CIS9760 Big Data Technologies

Semester Data Engineering Project
Spring 2025

Yashasvi Bhati | 24559155
yashasvi.bhati@baruchmail.cuny.edu

May 14th, 2025

Milestone 1 : The Proposal

Predicting Subway Ridership Based on Weather Conditions and Holidays

DESCRIPTION

For this project, I will be using two key datasets: the MTA Subway Hourly Ridership dataset and the Open-Meteo weather dataset. The MTA dataset provides detailed ridership data across different subway stations in New York City, while the weather dataset offers historical and real-time meteorological data, including temperature, precipitation, and other relevant factors. By integrating these datasets, I aim to analyze how external conditions influence subway usage and identify patterns that could help predict ridership trends.

DATASET LINKS

- MTA Subway Ridership Dataset: https://data.ny.gov/Transportation/MTA-Subway-Hourly-Ridership-2020-2024/wujg-7c2s/about_data
- Weather Dataset: https://open-meteo.com/

ATTRIBUTES OF THE DATASETS

MTA Subway Ridership Data

`transit_timestamp, transit_mode, station_complex_id, station_complex, borough, payment_method, fare_class_category, ridership, transfers, latitude, longitude, georeference`

Weather API Data

`temperature, precipitation, wind_speed, weather_description`

Additionally, I will create a `holiday` feature by importing data on holiday dates.

PREDICTION GOAL

The goal of this project is to predict whether subway ridership will be higher or lower than average based on weather conditions and holidays. The **target variable** will be **ridership**, categorized as either above or below the average. To make these predictions, I will use **logistic regression** since the outcome is binary (high or low ridership). Additionally, I may explore other machine learning models to see if they improve accuracy.

The goal is to develop a predictive model to help transit authorities optimize operations, staffing, and scheduling based on expected ridership.

Milestone 2 : Data Acquisition

SUMMARY OF APPROACH

For Milestone 2, the goal was to acquire MTA ridership data from the NYC Open Data API and historical weather data. I initially used a script to bypass the API limits for the MTA data, then automated the download process to accumulate at least 10GB of data. For the weather data, I leveraged a user-friendly website interface to filter the relevant 2020-2024 data, then used a Python script to automate the extraction and storage in CSV format. Both datasets were ultimately stored in my VM and transferred to cloud storage for easier access.

CHALLENGES

API Limitations

The biggest challenge came from the 1000 rows per request API limit from the NYC Open Data API. Despite running and debugging several codes, the issue persisted. Eventually, I found an online solution that helped download 48 MB of data, but I needed a method to continuously collect data until it reached at least 10GB.

Disk Space

Downloading the data in JSON format without compression quickly consumed disk space. Although I still had sufficient space, the process took a long time due to the file's size and format.

Accessing Data

I managed to download only 5GB of MTA data before encountering issues with storage. With help from my professor, I accessed the remaining data via his server and bucket.

File Upload Issues

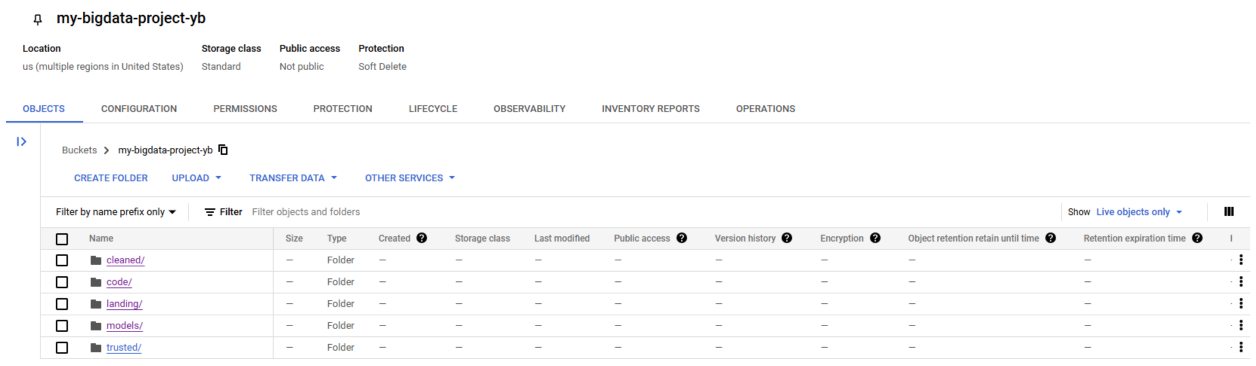
After transferring the data to my system, I encountered a permission access issue during the upload process. After troubleshooting, I realized the file was in a ZIP format on the server, and once I unzipped it, the upload worked.

ACKNOWLEDGEMENT

A big thank you to **Professor Richard Holowczak** for all the help with the technical challenges, especially when it came to accessing the MTA ridership data. Your support and guidance really made a huge difference, and I’m incredibly grateful for it!

Attached below are the console snippets confirming that the requirements for Milestone 2 have been met:

1. Created five folders in the project bucket, **`my-bigdata-project-yb`**, and copied the data to the **`landing`** folder

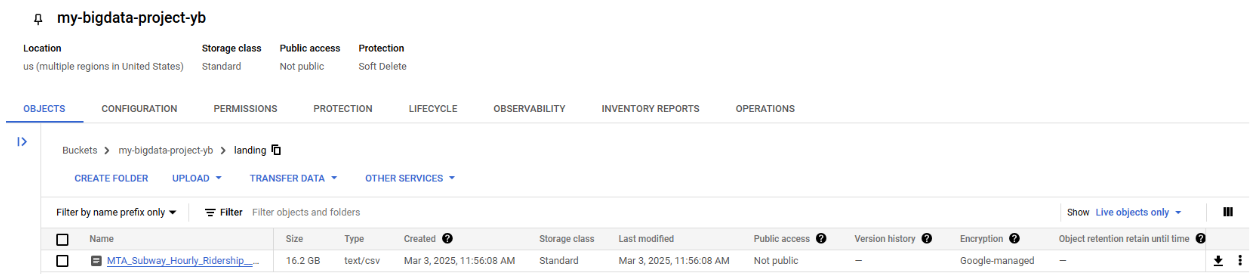


my-bigdata-project-yb

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

Buckets > my-bigdata-project-yb

Name	Size	Type	Created	Storage class	Last modified	Public access	Version history	Encryption	Object retention retain until time	Retention expiration time
cleaned/	—	Folder	—	—	—	—	—	—	—	—
code/	—	Folder	—	—	—	—	—	—	—	—
landing/	—	Folder	—	—	—	—	—	—	—	—
models/	—	Folder	—	—	—	—	—	—	—	—
trusted/	—	Folder	—	—	—	—	—	—	—	—



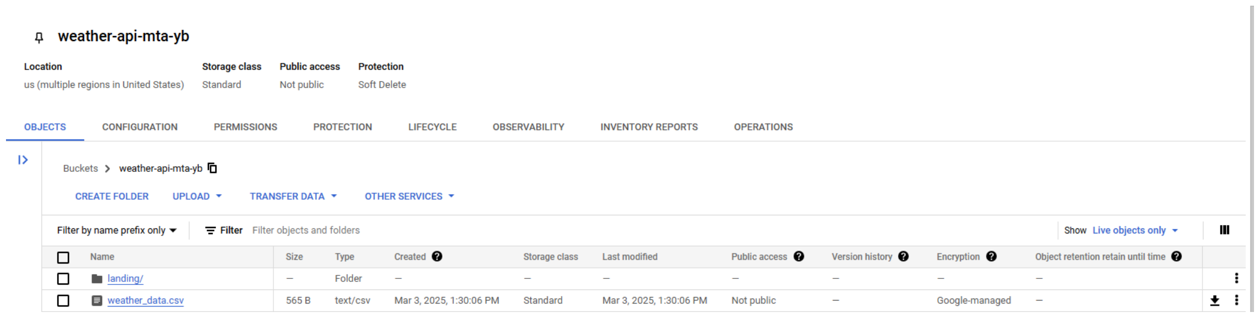
my-bigdata-project-yb

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

Buckets > my-bigdata-project-yb > landing

Name	Size	Type	Created	Storage class	Last modified	Public access	Version history	Encryption	Object retention retain until time
MTA_Subway_Hourly_Ridership_...	16.2 GB	text/csv	Mar 3, 2025, 11:56:08 AM	Standard	Mar 3, 2025, 11:56:08 AM	Not public	—	Google-managed	—

2. Transferred the weather data to a separate bucket, **`weather-api-mta-yb`**.



weather-api-mta-yb

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

Buckets > weather-api-mta-yb

Name	Size	Type	Created	Storage class	Last modified	Public access	Version history	Encryption	Object retention retain until time
landing/	—	Folder	—	—	—	—	—	—	—
weather_data.csv	565 B	text/csv	Mar 3, 2025, 1:30:06 PM	Standard	Mar 3, 2025, 1:30:06 PM	Not public	—	Google-managed	—

Milestone 3 : Exploratory Data Analysis

Primary Dataset: MTA Subway Ridership

PREREQUISITE STEP

To proceed with the data analysis, the quota size had to be increased from 500 GB to 600 GB. This step was essential due to the large size of the dataset. Setting up a multi-node cluster took a while as part of this preparation.

APPROACH

1. Using the professor's code to define functions for reading CSV files, performing exploratory data analysis (EDA), numerical EDA, and categorical EDA.
2. The initial attempt to call `main_subway()` successfully outputted EDA results, but visualization attempts kept crashing the kernel.
3. To solve this, a new function named `small_chunk_df` was defined for visualizations, processing data in chunks of 10,000,000 records each to avoid memory overload.

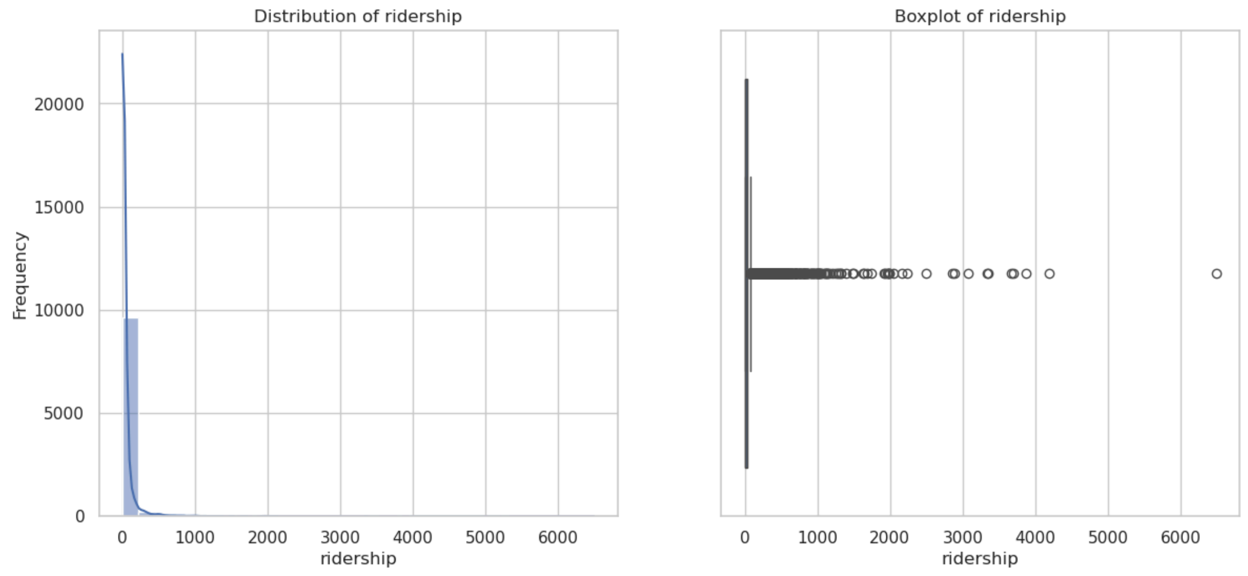
SUMMARY STATISTICS

- Number of Rows with Null Values: 0
- Integer Data Type Columns: `ridership, transfers`
- Float Data Type Columns: `latitude, longitude`

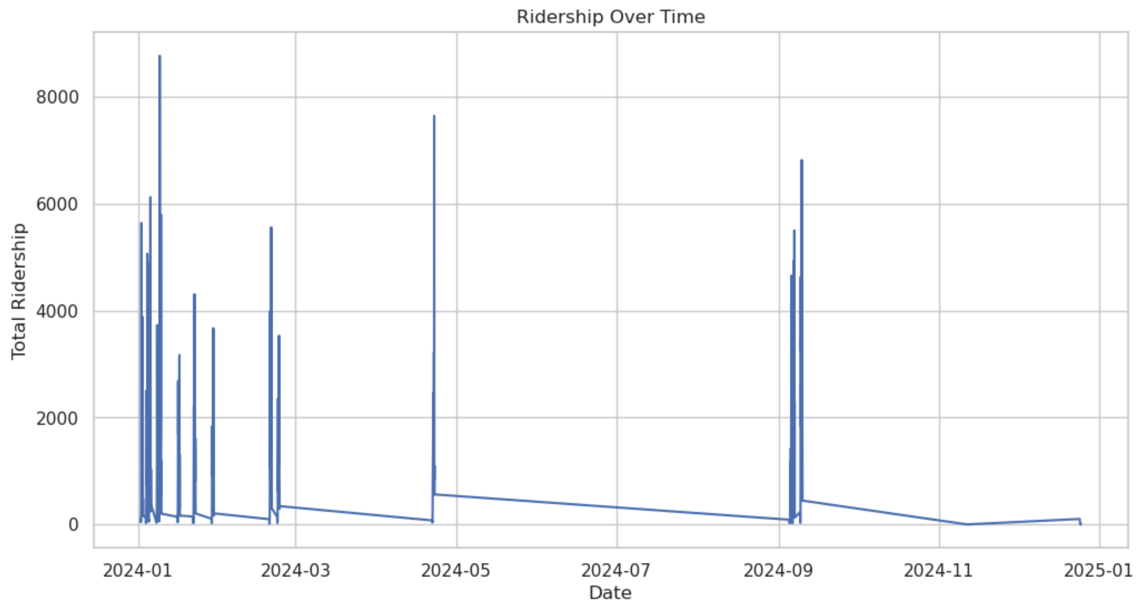
```
    ridership  transfers  latitude  longitude
count  696365.00  696365.00 696365.00  696365.00
mean       48.95       2.15     40.73     -73.94
std       182.41      13.36      0.08       0.06
min         1.00       0.00     40.58     -74.07
25%         4.00       0.00     40.68     -73.98
50%        12.00       0.00     40.72     -73.95
75%        36.00       1.00     40.79     -73.90
max     13666.00    1385.00     40.90     -73.76
```

