds[1]","segmentsDurationInSeconds[2]").show(10)

In[ ]:

Checking outliers for the split segmentsDuration column. These outliers have yet to be dropped for the ML model.

Function from Clean Data script that will be used to remove outliers on newly split columns

```
def set_min_max_col(spark_df, col_name: str, min: float, max: float):
   new_df = spark_df.where((col(col_name) <= max) & (col(col_name) >= min))
   return new_df

# Function to view outliers on newly split columns.
def view_seg_splits(sdf, col_name, splits=2):
   for i in range(splits):
      col_to_select = col_name+"["+str(i)+"]"
      df = sdf.select(col_to_select).sample(False, 0.08, seed=SEED).toPandas()
      df.boxplot(column=[col_to_select])
      plt.show()

# Count before and after setting min-max on columns.
count = df.count()
print("BEFORE-------"+"[ OLD Count: "+str(count)+" ]--------------------")
view_seg_splits(df, "segmentsDurationInSeconds", splits=2)

df = set_min_max_col(df,"segmentsDurationInSeconds[0]", 0, 20000)
df = set_min_max_col(df,"segmentsDurationInSeconds[1]", 0, 19000)
#sdf_adj = set_min_max_col(sdf_adj,"segmentsDurationInSeconds[2]", 0, 4000)

count = df.count()
print("AFTER-------"+"[ NEW Count: "+str(count)+" ]--------------------")
view_seg_splits(df, "segmentsDurationInSeconds", splits=2)

# In[12]:
```

```
# Extract features from date columns
df = date_feature_extraction(df, "searchDate")
df = date_feature_extraction(df, "flightDate")

# Visualize old cols vs new cols
#
#df.select("searchDate","searchDateMonth","searchDateDay","searchDateisWeekend","searchDateDayOfWeek").filter(col("searchDateisWeekend") == 1).show(10)
#df.select("flightDate","flightDateMonth","flightDateDay","flightDateisWeekend","flightDateDayOfWeek").show(10)

# In[13]:

# Columns to binarize: isBasicEconomy, isRefundable, isNonStop
df = boolean_binarizer(df, "isBasicEconomy")
df = boolean_binarizer(df, "isRefundable")
df = boolean_binarizer(df, "isNonStop")

# Visualize old cols vs new cols
#
#df.select("isBasicEconomy","isBasicEconomyBinarized").show()
#df.select("isRefundable","isRefundableBinarized").show()
#df.select("isNonStop","isNonStopBinarized").show()

# In[14]:
```

```
# Check schema to make all features are correct datatypes
df.printSchema()


# # Create pipeline


# In[15]:


# Define the string columns to be indexed and encoded
string_columns = [
    "startingAirport", "destinationAirport", "fareBasisCode",
    "segmentsArrivalAirportCode", "segmentsDepartureAirportCode",
    "segmentsAirlineName", "segmentsAirlineCode",
    "segmentsEquipmentDescription", "segmentsDistance", "segmentsCabinCode"
]

# Dynamically generate input and output column names for indexer and encoder
indexer_output_columns = [f"{col}Index" for col in string_columns]
encoder_output_columns = [f"{col}Vector" for col in string_columns]

# Create the StringIndexer
indexer = StringIndexer(
    inputCols=string_columns,
    outputCols=indexer_output_columns,
    handleInvalid="keep"
)
```

```
# Create OneHotEncoder
encoder = OneHotEncoder(
   inputCols=indexer_output_columns,
   outputCols=encoder_output_columns,
   dropLast=True,
   handleInvalid="keep"
)

# Define all numerical columns that will be put through their own assembler, then scaled.
num_columns = [ "searchDateMonth", "searchDateDay", "flightDateMonth",
"flightDateDay",
         "elapsedDays", "seatsRemaining", "totalTravelDistance",
         "travelDurationMinutes", "searchDateDayOfWeek", "flightDateDayOfWeek",
          ] + split_segments

numerical_assembler =
VectorAssembler(inputCols=num_columns,outputCol="numVector")
scaler = StandardScaler(inputCol="numVector",outputCol="numScaled")

# Create an assembler
assembler = VectorAssembler(inputCols=["startingAirportVector",
"destinationAirportVector", "fareBasisCodeVector",
                         "segmentsArrivalAirportCodeVector","segmentsDepartureAirportCodeVect
or",
                         "segmentsAirlineNameVector","segmentsAirlineCodeVector","segmentsEq
uipmentDescriptionVector",
                         "segmentsDistanceVector","segmentsCabinCodeVector","numScaled","sea
rchDateisWeekend","flightDateisWeekend",
```

```
                              "isBasicEconomyBinarized","isRefundableBinarized","isNonStopBinarize
d"],
                      outputCol="features")


# Create a Linear Regression Estimator
linear_reg = LinearRegression(labelCol="totalFare")


# In[16]:


# Create the pipeline
flights_pipe = Pipeline(stages=[indexer, encoder,
                         numerical_assembler,
                         scaler, assembler,
                         linear_reg])


# Call .fit to transform the data
transformed_df = flights_pipe.fit(df).transform(df)


# In[17]:


# Review features
transformed_df.select("features").show(10, truncate=False)


# # Train Models


# In[18]:
```

```
# Split the data into training and test sets
trainingData, testData = df.randomSplit([0.70, 0.3], seed=SEED)


# In[19]:


# Create a regression evaluator (to get RMSE, R2, RME, etc.)
evaluator = RegressionEvaluator(labelCol="totalFare")


# In[20]:


# Create a grid to hold hyperparameters
grid = ParamGridBuilder()


# In[21]:


# Build the parameter grid
grid = grid.build()


# In[22]:


# Create the CrossValidator using the hyperparameter grid
cv = CrossValidator(estimator=flights_pipe,
          estimatorParamMaps=grid,
          evaluator=evaluator,
          numFolds=3)
```

```
# In[23]:


# Train the models

all_models  = cv.fit(trainingData)


# # Metrics


# In[24]:


# Show the average performance over the six folds

print(f"Average metric {all_models.avgMetrics}")


# # Get best model


# In[25]:


# Get the best model from all of the models trained

bestModel = all_models.bestModel


# Use the model 'bestModel' to predict the test set

test_results = bestModel.transform(testData)


# Show the predicted totalFare

test_results.select('totalFare',
'prediction').orderBy(rand(seed=SEED)).limit(20).show(truncate=False)
```

```
# Calculate RMSE and R2

rmse = evaluator.evaluate(test_results, {evaluator.metricName:'rmse'})

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

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


# # Save model and data


# In[26]:


# Save best model

url_model = "gs://my-bigdata-project-mp/models/flight_prices_linear_regression_model" # save model here

bestModel.write().overwrite().save(url_model)


# Save transformed data

url_trusted = "gs://my-bigdata-project-mp/trusted" # save data with features here

transformed_df.write.parquet(path=url_trusted, mode="overwrite")
```