VISUAL ANALYSIS

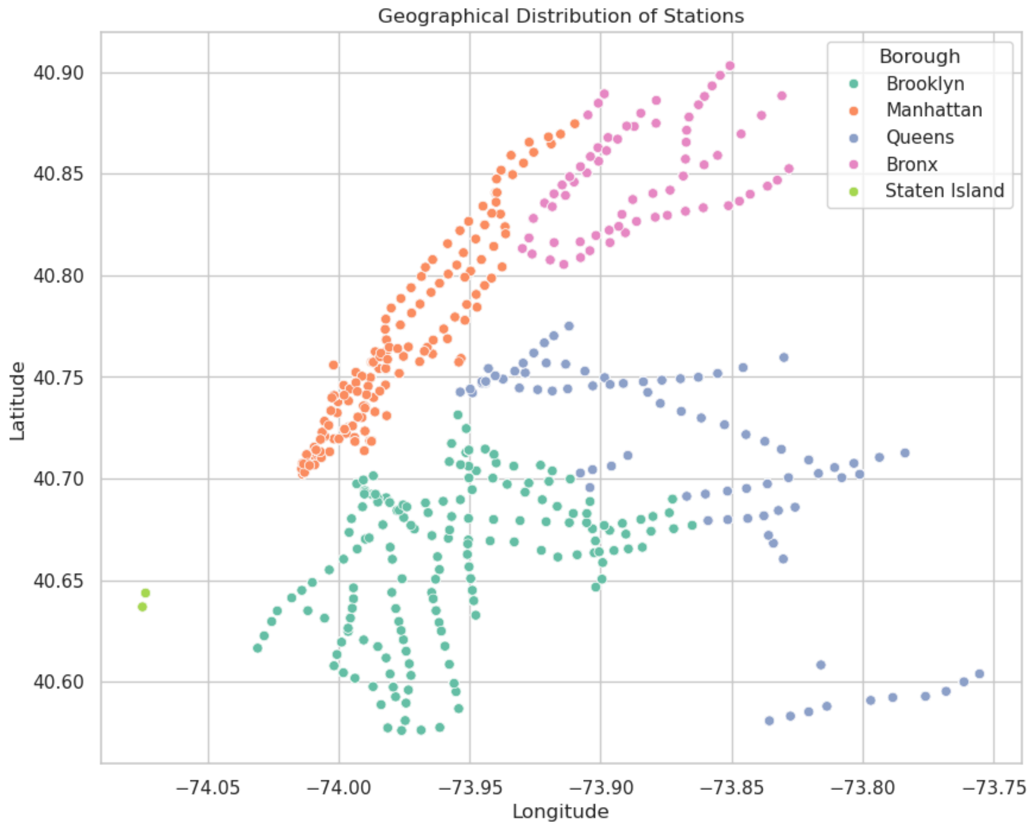
Ridership Distribution (Histogram & Boxplot)



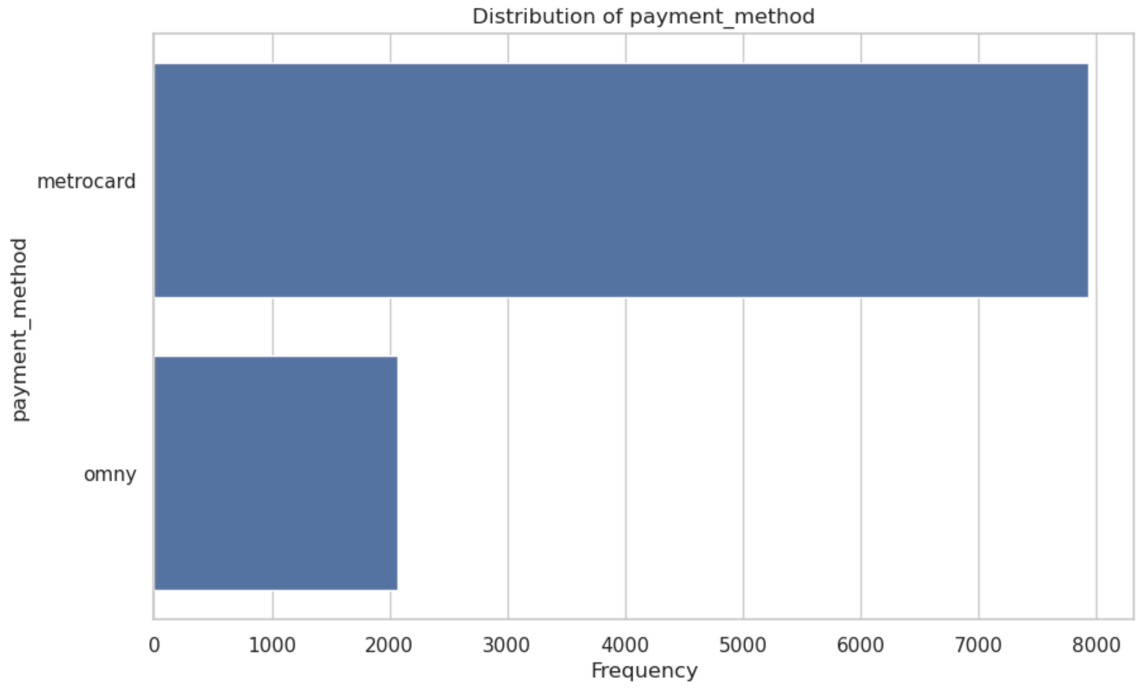
Ridership Analysis Overtime (Line Plot)



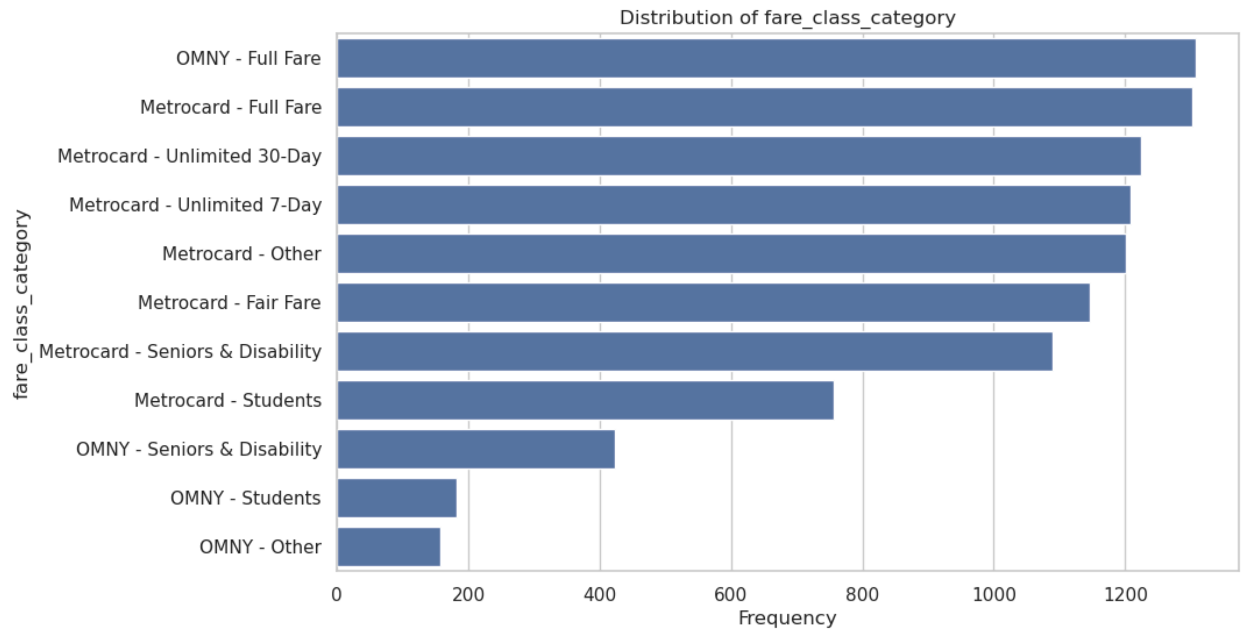
Geographical Representation of Subway Stations (Scatter Plot)



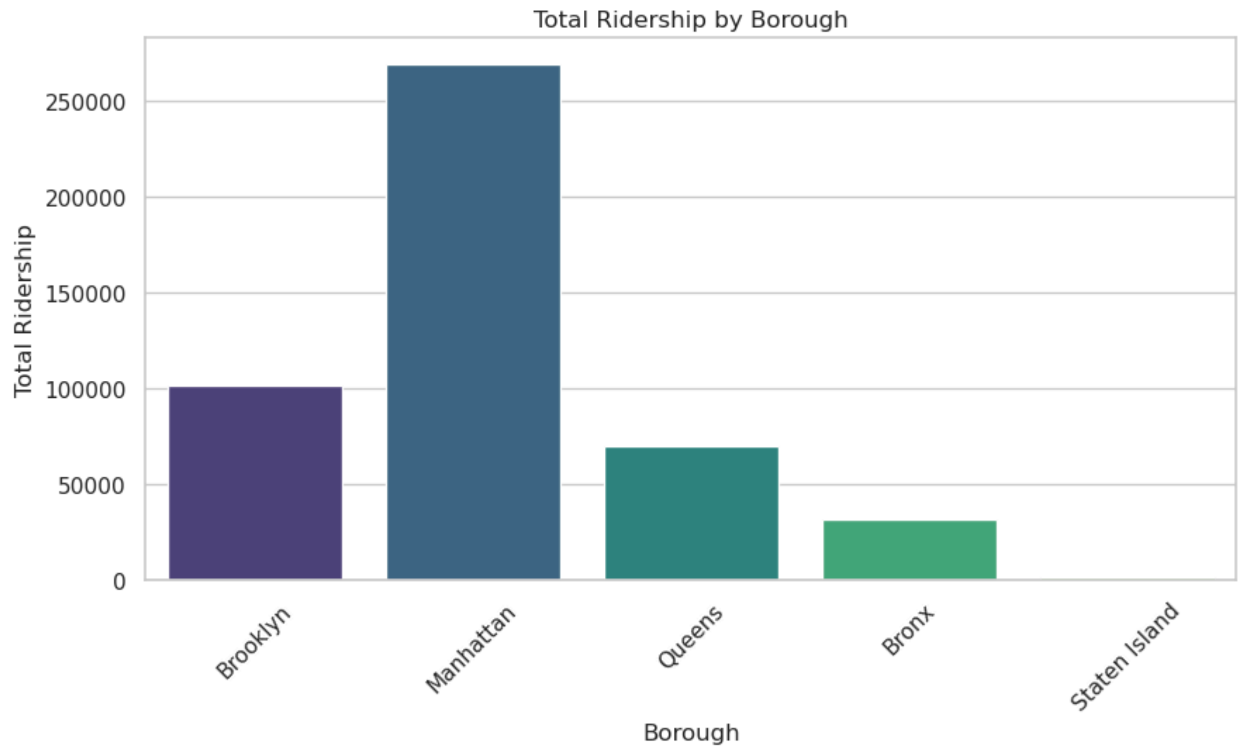
Payment Method Distribution



Class Category Distribution



Total Ridership by Borough



KEY FINDINGS

- Busiest Stations
  - `Times Sq-42 St` - 154,723,420 ridership
  - `Grand Central-42 St` - 105,899,395 ridership
  - `34 St-Herald Sq` - 89,210,247 ridership
- Borough-wise Ridership
  - Manhattan - 2,368,032,501
  - Brooklyn - 1,007,460,220
  - Bronx - 346,910,779
  - Queens - 677,083,907
  - Staten Island - 8,005,079
- Fare Class Category Analysis:
  - Highest ridership from `OMNY - Full Fare` (1,713,887,206).
  - High usage of `MetroCard - Full Fare` (1,044,386,772).
  - Unlimited MetroCards and `Fair Fare MetroCards` also have significant usage

HYPOTHESES & INSIGHTS

- `Times Sq-42 St` and `Grand Central-42 St` are critical transit hubs due to their central location.
- Ridership varies significantly by borough, with Manhattan showing the highest usage.
- Payment method distribution suggests high adoption of OMNY over MetroCard.
- Transfers correlate with major interchange stations (e.g., `Flushing-Main St, Jamaica Center`).

SUMMARY OF DATA

The MTA Subway Ridership dataset is a massive collection of over 110 million records from 2020 to 2024, covering details like transit modes, station complexes, boroughs, payment methods, fare categories, ridership numbers, transfers, and even geographical locations. Thankfully, the dataset is pretty clean, with almost no missing values. However, there are some quirks, like the `station_complex_id` column having non-numeric entries which need careful handling. Plus, the `transit_timestamp` column has inconsistent formatting, which makes working with time-based analysis a bit tricky.

ANTICIPATED CHALLENGES

When it comes to feature engineering, there are definitely a few hurdles to watch out for. High-cardinality categorical features like station_complex and borough could make building useful models more complicated. Making sure all the data types are consistent is also something to keep an eye on. And if I want to use this data for time-series analysis, I'll need to make sure those timestamps are properly cleaned and formatted. Things might get even more challenging when I try to combine this dataset with weather data to find correlations between transit ridership and weather conditions.

EXPLORATORY DATA ANALYSIS OUTPUTS

Summary statistics for int columns [ridership, transfers]

```
+-------+---------+---------+
|summary|ridership|transfers|
+-------+---------+---------+
|  count|110696365|110696365|
|    min|        1|        0|
|    max|    16217|     1450|
+-------+---------+---------+
```

Distinct Values in transit_mode

```
+---------------------+
|transit_mode         |
+---------------------+
|subway               |
|tram                 |
|staten_island_railway|
+---------------------+
```

Min & Max timestamp

```
+--------------------+--------------------+
|             MinDate|             MaxDate|
+--------------------+--------------------+
|01/01/2021 01:00:...|12/31/2024 12:00:...|
+--------------------+--------------------+
```

Top 10 busiest stations c

```
+--------------------+--------------+
|     station_complex|sum(ridership)|
+--------------------+--------------+
|Times Sq-42 St (N...|     154723420|
|Grand Central-42 ...|     105899395|
|34 St-Herald Sq (...|      89210247|
|14 St-Union Sq (L...|      79535793|
|Fulton St (A,C,J,...|      64709979|
|34 St-Penn Statio...|      62911729|
|59 St-Columbus Ci...|      58627877|
|34 St-Penn Statio...|      57325878|
|74-Broadway (7)/J...|      54144899|
|Flushing-Main St (7)|      53384616|
+--------------------+--------------+
```

Borough-wise Ridership

```
+-------------+--------------+
|      borough|sum(ridership)|
+-------------+--------------+
|     Brooklyn|    1007460220|
|    Manhattan|    2368032501|
|        Bronx|     346910779|
|       Queens|     677083907|
|Staten Island|       8005079|
+-------------+--------------+
```

Fare-Class-Category by Ridership

```
+--------------------+--------------+
| fare_class_category|sum(ridership)|
+--------------------+--------------+
|     OMNY - Students|      28161223|
|Metrocard - Fair ...|     144936001|
|OMNY - Seniors & ...|       6699026|
|Metrocard - Senio...|     157226817|
|Metrocard - Full ...|    1044386772|
|        OMNY - Other|       3454823|
|    OMNY - Full Fare|    1713887206|
|Metrocard - Unlim...|     499682220|
|Metrocard - Unlim...|     457027037|
|Metrocard - Students|     148635939|
|    OMNY - Fair Fare|          2655|
|   Metrocard - Other|     203392767|
+--------------------+--------------+
```

Top 10 stations with highest transfers

```
+--------------------+--------------+
|     station_complex|sum(transfers)|
+--------------------+--------------+
|Flushing-Main St (7)|      13745577|
|Jamaica Center-Pa...|       7659433|
|Kew Gardens-Union...|       5881825|
|74-Broadway (7)/J...|       5388608|
|  Jamaica-179 St (F)|       4960198|
|Lexington Av (N,R...|       4106715|
|14 St (F,M,1,2,3)...|       3837459|
|South Ferry (1)/W...|       3145754|
|Forest Hills-71 A...|       2985950|
|Flatbush Av-Brook...|       2805320|
+--------------------+--------------+
```

Supporting Dataset: Weather API

APPROACH

- Utilized Weather API data stored in Google Cloud Storage (GCS).
- Processed data in chunks to handle large volumes efficiently.
- Conducted Exploratory Data Analysis (EDA) on temperature, precipitation, and weather conditions.
- Filtered relevant features like temperature, precipitation, rain, snow depth, and weather codes.

KEY FINDINGS & INSIGHTS

- Significant missing values in key weather attributes.
- Dataset was sparsely populated
- All columns are float64 making it super efficient to use for model training
- Dropped columns such as `timezone` and `georeference` as the focus is only on the New York City areas where MTA Subway is functional

DATASET SCHEMA

```
|-- time: string (nullable = true)
|-- temperature_F: string (nullable = true)
|-- apparent_temperature_F: string (nullable = true)
|-- rain_mm: string (nullable = true)
|-- snowfall_cm: string (nullable = true)
|-- precipitation_mm: string (nullable = true)
```

Milestone 4: Feature Engineering

PREDICTION OBJECTIVE

This project predicts whether subway ridership at a specific station, date, and time will be higher than the system-wide median. A new label, `high_ridership`, was created (1 for high, 0 for normal or low). Predictions are based on time features, weather conditions, holidays, and peak hour indicators.

TABLE: COLUMNS, DATA TYPES & FEATURE ENGINEERING

COLUMN NAME	DATA TYPE	TREATMENT
hour	String	StringIndexer → OneHotEncoder
station_name	String	StringIndexer → OneHotEncoder
borough	String	StringIndexer → OneHotEncoder
transfers	Double	VectorAssembler + Scaler
is_holiday	Double	Binary feature
temperture_F	Double	VectorAssembler + Scaler
apparent_temperature_F	Double	VectorAssembler + Scaler
rain_mm	Double	VectorAssembler + Scaler
snowfall_cm	Double	VectorAssembler + Scaler
precipitation_mm	Double	VectorAssembler + Scaler
year	String	StringIndexer → OneHotEncoder
month	String	StringIndexer → OneHotEncoder
day_name	String	StringIndexer → OneHotEncoder
season	String	StringIndexer → OneHotEncoder
time_of_day	String	StringIndexer → OneHotEncoder
is_weekend	Double	Binary feature
is_peakhour	Double	Binary feature
ridership_z_score	Double	VectorAssembler + Scaler
high_ridership (target)	Double	Label Column for Classification

Note: Binary Features were passed directly to the model as they are machine learning ready.

STEPS TO FEATURE ENGINEER

1. Read and sample the subway and weather datasets.
2. Used holidays python package to generate `is_holiday` column
3. Standardize timestamps and clean the data.
4. Engineer features: holidays, seasons, peak hours, weather conditions.
5. Merge ridership and weather data by date and hour.
6. Remove outliers and create the `high_ridership` target.
7. Prepare features with indexing, encoding, and scaling.
    a. Saved the transformed_sdf in /trusted folder.
8. Train a logistic regression model with a machine learning pipeline.
9. Evaluate model performance and save results.
    a. Saved the metrics and test prediction as a .csv file.

CHALLENGES

During the project, I ran into a few challenges, starting with running out of cloud credits, which pushed me to switch to Colab and GitHub for handling my data. I downloaded about 25 chunks locally, uploaded them to Drive. Setting up Spark on Colab was smooth using the professor's code, but reading the timestamps properly was a challenge as Spark treated them differently. So I switched to pandas and pyarrow for the initial reads. I also struggled with the **holidays** library not working in Colab (maybe because of small sample sizes), but when I retried it on Dataproc with a larger dataset, it worked.. In the end, I dropped the `holiday_name` field since it didn't add much value for the model. I also tweaked the Z-score thresholds carefully — I wanted to keep most of the natural outliers because it's New York City, after all, and extreme ridership numbers are expected!

Additionally, while restarting the kernel and attempting to re-run all cells, I encountered an error with the **`holidays`** library, which could not be installed in the current session. To work around this, I switched to Google Colab, where I was able to successfully install the **`holidays`** package and extract the required data. I slightly extended the date range to ensure all relevant ridership dates were included. Once the dataset was ready, I exported it to a CSV file and uploaded it to my Google Cloud Storage. This allows me to read the file directly in the PySpark kernel in JupyterLab without needing to install the library locally.

LIBRARY AND PACKAGE REFERENCES

pandas **|** seaborn **|** matplotlib **|** holidays **|** pyarrow **|** os **|** pyspark.sql (like col, when, count, isnull, to_date, etc.) **|** pyspark.ml.feature(StringIndexer, OneHotEncoder, VectorAssembler, StandardScaler) **|** pyspark.ml.classification (LogisticReg model) **|** pyspark.ml **|** pyspark.ml.evaluation(BinaryClassificationEvaluator) **|** pyspark.ml.tuning (ParamGrid & CV)

MEDIAN AND GROUPING

I grouped the data by both `time_of_day` and `station_name` because ridership patterns can vary significantly depending on the time of day and the specific station. For example, what counts as "high" ridership at Times Square during rush hour might be completely normal there, but would be unusually high at a quieter station late at night. Grouping this way lets me compare each observation within its actual context. I also chose to use the **median** instead of the mean because the data tends to be skewed by occasional spikes—like special events, holidays, or disruptions. The median gives a more reliable baseline that isn't thrown off by those outliers, making the high-ridership label much more accurate and meaningful

SUMMARY OF OUTPUTS

Processed Dataset:
A cleaned and feature-engineered dataset was saved in the trusted folder as a Parquet file, containing weather features, holiday flags, time-based features, and engineered labels.

Trained Model:
A logistic regression model was trained using a machine learning pipeline and saved in the models folder for future use.

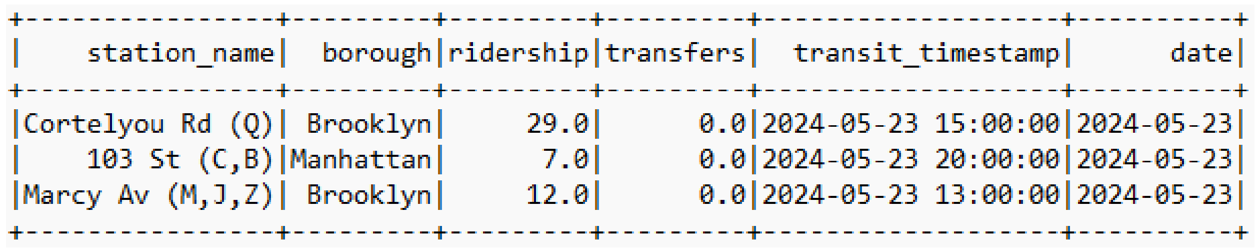
Model Evaluation Metrics:
Accuracy, Precision, Recall, and F1 Score were calculated and saved to a CSV file.

Test Predictions:
A sample of prediction results, including station name, day, holiday indicator, true label, and predicted label, was saved to a CSV file.

Specifics:
Num records before sampling → 34788721
Sampling fraction → 0.20
Num records after sampling → 6958512
Seed → 42
Train/Test Split → 70:30
Number of models to be tested → 6

Schema of ridership data:
```
 |-- transit_mode: string (nullable = true)
 |-- station_complex_id: long (nullable = true)
 |-- station_complex: string (nullable = true)
 |-- borough: string (nullable = true)
 |-- payment_method: string (nullable = true)
 |-- fare_class_category: string (nullable = true)
 |-- ridership: double (nullable = true)
 |-- transfers: double (nullable = true)
 |-- latitude: double (nullable = true)
 |-- longitude: double (nullable = true)
 |-- transit_timestamp: string (nullable = true)
```

Dataframe after dropping unnecessary columns

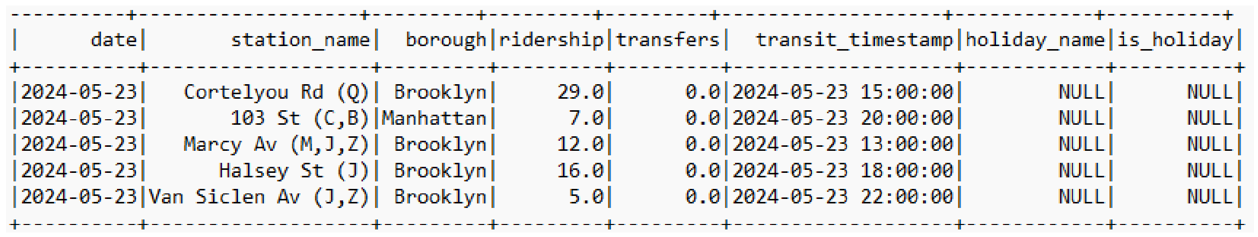


```
+----------------+---------+---------+---------+-------------------+----------+
|    station_name|  borough|ridership|transfers|  transit_timestamp|      date|
+----------------+---------+---------+---------+-------------------+----------+
|Cortelyou Rd (Q)| Brooklyn|     29.0|      0.0|2024-05-23 15:00:00|2024-05-23|
|     103 St (C,B)|Manhattan|      7.0|      0.0|2024-05-23 20:00:00|2024-05-23|
|Marcy Av (M,J,Z)| Brooklyn|     12.0|      0.0|2024-05-23 13:00:00|2024-05-23|
+----------------+---------+---------+---------+-------------------+----------+
```

Holiday dataframe and schema

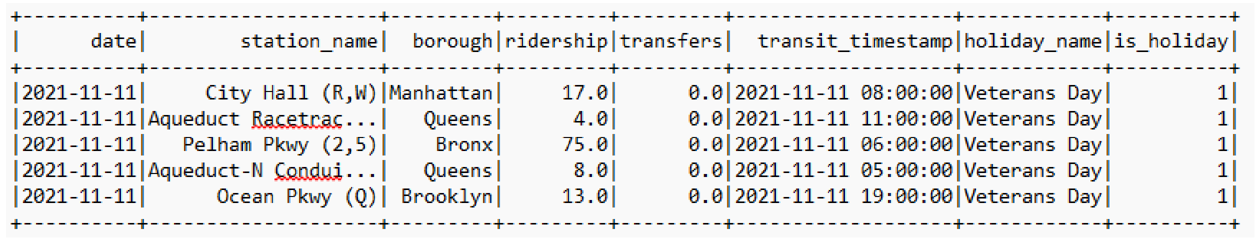
```
+----------+----------------+----------+
|date      |holiday_name    |is_holiday|
+----------+----------------+----------+
|2016-01-01|New Year's Day  |1         |
|2016-05-30|Memorial Day    |1         |
|2016-07-04|Independence Day|1         |
|2016-09-05|Labor Day       |1         |
|2016-11-11|Veterans Day    |1         |
+----------+----------------+----------+
|-- station_name: string (nullable = true)
|-- borough: string (nullable = true)
|-- ridership: double (nullable = true)
|-- transfers: double (nullable = true)
|-- transit_timestamp: string (nullable = true)
|-- date: date (nullable = true)
```

Ridership and Holiday merged (with NULL)



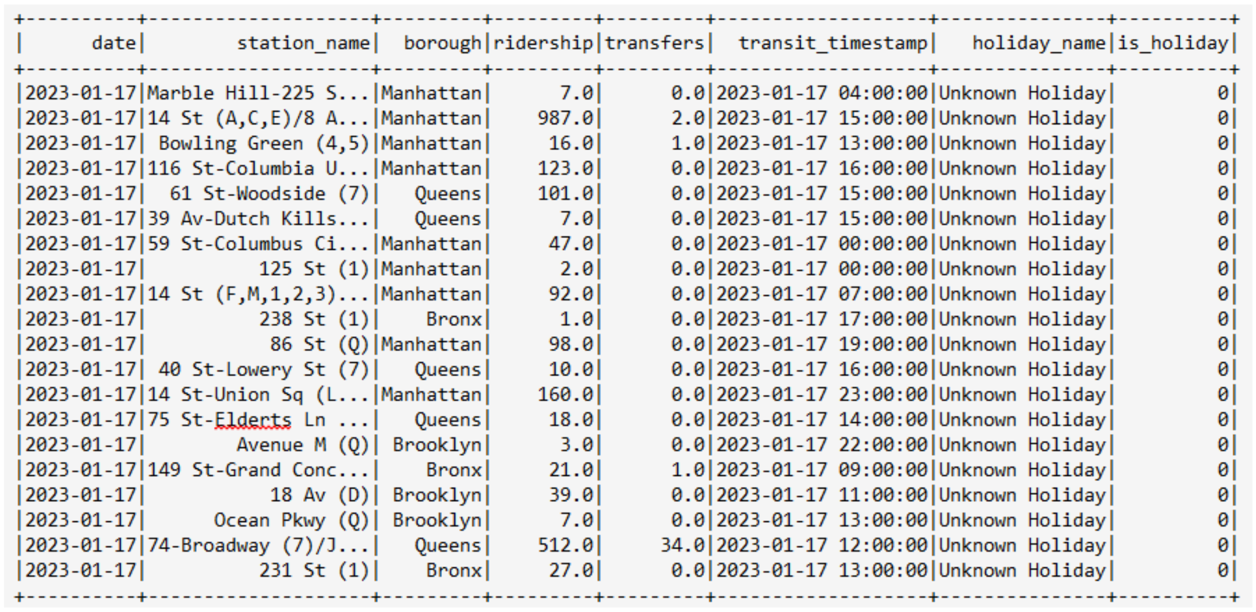
```
----------+------------------+---------+---------+---------+-------------------+------------+----------+
|      date|      station_name|  borough|ridership|transfers|  transit_timestamp|holiday_name|is_holiday|
+----------+------------------+---------+---------+---------+-------------------+------------+----------+
|2024-05-23|  Cortelyou Rd (Q)| Brooklyn|     29.0|      0.0|2024-05-23 15:00:00|        NULL|      NULL|
|2024-05-23|      103 St (C,B)|Manhattan|      7.0|      0.0|2024-05-23 20:00:00|        NULL|      NULL|
|2024-05-23|  Marcy Av (M,J,Z)| Brooklyn|     12.0|      0.0|2024-05-23 13:00:00|        NULL|      NULL|
|2024-05-23|     Halsey St (J)| Brooklyn|     16.0|      0.0|2024-05-23 18:00:00|        NULL|      NULL|
|2024-05-23|Van Siclen Av (J,Z)| Brooklyn|      5.0|      0.0|2024-05-23 22:00:00|        NULL|      NULL|
+----------+------------------+---------+---------+---------+-------------------+------------+----------+
```

Ridership and Holiday merged ("is_holiday = 1")



```
+----------+--------------------+---------+---------+---------+-------------------+------------+----------+
|      date|        station_name|  borough|ridership|transfers|  transit_timestamp|holiday_name|is_holiday|
+----------+--------------------+---------+---------+---------+-------------------+------------+----------+
|2021-11-11|     City Hall (R,W)|Manhattan|     17.0|      0.0|2021-11-11 08:00:00|Veterans Day|         1|
|2021-11-11|Aqueduct Racetrac...|   Queens|      4.0|      0.0|2021-11-11 11:00:00|Veterans Day|         1|
|2021-11-11|    Pelham Pkwy (2,5)|    Bronx|     75.0|      0.0|2021-11-11 06:00:00|Veterans Day|         1|
|2021-11-11|Aqueduct-N Condui...|   Queens|      8.0|      0.0|2021-11-11 05:00:00|Veterans Day|         1|
|2021-11-11|      Ocean Pkwy (Q)| Brooklyn|     13.0|      0.0|2021-11-11 19:00:00|Veterans Day|         1|
+----------+--------------------+---------+---------+---------+-------------------+------------+----------+
```

Ridership and Holiday merged (after replacing Nulls with 0s)