Appendix E

Source code for Model Evaluation and Data Visualization file.

```
#!/usr/bin/env python

# coding: utf-8


# In[1]:


spark


# In[2]:


import io

import numpy as np

from matplotlib import pyplot as plt

import pandas as pd

import seaborn as sns

from google.cloud import storage

from pyspark.ml import PipelineModel


SEED = 645

bucket_name = "my-bigdata-project-mp"


# # Get data with features vector and model


# In[3]:
```

```
# Path for data used for model
data_path = "gs://my-bigdata-project-mp/trusted"

# Load data into a PySpark DataFrame
sdf = spark.read.parquet(data_path)

# In[4]:

# Checking schema
sdf.printSchema()

# In[5]:

sdf.count()

# In[6]:

# Path for the linear regression model
model_path = "gs://my-bigdata-project-mp/models/flight_prices_linear_regression_model"

# Load PipelineModel into a variable
pipeline = PipelineModel.load(model_path)

# Extract the model
lr_model = pipeline.stages[-1]
```

```
# # Function to save figure

# In[7]:

# FUNCTION
# ARG1 - matplot variable you used for your plot
# ARG1 - Name you want to give the image.
# ARG2 - The type you want the image to be. This function assumes we want a PNG.

def save_fig(plt, img_name, img_type="png"):
    print("Saving figure...")
    # Create a memory buffer to hold the figure
    img_data = io.BytesIO()
    # Write the figure to the buffer
    plt.savefig(img_data, format=img_type, bbox_inches='tight')
    # Rewind the pointer to the start of the data
    img_data.seek(0)
    # Connect to Google Cloud Storage
    storage_client = storage.Client()
    # Point to the bucket
    bucket = storage_client.get_bucket(bucket_name)
    # Create a blob to hold the data. Give it a file name
    blob = bucket.blob(img_name+"."+img_type)
    # Upload the img_data contents to the blob
    blob.upload_from_file(img_data)
    print("Picture successfully uploaded!")
```

```
# # Predicted vs Actual

# - Scatter plot of predicted vs actual

# - Shows how accurate the model is (closer to the line means better prediction)


# In[8]:


# Scatter plot of predicted vs. actual


# Define what name the image file for this picture will have and the type of image it will be saved as

img_name = "actual_vs_predicted"

img_type = "png"


df = sdf.select("prediction","totalFare").sample(False, 0.01, seed=SEED).toPandas()


plt.figure(figsize=(8, 8))

sns.scatterplot(x=df['totalFare'], y=df['prediction'], alpha=0.2)

plt.plot([df['totalFare'].min(), df['totalFare'].max()],

    [df['totalFare'].min(), df['totalFare'].max()],

    color='red', linestyle='--', label='Ideal Fit')  # Add a reference line for ideal fit

plt.title('Predicted vs Actual')

plt.xlabel('Actual Values')

plt.ylabel('Predicted Values')

plt.xticks([0,100,200,300,400,500,600,700,800])  # Tick marks from 0 to 1000 with step of 100

plt.yticks([0,100,200,300,400,500,600,700,800,900,1000])  # Same for y-axis
```

```
plt.legend()

plt.grid()


save_fig(plt,img_name,img_type)

plt.show()


# ## Histogram of Residuals

# - Normality: If the residuals are normally distributed (bell-shaped curve), this supports the normality assumption of linear regression. If the residuals are skewed or have outliers, this suggests violations of the normality assumption.


# In[9]:


# Define what name the image file for this picture will have and the type of image it will be saved as

img_name = "histogram_of_residuals"

img_type = "png"


# Extract actual values and predicted values

result_df = sdf.select("prediction","totalFare").sample(False, 0.01, seed=SEED).toPandas()


# Compute residuals (difference between actual and predicted)

result_df['residual'] = result_df['totalFare'] - result_df['prediction']