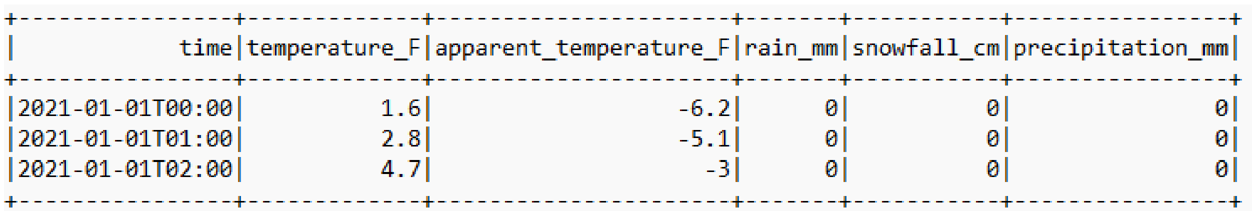


```
+----------+------------------+--------+---------+---------+-------------------+---------------+----------+
|      date|      station_name| borough|ridership|transfers|  transit_timestamp|   holiday_name|is_holiday|
+----------+------------------+--------+---------+---------+-------------------+---------------+----------+
|2023-01-17|Marble Hill-225 S...|Manhattan|      7.0|      0.0|2023-01-17 04:00:00|Unknown Holiday|         0|
|2023-01-17|14 St (A,C,E)/8 A...|Manhattan|    987.0|      2.0|2023-01-17 15:00:00|Unknown Holiday|         0|
|2023-01-17| Bowling Green (4,5)|Manhattan|     16.0|      1.0|2023-01-17 13:00:00|Unknown Holiday|         0|
|2023-01-17|116 St-Columbia U...|Manhattan|    123.0|      0.0|2023-01-17 16:00:00|Unknown Holiday|         0|
|2023-01-17|  61 St-Woodside (7)|   Queens|    101.0|      0.0|2023-01-17 15:00:00|Unknown Holiday|         0|
|2023-01-17|39 Av-Dutch Kills...|   Queens|      7.0|      0.0|2023-01-17 15:00:00|Unknown Holiday|         0|
|2023-01-17|59 St-Columbus Ci...|Manhattan|     47.0|      0.0|2023-01-17 00:00:00|Unknown Holiday|         0|
|2023-01-17|          125 St (1)|Manhattan|      2.0|      0.0|2023-01-17 00:00:00|Unknown Holiday|         0|
|2023-01-17|14 St (F,M,1,2,3)...|Manhattan|     92.0|      0.0|2023-01-17 07:00:00|Unknown Holiday|         0|
|2023-01-17|          238 St (1)|    Bronx|      1.0|      0.0|2023-01-17 17:00:00|Unknown Holiday|         0|
|2023-01-17|           86 St (Q)|Manhattan|     98.0|      0.0|2023-01-17 19:00:00|Unknown Holiday|         0|
|2023-01-17| 40 St-Lowery St (7)|   Queens|     10.0|      0.0|2023-01-17 16:00:00|Unknown Holiday|         0|
|2023-01-17|14 St-Union Sq (L...|Manhattan|    160.0|      0.0|2023-01-17 23:00:00|Unknown Holiday|         0|
|2023-01-17|75 St-Elderts Ln ...|   Queens|     18.0|      0.0|2023-01-17 14:00:00|Unknown Holiday|         0|
|2023-01-17|        Avenue M (Q)| Brooklyn|      3.0|      0.0|2023-01-17 22:00:00|Unknown Holiday|         0|
|2023-01-17|149 St-Grand Conc...|    Bronx|     21.0|      1.0|2023-01-17 09:00:00|Unknown Holiday|         0|
|2023-01-17|           18 Av (D)| Brooklyn|     39.0|      0.0|2023-01-17 11:00:00|Unknown Holiday|         0|
|2023-01-17|       Ocean Pkwy (Q)| Brooklyn|      7.0|      0.0|2023-01-17 13:00:00|Unknown Holiday|         0|
|2023-01-17|74-Broadway (7)/J...|   Queens|    512.0|     34.0|2023-01-17 12:00:00|Unknown Holiday|         0|
|2023-01-17|          231 St (1)|    Bronx|     27.0|      0.0|2023-01-17 13:00:00|Unknown Holiday|         0|
+----------+------------------+--------+---------+---------+-------------------+---------------+----------+
```

Ridership and Holiday merged (class distribution)

```
+----------+-------+
|is_holiday|  count|
+----------+-------+
|         1| 128994|
|         0|6829518|
+----------+-------+
```

Weather data (head)



```
+----------------+-------------+---------------------+-------+-----------+----------------+
|            time|temperature_F|apparent_temperature_F|rain_mm|snowfall_cm|precipitation_mm|
+----------------+-------------+---------------------+-------+-----------+----------------+
|2021-01-01T00:00|          1.6|                 -6.2|      0|          0|               0|
|2021-01-01T01:00|          2.8|                 -5.1|      0|          0|               0|
|2021-01-01T02:00|          4.7|                   -3|      0|          0|               0|
+----------------+-------------+---------------------+-------+-----------+----------------+
```

Weather data (where precipitation is greater than 0 or temperature is not constant)

```
+----------------+-------------+----------------------+-------+-----------+----------------+
|            time|temperature_F|apparent_temperature_F|rain_mm|snowfall_cm|precipitation_mm|
+----------------+-------------+----------------------+-------+-----------+----------------+
|2021-01-01T00:00|          1.6|                  -6.2|      0|          0|               0|
|2021-01-01T01:00|          2.8|                  -5.1|      0|          0|               0|
|2021-01-01T02:00|          4.7|                    -3|      0|          0|               0|
|2021-01-01T03:00|          5.7|                  -2.4|      0|          0|               0|
|2021-01-01T04:00|          6.1|                  -2.5|      0|          0|               0|
|2021-01-01T05:00|            6|                  -2.4|      0|          0|               0|
|2021-01-01T06:00|          5.2|                  -3.1|      0|          0|               0|
|2021-01-01T07:00|          4.4|                  -3.5|      0|          0|               0|
|2021-01-01T08:00|          3.8|                    -4|      0|          0|               0|
|2021-01-01T09:00|          2.4|                  -5.3|      0|          0|               0|
+----------------+-------------+----------------------+-------+-----------+----------------+
only showing top 10 rows
```

Weather data (sorting by precipitation or specific weather events like rain/snow)

```
+----------------+-------------+----------------------+-------+-----------+----------------+
|            time|temperature_F|apparent_temperature_F|rain_mm|snowfall_cm|precipitation_mm|
+----------------+-------------+----------------------+-------+-----------+----------------+
|2024-05-18T08:00|         26.3|                  21.1|    2.8|       2.66|             6.6|
|2022-08-07T14:00|         33.9|                  29.4|    3.3|       1.96|             6.1|
|2021-07-28T04:00|         51.3|                    49|      6|          0|               6|
|2023-10-31T08:00|         21.7|                  13.6|    0.1|       4.13|               6|
|2024-09-26T11:00|         26.1|                  20.5|    0.4|       3.57|             5.5|
|2022-04-27T01:00|         24.5|                  18.1|      0|       3.78|             5.4|
|2022-04-27T00:00|         24.2|                  17.9|      0|       3.78|             5.4|
|2022-06-21T04:00|         36.6|                  31.5|    1.8|       2.45|             5.3|
|2023-05-20T23:00|           27|                  21.1|      0|       3.71|             5.3|
|2024-09-26T12:00|         25.6|                  20.2|    0.2|       3.29|             4.9|
+----------------+-------------+----------------------+-------+-----------+----------------+
```

Weather data (descriptive statistics)

```
+-------+-----------------+----------------------+--------------------+--------------------+------------------+
|summary|    temperature_F|apparent_temperature_F|             rain_mm|         snowfall_cm|  precipitation_mm|
+-------+-----------------+----------------------+--------------------+--------------------+------------------+
|  count|            35064|                 35064|               35064|               35064|             35064|
|   mean|21.18750570385588|    14.654631530914894|0.03257186858316248|0.059998288843258085|0.1181582249600676|
| stddev|16.880022766257227|    18.268551486494243|0.18352017022346923| 0.22465070790273348|0.3766533453753407|
|    min|             -0.1|                  -0.1|                   0|                   0|                 0|
|    max|              9.9|                   9.9|                   6|                4.13|               6.6|
+-------+-----------------+----------------------+--------------------+--------------------+------------------+
```

Merged dataframe (Ridership + Holiday + Weather + Date features extracted)

```
 |-- date: date (nullable = true)
 |-- hour: integer (nullable = true)
 |-- station_name: string (nullable = true)
 |-- borough: string (nullable = true)
 |-- ridership: double (nullable = true)
 |-- transfers: double (nullable = true)
 |-- transit_timestamp: string (nullable = true)
 |-- holiday_name: string (nullable = false)
 |-- is_holiday: long (nullable = false)
 |-- time: string (nullable = true)
 |-- temperature_F: string (nullable = true)
 |-- apparent_temperature_F: string (nullable = true)
 |-- rain_mm: string (nullable = true)
 |-- snowfall_cm: string (nullable = true)
 |-- precipitation_mm: string (nullable = true)
 |-- year: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- day_name: string (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- season: string (nullable = false)
 |-- time_of_day: string (nullable = false)
 |-- is_weekend: integer (nullable = false)
 |-- is_peakhour: integer (nullable = false)
```

high_ridership class distribution (Target Variable)

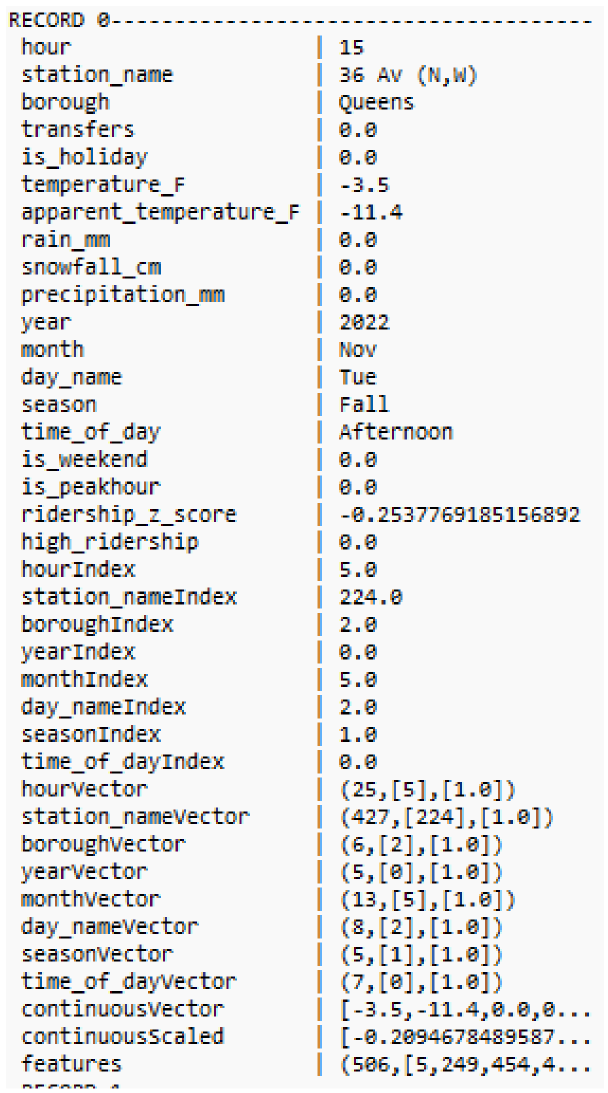
```
+--------------+------+-------+
|high_ridership| count|percent|
+--------------+------+-------+
|             1|166590|  48.19|
|             0|179137|  51.81|
+--------------+------+-------+
```

Final schema after dropping columns

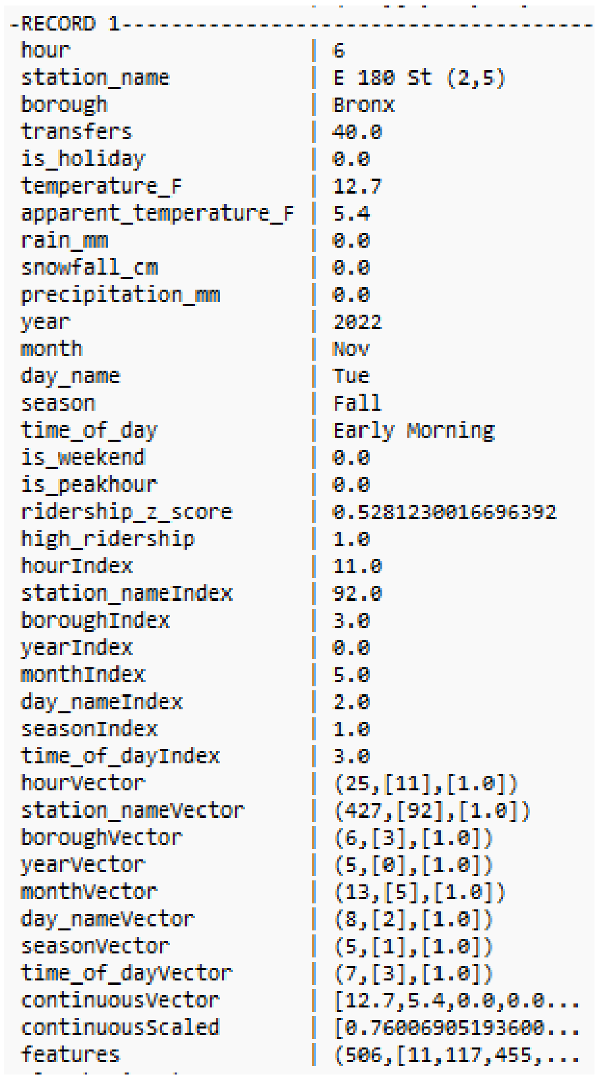
```
 |-- time_of_day: string (nullable = false)
 |-- station_name: string (nullable = true)
 |-- hour: integer (nullable = true)
 |-- borough: string (nullable = true)
 |-- transfers: double (nullable = true)
 |-- is_holiday: long (nullable = false)
 |-- temperature_F: string (nullable = true)
 |-- apparent_temperature_F: string (nullable = true)
 |-- rain_mm: string (nullable = true)
 |-- snowfall_cm: string (nullable = true)
 |-- precipitation_mm: string (nullable = true)
 |-- year: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- day_name: string (nullable = true)
```

```
|-- season: string (nullable = false)
|-- is_weekend: integer (nullable = false)
|-- is_peakhour: integer (nullable = false)
|-- ridership_z_score: double (nullable = true)
|-- high_ridership: integer (nullable = false)
```

Display of transformed_sdf

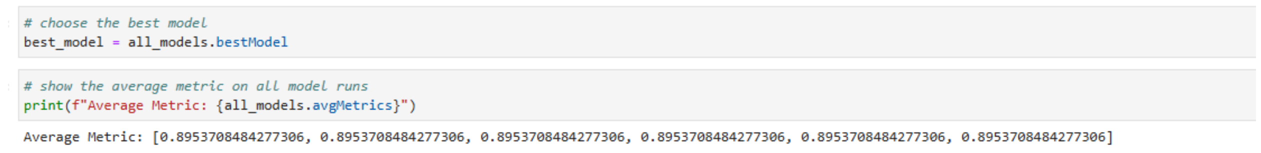


```
RECORD 0--------------------------------------
 hour                    | 15
 station_name            | 36 Av (N,W)
 borough                 | Queens
 transfers               | 0.0
 is_holiday              | 0.0
 temperature_F           | -3.5
 apparent_temperature_F  | -11.4
 rain_mm                 | 0.0
 snowfall_cm             | 0.0
 precipitation_mm        | 0.0
 year                    | 2022
 month                   | Nov
 day_name                | Tue
 season                  | Fall
 time_of_day             | Afternoon
 is_weekend              | 0.0
 is_peakhour             | 0.0
 ridership_z_score       | -0.2537769185156892
 high_ridership          | 0.0
 hourIndex               | 5.0
 station_nameIndex       | 224.0
 boroughIndex            | 2.0
 yearIndex               | 0.0
 monthIndex              | 5.0
 day_nameIndex           | 2.0
 seasonIndex             | 1.0
 time_of_dayIndex        | 0.0
 hourVector              | (25,[5],[1.0])
 station_nameVector      | (427,[224],[1.0])
 boroughVector           | (6,[2],[1.0])
 yearVector              | (5,[0],[1.0])
 monthVector             | (13,[5],[1.0])
 day_nameVector          | (8,[2],[1.0])
 seasonVector            | (5,[1],[1.0])
 time_of_dayVector       | (7,[0],[1.0])
 continuousVector        | [-3.5,-11.4,0.0,0...
 continuousScaled        | [-0.2094678489587...
 features                | (506,[5,249,454,4...
```



```
-RECORD 1--------------------------------------
 hour                    | 6
 station_name            | E 180 St (2,5)
 borough                 | Bronx
 transfers               | 40.0
 is_holiday              | 0.0
 temperature_F           | 12.7
 apparent_temperature_F  | 5.4
 rain_mm                 | 0.0
 snowfall_cm             | 0.0
 precipitation_mm        | 0.0
 year                    | 2022
 month                   | Nov
 day_name                | Tue
 season                  | Fall
 time_of_day             | Early Morning
 is_weekend              | 0.0
 is_peakhour             | 0.0
 ridership_z_score       | 0.5281230016696392
 high_ridership          | 1.0
 hourIndex               | 11.0
 station_nameIndex       | 92.0
 boroughIndex            | 3.0
 yearIndex               | 0.0
 monthIndex              | 5.0
 day_nameIndex           | 2.0
 seasonIndex             | 1.0
 time_of_dayIndex        | 3.0
 hourVector              | (25,[11],[1.0])
 station_nameVector      | (427,[92],[1.0])
 boroughVector           | (6,[3],[1.0])
 yearVector              | (5,[0],[1.0])
 monthVector             | (13,[5],[1.0])
 day_nameVector          | (8,[2],[1.0])
 seasonVector            | (5,[1],[1.0])
 time_of_dayVector       | (7,[3],[1.0])
 continuousVector        | [12.7,5.4,0.0,0.0...
 continuousScaled        | [0.76006905193600...
 features                | (506,[11,117,455,...
```

Average Metrics Results



```
# choose the best model
best_model = all_models.bestModel
```

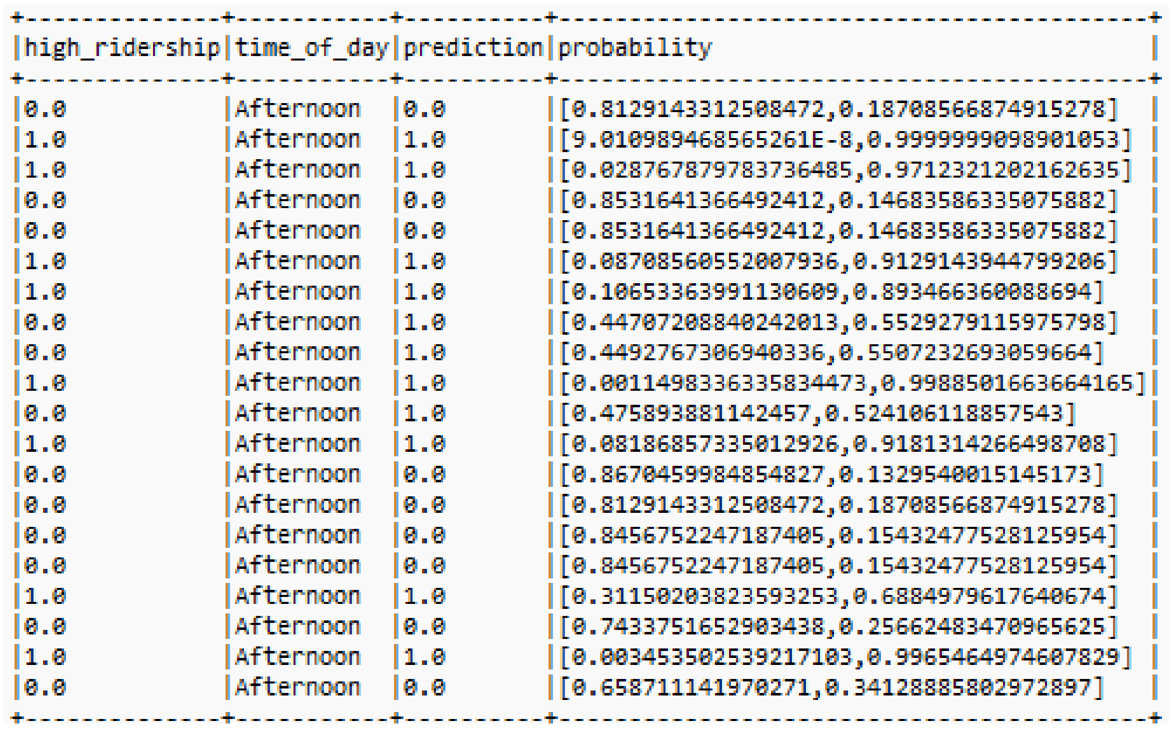
```
# show the average metric on all model runs
print(f"Average Metric: {all_models.avgMetrics}")
```

```
Average Metric: [0.8953708484277306, 0.8953708484277306, 0.8953708484277306, 0.8953708484277306, 0.8953708484277306, 0.8953708484277306]
```

Predicted test results

```
+------------+-----------+----------+--------------+----------+
|station_name|time_of_day|is_holiday|high_ridership|prediction|
+------------+-----------+----------+--------------+----------+
|1 Av (L)    |Afternoon  |0.0       |0.0           |0.0       |
|1 Av (L)    |Afternoon  |0.0       |1.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |1.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |0.0           |0.0       |
|1 Av (L)    |Afternoon  |0.0       |0.0           |0.0       |
|1 Av (L)    |Afternoon  |0.0       |1.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |1.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |0.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |0.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |1.0           |1.0       |
|1 Av (L)    |Afternoon  |0.0       |0.0           |1.0       |
|103 St (1)  |Afternoon  |0.0       |1.0           |1.0       |
|103 St (1)  |Afternoon  |0.0       |0.0           |0.0       |
|103 St (1)  |Afternoon  |0.0       |0.0           |0.0       |
|103 St (1)  |Afternoon  |0.0       |0.0           |0.0       |
|103 St (1)  |Afternoon  |0.0       |0.0           |0.0       |
|103 St (1)  |Afternoon  |0.0       |1.0           |1.0       |
|103 St (1)  |Afternoon  |0.0       |0.0           |0.0       |
|103 St (1)  |Afternoon  |0.0       |1.0           |1.0       |
|103 St (1)  |Afternoon  |0.0       |0.0           |0.0       |
+------------+-----------+----------+--------------+----------+
```

Predicted test results (with probabilities)



```
+--------------+-----------+----------+------------------------------------------+
|high_ridership|time_of_day|prediction|probability                               |
+--------------+-----------+----------+------------------------------------------+
|0.0           |Afternoon  |0.0       |[0.8129143312508472,0.18708566874915278]  |
|1.0           |Afternoon  |1.0       |[9.010989468565261E-8,0.9999999098901053] |
|1.0           |Afternoon  |1.0       |[0.028767879783736485,0.9712321202162635] |
|0.0           |Afternoon  |0.0       |[0.8531641366492412,0.14683586335075882]  |
|0.0           |Afternoon  |0.0       |[0.8531641366492412,0.14683586335075882]  |
|1.0           |Afternoon  |1.0       |[0.08708560552007936,0.9129143944799206]  |
|1.0           |Afternoon  |1.0       |[0.10653363991130609,0.893466360088694]   |
|0.0           |Afternoon  |1.0       |[0.44707208840242013,0.5529279115975798]  |
|0.0           |Afternoon  |1.0       |[0.4492767306940336,0.5507232693059664]   |
|1.0           |Afternoon  |1.0       |[0.0011498336335834473,0.9988501663664165]|
|0.0           |Afternoon  |1.0       |[0.475893881142457,0.524106118857543]     |
|1.0           |Afternoon  |1.0       |[0.08186857335012926,0.9181314266498708]  |
|0.0           |Afternoon  |0.0       |[0.8670459984854827,0.1329540015145173]   |
|0.0           |Afternoon  |0.0       |[0.8129143312508472,0.18708566874915278]  |
|0.0           |Afternoon  |0.0       |[0.8456752247187405,0.15432477528125954]  |
|0.0           |Afternoon  |0.0       |[0.8456752247187405,0.15432477528125954]  |
|1.0           |Afternoon  |1.0       |[0.31150203823593253,0.6884979617640674]  |
|0.0           |Afternoon  |0.0       |[0.7433751652903438,0.25662483470965625]  |
|1.0           |Afternoon  |1.0       |[0.003453502539217103,0.9965464974607829] |
|0.0           |Afternoon  |0.0       |[0.658711141970271,0.34128885802972897]   |
+--------------+-----------+----------+------------------------------------------+
```

Confusion Matrix Results

```
[Row(high_ridership=0.0, 0.0=49259, 1.0=4201),
 Row(high_ridership=1.0, 0.0=16963, 1.0=32795)]

label,0.0,1.0
 0.0 , 49259 , 4201
 1.0 , 16963 , 32795
```

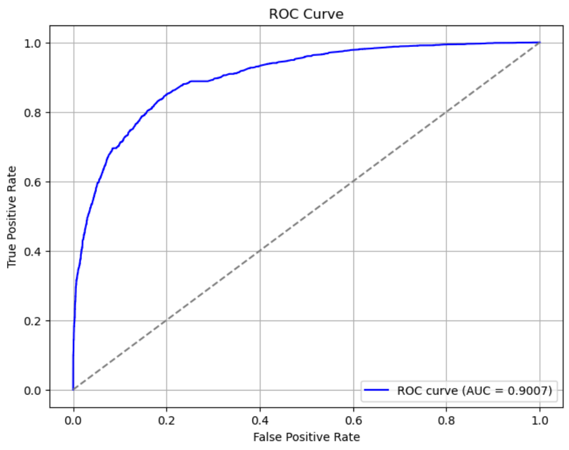
Other Metrics

```
accuracy, precision, recall, f1_score
0.7949582437171812 0.8864471834792951 0.6590899955786004 0.7560458307397929
```

Milestone 5: Data Visualization

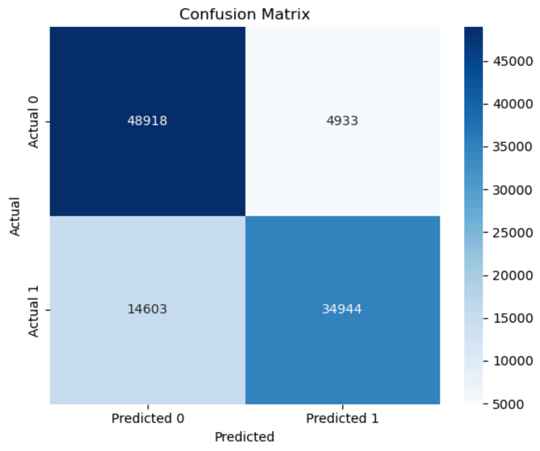
ROC CURVE

When I ran the ROC curve on my model, the accuracy achieved was 0.8955. To visualize the performance, I used a 10% sample of my DataFrame and plotted the ROC curve using the Python library matplotlib. As shown in the plot, the AUC (Area Under the Curve) is approximately 0.9007, which indicates strong model performance. The ROC curve clearly bows towards the top-left corner, suggesting a high true positive rate and a low false positive rate. It's important to note that this curve represents the model's performance on the sampled dataset used specifically for plotting purposes.



CONFUSION MATRIX

This confusion matrix is also plotted from the sample dataframe and here's the results. Overall model performance is good with it correctly identifying most negative cases (48,918) and a good number of positives (34,944). False positives are relatively low (4,933), indicating strong precision. However, it misses 14,603 actual positives, showing room to improve recall. The model leans slightly toward predicting the negative class.



TEST PREDICTIONS AND VISUALIZATIONS

All subsequent visualizations are based on `test_results`, generated after the feature engineering and model evaluation steps in the previous milestone. The data was loaded from the predictions_df.csv file, which was saved in the test_results/ folder on Google Cloud Storage.

Here's the schema:

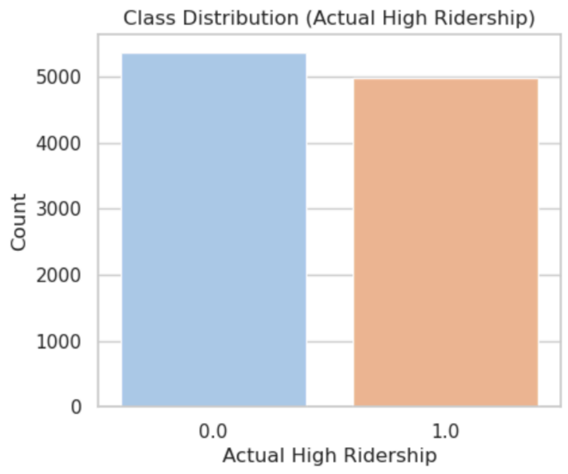
```
 |-- time_of_day: string (nullable = true)
 |-- station_name: string (nullable = true)
 |-- hour: integer (nullable = true)
 |-- borough: string (nullable = true)
 |-- transfers: double (nullable = true)
 |-- is_holiday: double (nullable = true)
 |-- temperature_F: double (nullable = true)
 |-- apparent_temperature_F: double (nullable = true)
 |-- rain_mm: double (nullable = true)
 |-- snowfall_cm: double (nullable = true)
 |-- precipitation_mm: double (nullable = true)
 |-- year: integer (nullable = true)
 |-- month: string (nullable = true)
 |-- day_name: string (nullable = true)
 |-- season: string (nullable = true)
 |-- is_weekend: double (nullable = true)
 |-- is_peakhour: double (nullable = true)
 |-- ridership_z_score: double (nullable = true)
 |-- high_ridership: double (nullable = true)
 |-- time_of_dayIndex: double (nullable = true)
 |-- station_nameIndex: double (nullable = true)
 |-- hourIndex: double (nullable = true)
 |-- boroughIndex: double (nullable = true)
 |-- yearIndex: double (nullable = true)
 |-- monthIndex: double (nullable = true)
 |-- day_nameIndex: double (nullable = true)
 |-- seasonIndex: double (nullable = true)
 |-- prediction: double (nullable = true)
```

To calculate the visualizations, I went ahead and dropped all the `index` columns while keeping my target variable `high_ridership` and `prediction` columns to visualise them.

ANALYSIS OF PREDICTION CLASSES

CLASS DISTRIBUTIONS (ACTUAL HIGH RIDERSHIP)

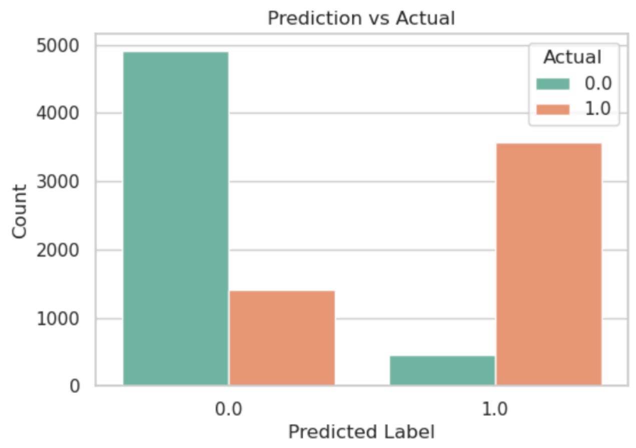
The class distribution chart shows a relatively balanced dataset, with slightly more instances labeled as low ridership (0) than high ridership (1). This balance helps ensure that the model does not become biased toward one class, supporting fair evaluation metrics.