plt.figure(figsize=(10, 6))

sns.histplot(result_df['residual'], kde=True, bins=30, color='blue')
```

```
plt.title('Residuals Histogram')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.grid(True)

save_fig(plt,img_name,img_type)
plt.show()

del result_df

# # Correlation Matrix
# - With many new features created and most numerical, we can check correlations for more columns

# In[10]:

# Correlation Matrix

# 1st, get all numerical columns to do the correlation matrix

# Step 1: Grab one row from the PySpark DataFrame and convert it to Pandas
row_df = sdf.limit(1).toPandas()
# Step 2: Extract numerical columns from the Pandas DataFrame
numeric_columns = row_df.select_dtypes(include=['number']).columns.tolist()
# Step 3: Re-select the numerical columns from the original PySpark DataFrame
sdf_numeric = sdf.select(*numeric_columns).drop("isRefundableBinarized","baseFare")
```

```
# Step 4: Convert the selected PySpark DataFrame to a Pandas DataFrame
df = sdf_numeric.sample(False, 0.01, seed=SEED).toPandas()


# In[11]:


# 2nd, Compute correlation matrix


# Define what name the image file for this picture will have and the type of image it will be saved as
img_name = "correlation_matrix_post_pipeline"
img_type = "png"


# Compute correlation matrix
corr_matrix = df.corr()


# Create a mask to remove the upper triangle of the correlation matrix
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))


# Apply the mask to the correlation matrix (set upper triangle to NaN or zero)
masked_corr_matrix = corr_matrix.mask(mask)


# Plot heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(masked_corr_matrix, annot=False, cmap='coolwarm', fmt=".2f", linewidths=1)
plt.title('Correlation Matrix')
```

```
save_fig(plt,img_name,img_type)

plt.show()


# # Feature Coefficients

# - Plots the coefficients of each feature

# - Note: If changing anything related to the ordering of features in model creation,

#   then the features here must also be changed to reflect the same order.

#   Otherwise, the coefficients will not actually correlate to the features.


# In[12]:


# Get the coefficients and intercept

coefficients = lr_model.coefficients


# Get the feature names (order must match the feature vector in assembler)

feature_columns = [

   "startingAirportVector", "destinationAirportVector", "fareBasisCodeVector",

   "segmentsArrivalAirportCodeVector", "segmentsDepartureAirportCodeVector",

   "segmentsAirlineNameVector", "segmentsAirlineCodeVector",
"segmentsEquipmentDescriptionVector",

   "segmentsDistanceVector", "segmentsCabinCodeVector", "numScaled",

   "searchDateisWeekend", "flightDateisWeekend",

   "isBasicEconomyBinarized", "isRefundableBinarized", "isNonStopBinarized"

]


# Combine coefficients with feature names
```

feature_coefficients = list(zip(feature_columns, coefficients))

Optionally, use pandas to display the coefficients for easier interpretation

feature_coefficients_df = pd.DataFrame(feature_coefficients, columns=["Feature", "Coefficient"])

Display the DataFrame

print(feature_coefficients_df)

In[13]:

Define what name the image file for this picture will have and the type of image it will be saved as

img_name = "feature_coefficients"

img_type = "png"

Plot the coefficients

plt.figure(figsize=(10, 6))

sns.barplot(x='Coefficient', y='Feature', data=feature_coefficients_df)

plt.title('Feature Coefficients from Linear Regression')

plt.xlabel('Coefficient Value')

plt.ylabel('Feature')

save_fig(plt,img_name,img_type)

plt.show()

In[14]:

```
# Print hyperparameters from Linear Regression Model
print("Best Model Parameters:")
for param, value in lr_model.extractParamMap().items():
    print(f"{param.name}: {value}")
```