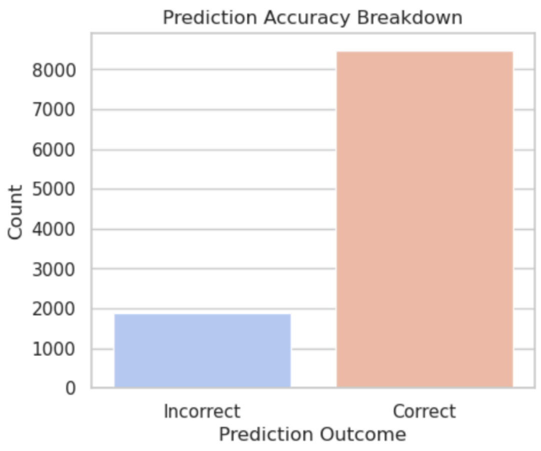


PREDICTION VS ACTUAL & ACCURACY BREAKDOWN

This plot compares predicted labels against actual values, revealing that the model performs better at correctly identifying low ridership instances (0) compared to high ridership ones (1). A noticeable portion of actual high-ridership cases are misclassified as low, highlighting a gap in recall for the positive class. This could impact model performance if capturing high ridership accurately is a priority.

The accuracy breakdown shows that the majority of predictions are correct, with relatively fewer incorrect classifications.

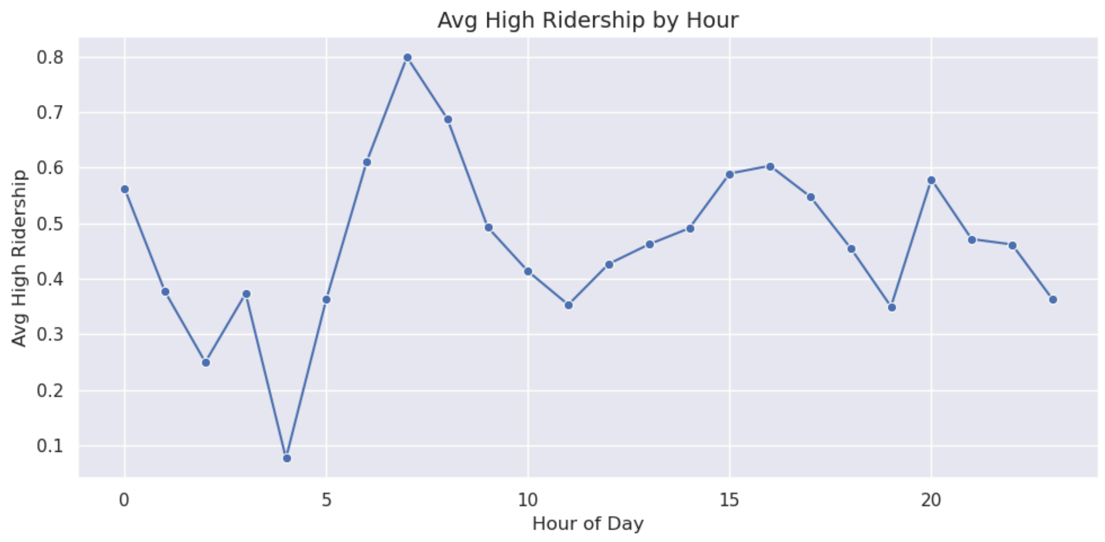




ANALYSIS OF RIDERSHIP

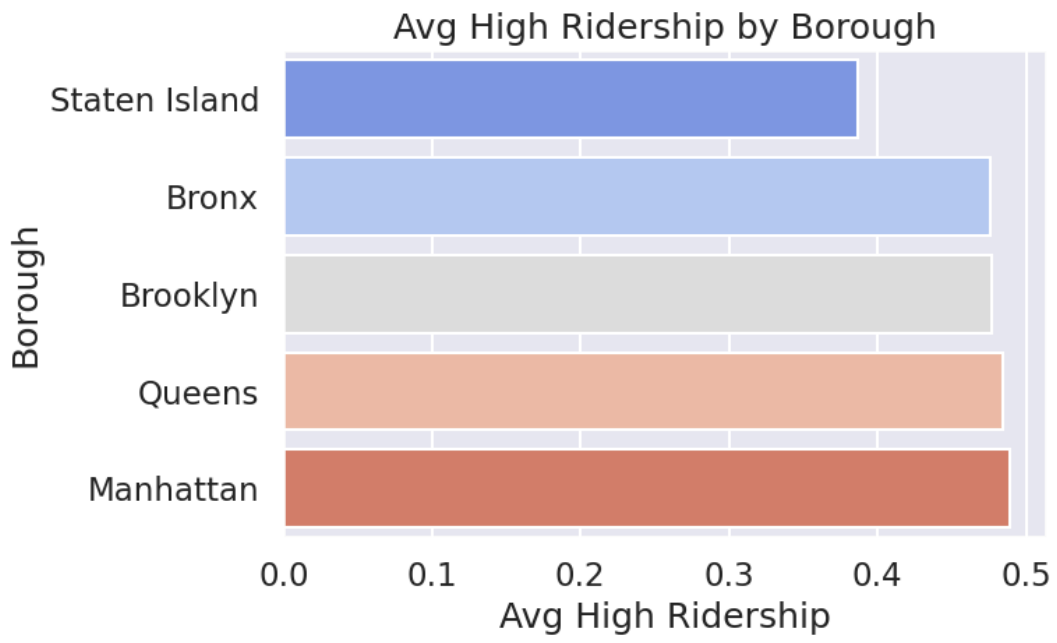
AVG HIGH RIDERSHIP BY HOUR

Insight: High ridership peaks sharply around 7–8 AM, indicating strong morning rush hour activity. A smaller peak is also visible around 4–5 PM, aligning with evening commute times. Early morning (around 4 AM) shows the lowest average high ridership.



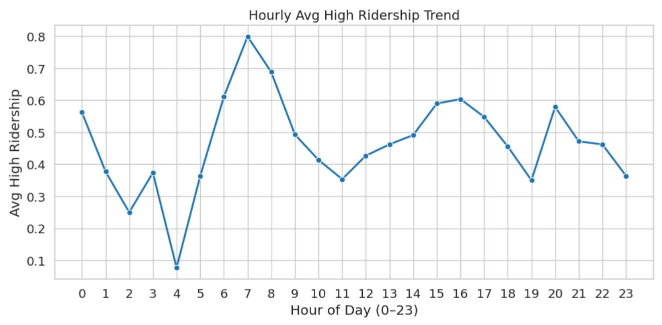
AVG HIGH RIDERSHIP BY BOROUGH

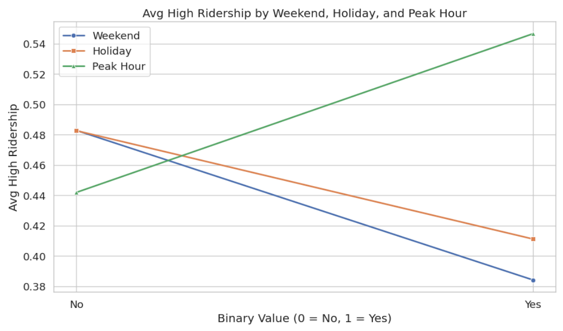
Insight: Manhattan leads in average high ridership, followed by Queens and Brooklyn, reflecting the borough's commercial density and commuter volume. Staten Island consistently shows the lowest high ridership.



IMPACT OF TIME SEGMENTS & SPECIAL DAYS

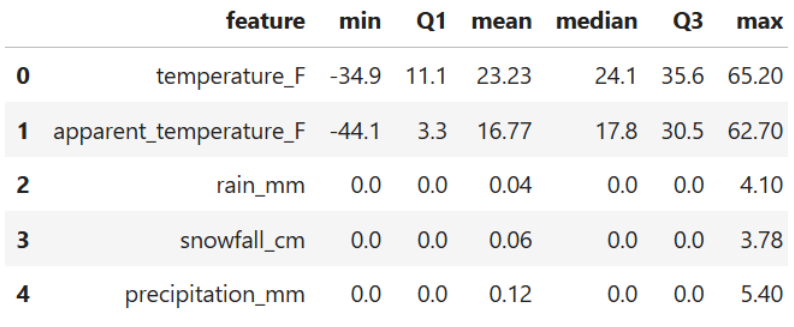
Insight: High ridership is more likely during peak hours and weekdays. Weekends and holidays show noticeably lower averages, suggesting a drop in transit demand during non-working periods.





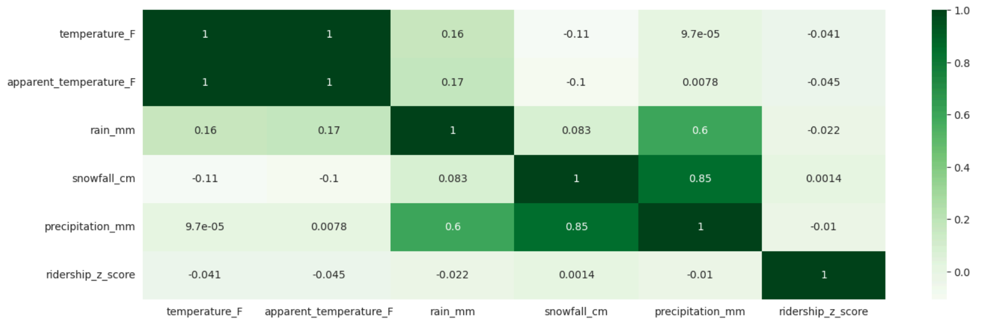
WEATHER FEATURE SUMMARY

Insight: The dataset includes five weather variables: temperature, apparent temperature, rain, snowfall, and overall precipitation. Most observations for rain, snowfall, and precipitation are zero, as seen from their median and Q3 values, indicating a high frequency of dry days. Temperature variables are normally distributed, ranging widely from extreme cold (as low as -44.1°F) to moderate warmth (~65°F).



	feature	min	Q1	mean	median	Q3	max
0	temperature_F	-34.9	11.1	23.23	24.1	35.6	65.20
1	apparent_temperature_F	-44.1	3.3	16.77	17.8	30.5	62.70
2	rain_mm	0.0	0.0	0.04	0.0	0.0	4.10
3	snowfall_cm	0.0	0.0	0.06	0.0	0.0	3.78
4	precipitation_mm	0.0	0.0	0.12	0.0	0.0	5.40

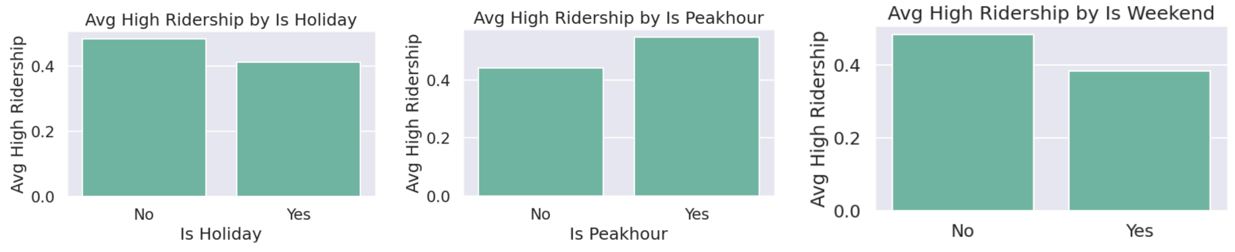
Correlation analysis shows strong collinearity between temperature and apparent temperature (1.0), and between snowfall and precipitation (0.85), suggesting some redundancy. However, all weather features exhibit weak correlation with ridership (|r| < 0.05), implying limited direct influence of weather conditions on high ridership patterns.



	temperature_F	apparent_temperature_F	rain_mm	snowfall_cm	precipitation_mm	ridership_z_score
temperature_F	1	1	0.16	-0.11	9.7e-05	-0.041
apparent_temperature_F	1	1	0.17	-0.1	0.0078	-0.045
rain_mm	0.16	0.17	1	0.083	0.6	-0.022
snowfall_cm	-0.11	-0.1	0.083	1	0.85	0.0014
precipitation_mm	9.7e-05	0.0078	0.6	0.85	1	-0.01
ridership_z_score	-0.041	-0.045	-0.022	0.0014	-0.01	1

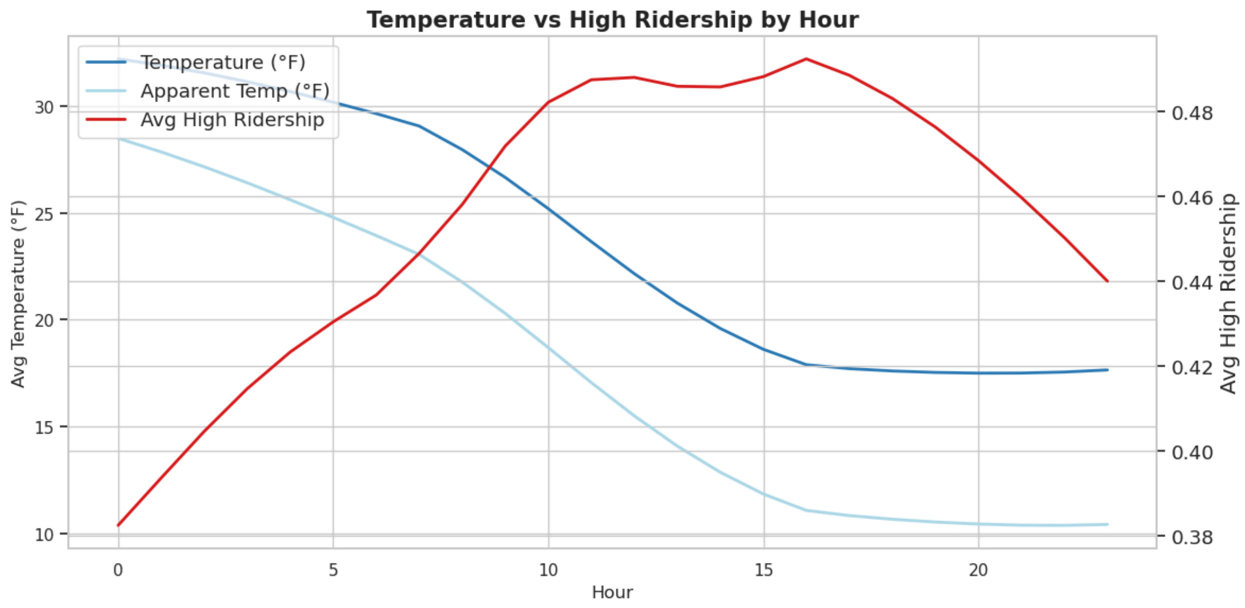
TEMPORAL IMPACT ON HIGH RIDERSHIP

Insight: High ridership is more frequent on non-holidays and weekdays, suggesting that regular workdays drive greater subway usage. In contrast, both holidays and weekends show a drop in high ridership, likely due to reduced commuter traffic. Peak hours consistently show higher average high ridership compared to non-peak times, confirming expected rush-hour demand patterns.



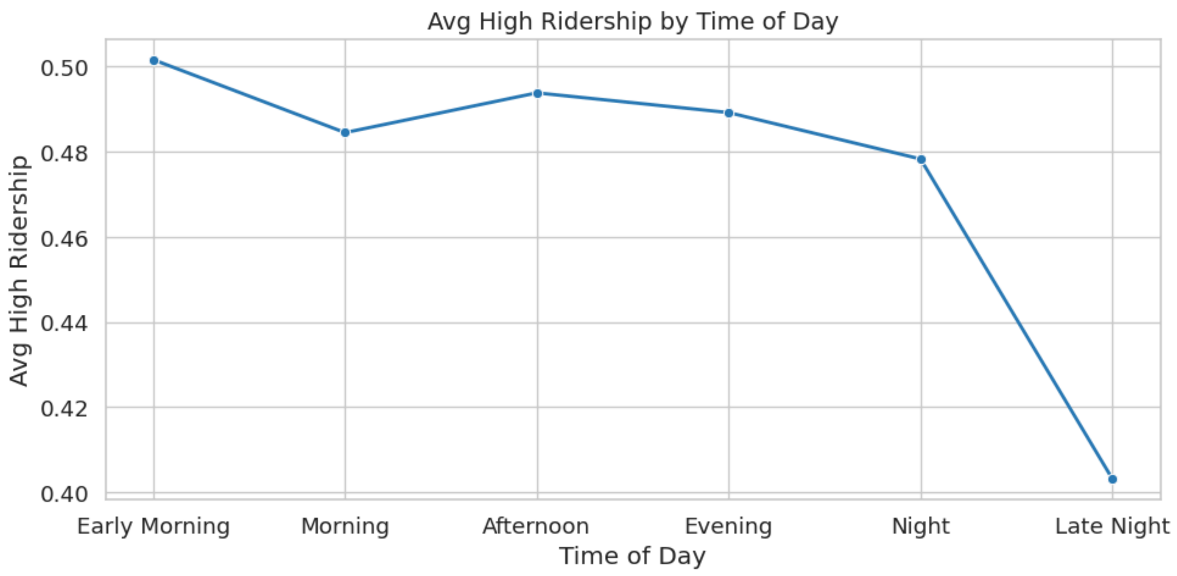
TEMPERATURE VS HIGH RIDERSHIP BY HOUR

Insight: As the day progresses, average temperature and apparent temperature steadily decline, while high ridership increases until mid-afternoon. This suggests ridership is more strongly influenced by time of day than by temperature.



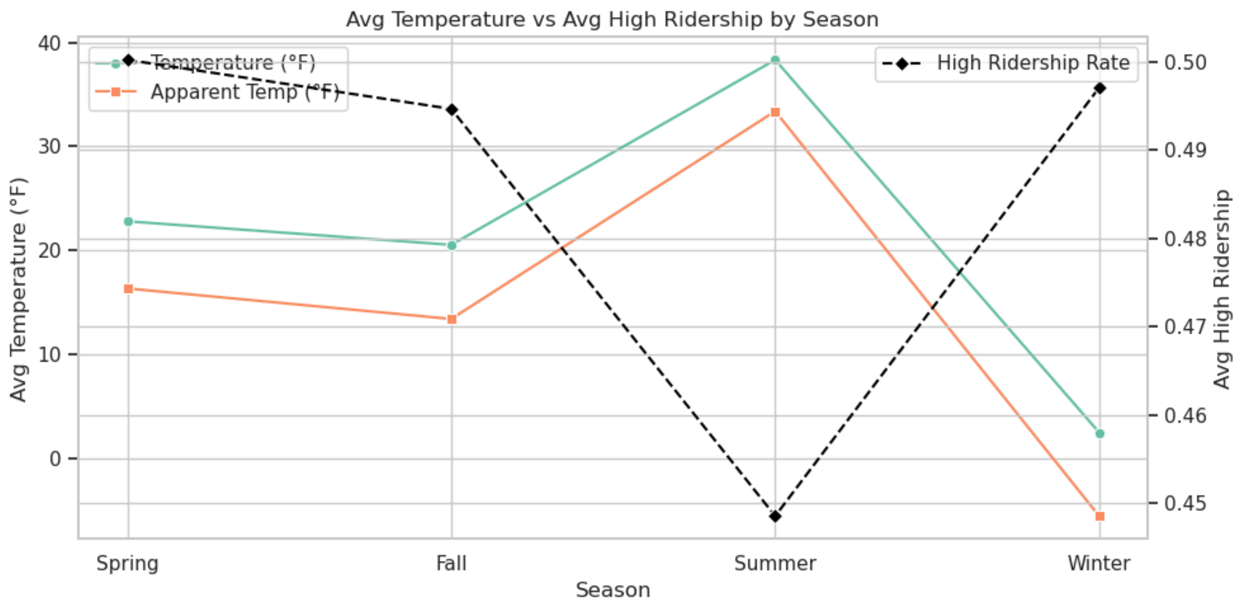
AVG HIGH RIDERSHIP BY TIME OF DAY

Insight: Early morning and afternoon time slots show the highest average high ridership, with a gradual decline into the night and the lowest levels during late night hours. This indicates commuters dominate ridership patterns, with reduced demand during nighttime periods.



AVG TEMPERATURE VS AVG HIGH RIDERSHIP BY SEASON

Insight: Despite summer showing the highest average temperature, it corresponds to the lowest high ridership rate. Conversely, winter shows the lowest temperature but among the highest ridership rates. This further confirms that high ridership is not driven by warmer weather, and may instead be shaped by commuting routines and seasonal events.

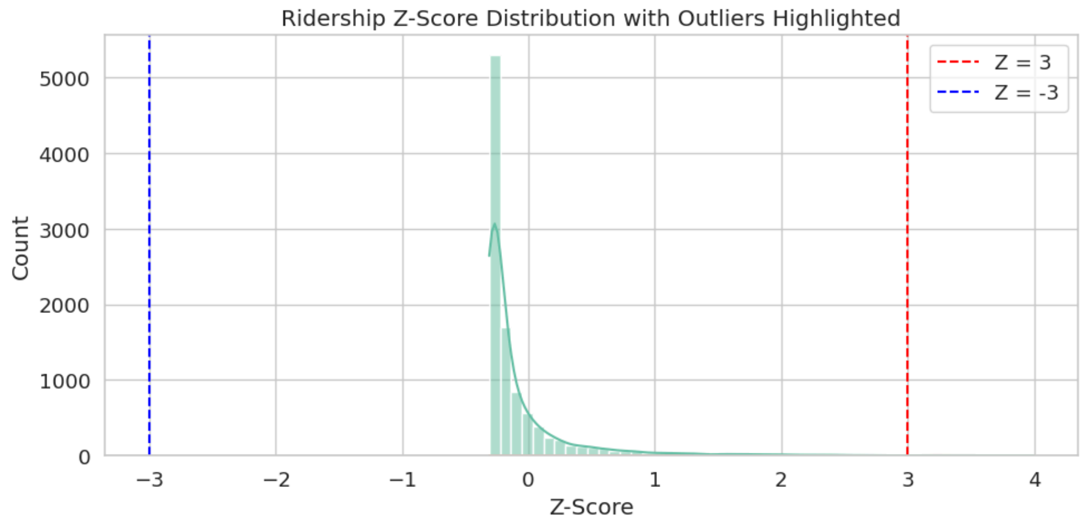


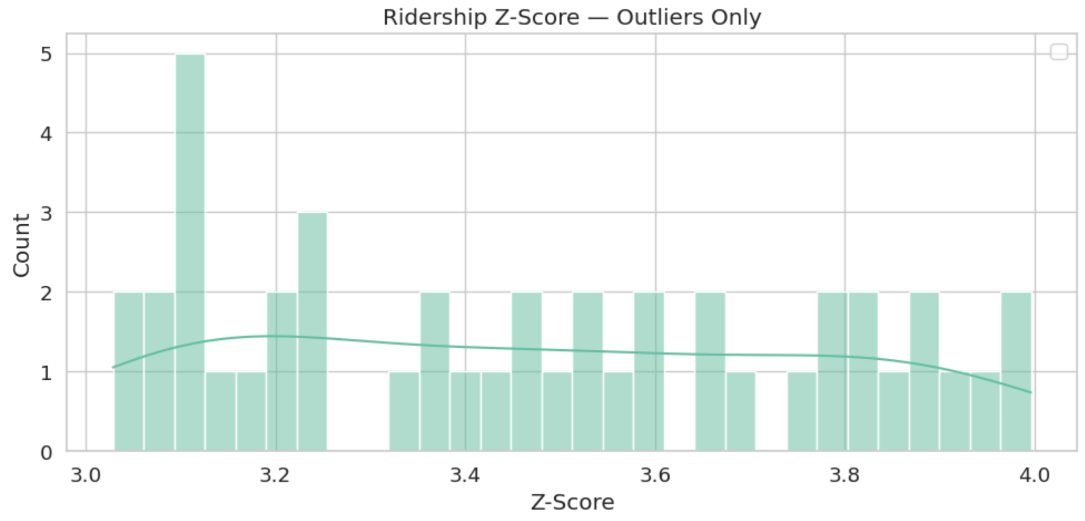
ANALYSIS OF OUTLIERS

VISUALISING THE OUTLIERS

Insight: The below chart shows that most ridership Z-scores are clustered around 0, with the distribution skewed to the right. This skewness is expected, as ridership can't be negative—data is only recorded when ridership occurs, and no entry exists when it doesn't. Outliers beyond ±3 are minimal, and the bottom chart zooms in on high-end outliers (Z > 3), revealing just a few extreme cases.

While creating the Z-score column, most outliers were already addressed by removing rows where Z-scores exceeded ±4. As a result, we observe only limited outliers in this sample. Additionally, since this analysis is based on a sampled dataframe, it's possible that rows with extreme Z-scores were underrepresented in the sample.

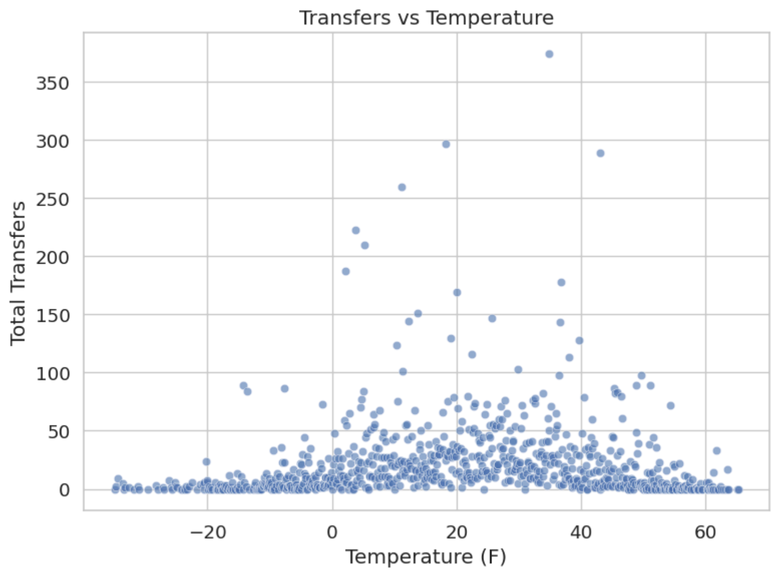




ANALYSIS OF TRANSFERS

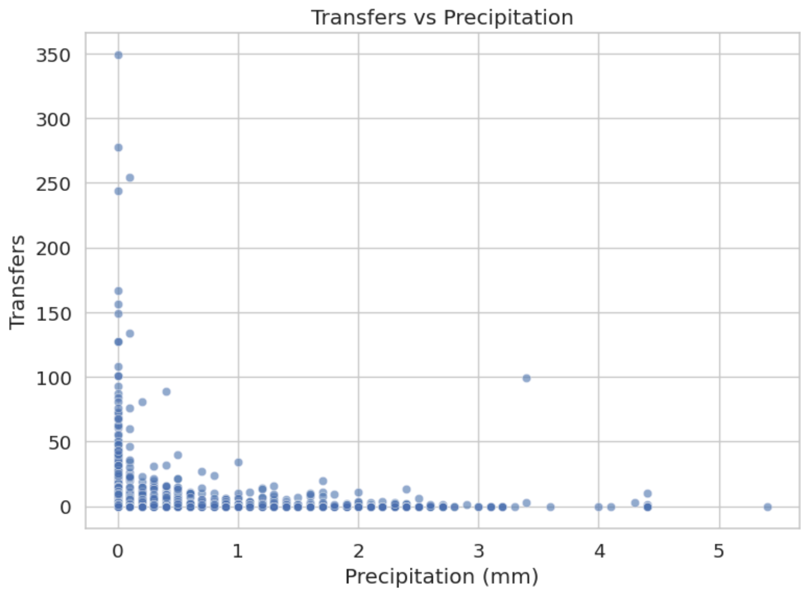
TRANSFERS VS TEMPERATURE

Insight: Transfers remain relatively stable across temperature ranges, but a few outliers show high transfer counts even during extreme cold or moderate heat. This indicates that temperature alone does not have a strong linear influence on transfer volume.



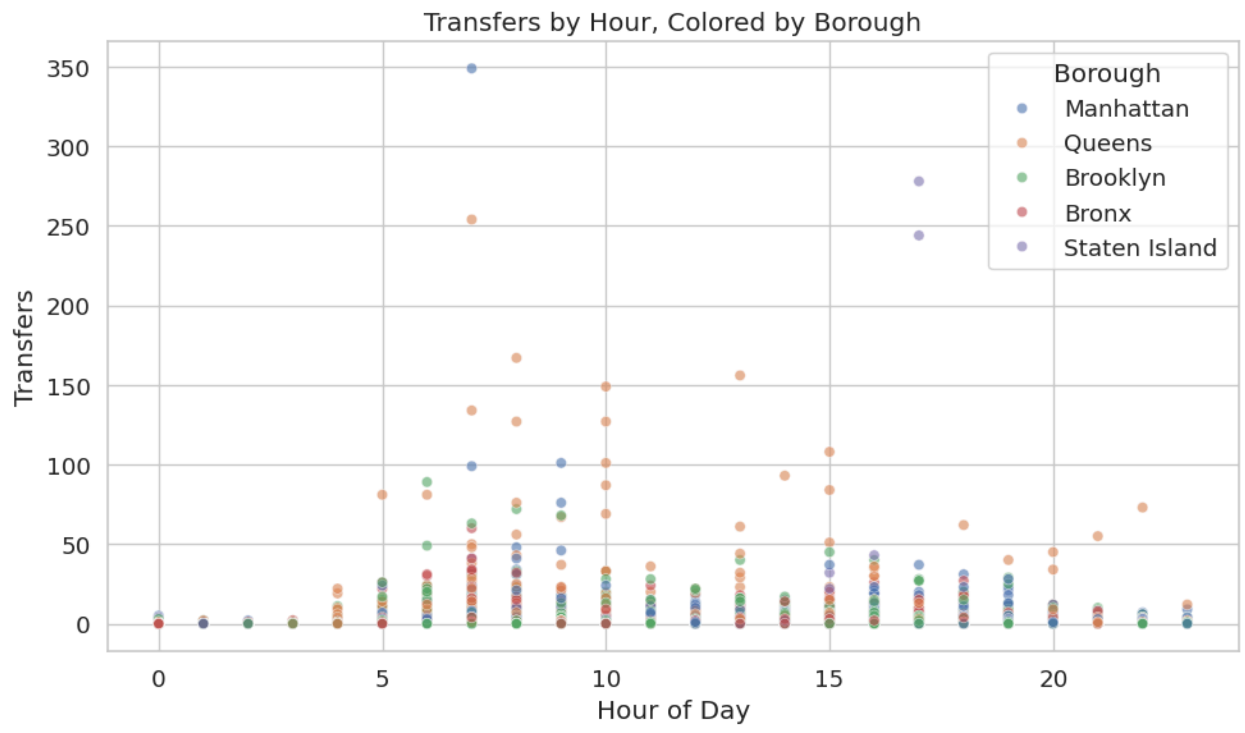
TRANSFERS VS PRECIPITATION

Insight: There is a visible drop in transfer counts as precipitation increases. Most high transfer counts occur when precipitation is low or zero, suggesting that rainy conditions may discourage multiple transfers or reduce trip chaining behavior.



TRANSFERS BY HOUR (COLOURED BY BOROUGH)

Insight: Transfer activity is highest during morning hours (around 6–10 AM), with Queens and Manhattan showing particularly high transfer spikes. Staten Island has fewer data points overall.



FEATURE IMPORTANCE

TOP 10 MOST IMPORTANT FEATURES

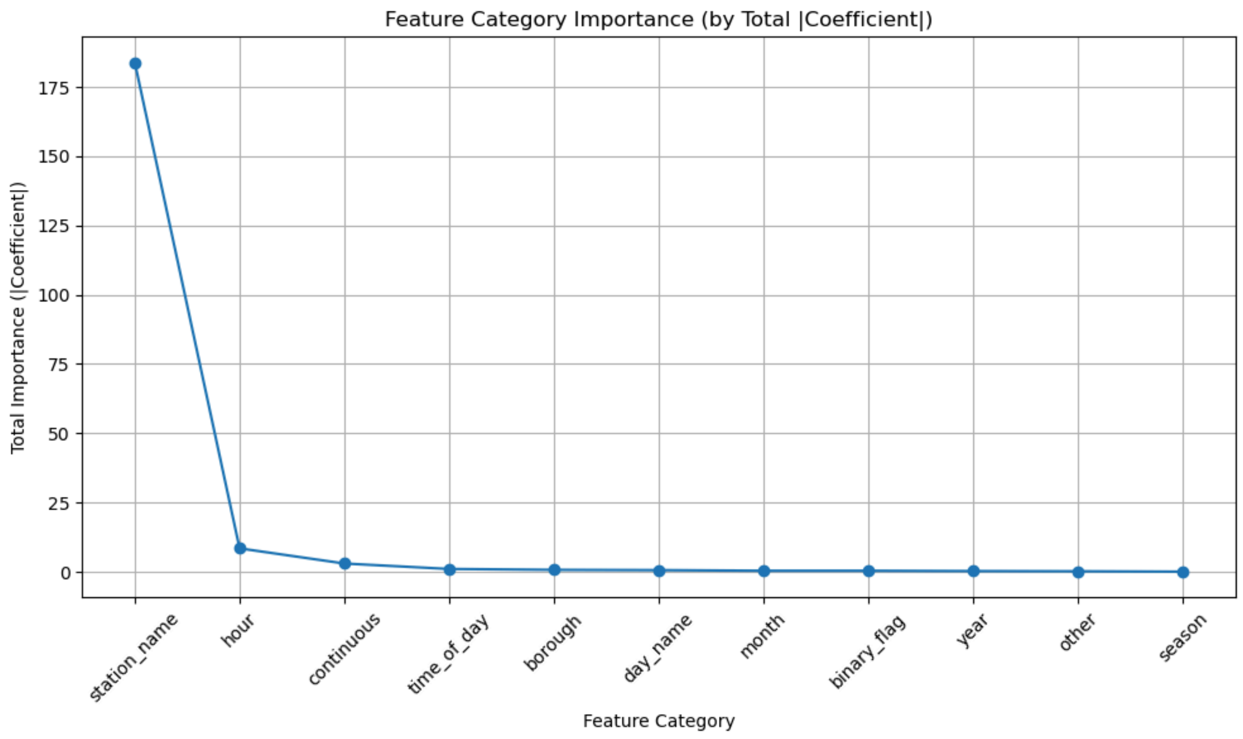
Findings:

```
Top 10 Most Important Features:

                       Feature  Coefficient  Importance
391           station_name_384    -3.591453    3.591453
28             station_name_21    -3.044640    3.044640
502   ridership_z_score_scaled     2.986795    2.986795
26             station_name_19    -2.263709    2.263709
96             station_name_89    -2.251639    2.251639
334           station_name_327    -2.216990    2.216990
452                    hour_18    -1.713221    1.713221
203           station_name_196    -1.654092    1.654092
128           station_name_121    -1.607062    1.607062
85             station_name_78    -1.489243    1.489243
```

TOTAL FEATURE IMPORTANCE BY CATEGORY

Insight: The feature importance analysis shows that `station_name` is by far the most influential variable in predicting high ridership, followed by `hour` and `continuous variables` (likely numeric weather or transfer features). Other temporal and categorical variables like `time_of_day`, `borough`, and `day_name` contribute marginally. Seasonal, yearly, and binary flags have minimal influence. This suggests that ridership patterns are highly localized and time-dependent, with specific stations and hours driving the most predictive power.

```
Total Feature Importance by Category:

        Category  Importance
8   station_name  183.923939
4           hour    8.472000
2     continuous    2.998718
9    time_of_day    1.035972
1        borough    0.724339
3       day_name    0.620601
5          month    0.394640
0    binary_flag    0.390658
10          year    0.276038
6          other    0.202222
7         season    0.065093
```



BUSIEST STATIONS GROUPED BY CATEGORIES

I calculated the average of the high_ridership column, grouping the data by `station_name` along with other features such as `hour`, `time_of_day`, `season`, and `month`. Based on this, I identified the top 10 stations with consistently high ridership. While this analysis is based on the sample dataset, it provides useful insights that can help inform broader inferences.

GROUPED BY HOUR

```
+----+--------------------------------------+-------------------+
|hour|station_name                          |avg_high_ridership|
+----+--------------------------------------+-------------------+
|0   |Myrtle Av (M,J,Z)                     |1.0               |
|1   |Avenue I (F)                          |1.0               |
|2   |137 St-City College (1)               |1.0               |
|3   |Jamaica Center-Parsons/Archer (E,J,Z)|1.0               |
|4   |Van Siclen Av (J,Z)                   |1.0               |
|5   |DeKalb Av (L)                         |1.0               |
|6   |96 St (1,2,3)                         |1.0               |
|7   |14 St-Union Sq (L,N,Q,R,W,4,5,6)      |1.0               |
|8   |20 Av (N)                             |1.0               |
|9   |57 St (F)                             |1.0               |
|10  |Wakefield-241 St (2)                  |1.0               |
+----+--------------------------------------+-------------------+
```

GROUPED BY TIME OF DAY

```
+--------------+-----------------------+-------------------+
|time_of_day   |station_name           |avg_high_ridership|
+--------------+-----------------------+-------------------+
|Afternoon     |104 St (A)             |1.0               |
|Early Morning|Lafayette Av (C)       |1.0               |
|Evening       |86 St (4,5,6)          |1.0               |
|Late Night    |Aqueduct Racetrack (A)|1.0               |
|Morning       |Wilson Av (L)          |1.0               |
|Night         |46 St (M,R)            |1.0               |
+--------------+-----------------------+-------------------+
```

GROUPED BY TIME OF SEASON

```
+-------+------------------------+-------------------+
|season|station_name             |avg_high_ridership|
+-------+------------------------+-------------------+
|Fall   |Junction Blvd (7)        |1.0               |
|Spring|116 St (2,3)             |1.0               |
|Summer|Queensboro Plaza (7,N,W)|1.0               |
|Winter|69 St (7)                |1.0               |
+-------+------------------------+-------------------+
```

GROUPED BY MONTH

```
+-----+------------------------------------+-------------------+
|month|station_name                        |avg_high_ridership|
+-----+------------------------------------+-------------------+
|Apr  |Utica Av (A,C)                      |1.0               |
|Aug  |Sutter Av-Rutland Rd (3)            |1.0               |
|Dec  |Beach 98 St (A,S)                   |1.0               |
|Feb  |Grand Ave-Newtown (M,R)             |1.0               |
|Jan  |20 Av (N)                           |1.0               |
|Jul  |77 St (6)                           |1.0               |
|Jun  |Utica Av (A,C)                      |1.0               |
|Mar  |Ralph Av (C)                        |1.0               |
|May  |Lorimer St (L)/Metropolitan Av (G)  |1.0               |
|Nov  |145 St (1)                          |1.0               |
|Oct  |Coney Island-Stillwell Av (D,F,N,Q)|1.0               |
|Sep  |Mt Eden Av (4)                      |1.0               |
+-----+------------------------------------+-------------------+
```

Milestone 6: Summary & Conclusions

SUMMARY

This project was aimed at predicting high subway ridership in New York City by combining detailed MTA hourly ridership data with historical weather and holiday information. Using PySpark and Google Cloud tools, I built a scalable data pipeline that processed large MTA data. Through feature engineering—such as identifying peak hours, holidays, and weather patterns—I trained a logistic regression model that performed well, achieving an AUC of 0.90. All key steps, from data cleaning to model training and evaluation, were documented and saved to the cloud for transparency and future use.

CONCLUSIONS

One of the most important takeaways was that station-level and time-based features (like the hour of the day or the station name) were far more predictive of high ridership than weather-related factors. As expected, ridership was consistently higher during weekday peak hours, especially in central boroughs like Manhattan. While weather had a minimal direct effect, including it added context and completeness to the analysis. These insights can help transit planners better anticipate demand, optimize service levels, and improve commuter experience.

ACKNOWLEDGEMENT

I'm incredibly grateful to **Professor Richard Holowczak**, who was the biggest motivator behind this project. His constant encouragement made even the most overwhelming parts of this assignment feel doable. Whether it was helping debug issues, answering questions at odd hours, or simply reminding us that we could figure it out—his support never wavered. This project truly wouldn't have come together without his guidance, and I'm thankful to have had a professor who believes in his students the way he does.

REFERENCES

- Holowczak, R. (2025). *Course-provided starter code and lab notebooks for CIS9760: Big Data Technologies. Baruch College, CUNY.*
- Javaly, V. (2024). *Code and visualization techniques adapted from CIS9660: Data Mining for Business Analytics. Baruch College, CUNY, Fall 2024 semester.*
- OpenAI. (2025). *ChatGPT-generated code snippets and troubleshooting support via GPT-4. Accessed periodically throughout the project.*
- Python Holidays Package. (n.d.). holidays: *Generate and work with holidays in Python.* https://pypi.org/project/holidays/
- Apache Spark MLlib. (n.d.). *Machine Learning Library (MLlib) Guide.* https://spark.apache.org/docs/latest/ml-guide.html
- Pandas Documentation. (n.d.). https://pandas.pydata.org/
- Seaborn Documentation. (n.d.). https://seaborn.pydata.org/

Appendix

Appendix A : Code and Commands Used for Milestone 2

1. **GCS Commands**
    a. Creating a new bucket
    ```
    gcloud storage buckets create my-bigdata-project-yb
    ```
    b. Checking if a Bucket Exists
    ```
    gcloud storage ls -1 gs://my-bigdata-project-yb
    ```
    c. Copying Data to a Bucket
    ```
    gcloud storage cp MTA_Subway_Hourly_Ridership_2020-2024_20250302.csv
    gs://my-bigdata-project-yb/landing
    ```

2. **MTA Ridership Data Copy (Professor's Code)**
    a. Accessing full data from his server
    ```
    gs://my_project_bucket_mta_subway/landing/MTA_Subway_Hourly_Ridership__2
    20-2024_20250302.csv.gz
    ```
    b. Copying file to the local VM
    ```
    gcloud storage cp
    gs://my_project_bucket_mta_subway/landing/MTA_Subway_Hourly_Ridership__20
    20-2024_20250302.csv.gz
    MTA_Subway_Hourly_Ridership__2020-2024_20250302.csv.gz
    ```

3. **Weather Data Fetching (Python Script)**
    a. Importing required libraries
    ```
    import requests
    import pandas as pd
    from datetime import datetime
    ```
    b. Defining latitude and longitude for New York City
    ```
    LATITUDE = 40.710335
    LONGITUDE = -73.99309
    ```
    c. Initialising the API endpoint
    ```
    API_URL = "https://api.open-meteo.com/v1/forecast"
    ```
    d. Fetching weather data from Open-Meteo (next page)

```
response  = requests.get(API_URL, parmas={
       “latitude”: LATITUDE,
       “Longitude”: LONGITUDE,
       “Current_weather”: "true",
       "timezone": "America/New_York"
})

if response.status_code == 200:
    data = response.json()
    current_weather = data["current_weather"]

    weather_data = {
        "latitude": LATITUDE,
        "longitude": LONGITUDE,
        "elevation": 51,
        "utc_offset_seconds": -18000,
        "timezone": "America/New_York",
        "timezone_abbreviation": "GMT-5",
        "time": datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
        "temperature_2m": current_weather.get("temperature", "N/A"),
        "apparent_temperature": current_weather.get("apparent_temperature",
"N/A"),
        "precipitation": current_weather.get("precipitation", "N/A"),
        "wind_speed_10m": current_weather.get("windspeed", "N/A"),
        "wind_direction_10m": current_weather.get("winddirection", "N/A")
    }

    df = pd.DataFrame([weather_data])

    df.to_csv("weather_data.csv", index=False)
    print("Weather data saved to weather_data.csv")

else:
    print(f"Failed to fetch data. HTTP Status Code: {response.status_code}")
```

Appendix B : Source Code for EDA Milestone 3
Dataset: MTA Subway Ridership Data

4. Importing Storage Modules

```
from google.cloud import storage
from io import StringIO, BytesIO
import pandas as pd
import gzip
pd.set_option('display.float_format', '{:.2f}'.format)
pd.set_option('display.width', 1000)
```

5. Defining read_csv_in_chunks_from_gcs()

```
def read_csv_in_chunks_from_gcs(bucket_name, file_path, chunk_size=1000):
    """
    Reads a CSV file from Google Cloud Storage in chunks.
    Args:
        bucket_name (str): The name of the GCS bucket.
        file_path (str): The path to the CSV file in GCS.
        chunk_size (int): The number of rows to read in each chunk.
    Yields:
        pd.DataFrame: A chunk of the CSV file as a pandas DataFrame.
    """
    # Set up the Google CLoud Storage client connection
    storage_client = storage.Client()
    # Reference the bucket
    bucket = storage_client.bucket(bucket_name)
    # Get the blob for the file path
    blob = bucket.blob(file_path)
    # Check to see if the file is compressed or not
    if file_path.endswith(".gz"):
        compression = 'gzip'
    else:
        compression = None

    # Open the blob as a file stream 'f'
    # with blob.open("r", encoding='utf-8') as f:
    with blob.open("r") as f:
        # Create a 'reader' that reads in one chunk at a time
        # reader = pd.read_csv(f, chunksize=chunk_size, encoding='utf-8',
low_memory=False, compression=compression)
        reader = pd.read_csv(f, chunksize=chunk_size, low_memory=False,
compression=compression)
        # Loop through the chunks and return each one
        for chunk in reader:
            yield chunk
```

6. **Defining perform_EDA()**

```
def perform_EDA(df : pd.DataFrame, filename : str):
    """
    perform_EDA(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on the data and outputs to the console.

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :returns:
    """
    print(f"{filename} Number of records:")
    print(df.count())
    number_of_duplicate_records = df.duplicated().sum()
    # old way number_of_duplicate_records = len(df)-len(df.drop_duplicates())
    print(f"{filename} Number of duplicate records:
{number_of_duplicate_records}" )
    print(f"{filename} Info")
    print(df.info())
    print(f"{filename} Describe")
    print(df.describe())
    print(f"{filename} Columns with null values")
    print(df.columns[df.isnull().any()].tolist())
    rows_with_null_values = df.isnull().any(axis=1).sum()
    print(f"{filename} Number of Rows with null values:
{rows_with_null_values}" )
    integer_column_list = df.select_dtypes(include='int64').columns
    print(f"{filename} Integer data type columns: {integer_column_list}")
    float_column_list = df.select_dtypes(include='float64').columns
    print(f"{filename} Float data type columns: {float_column_list}")
```

7. **Defining perform_EDA_numeric()**

```
def perform_EDA_numeric(df : pd.DataFrame, filename : str):
    """
    perform_EDA_numeric(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on numeric columns and saves the output in a
dataframe.

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :returns:
    :   pd.DataFrame: A new dataframe with summary statistics
    """
    # Initialize a list to collect summary data
    summary_data = []
    # Gather summary statistics on numeric columns
    for col in df.select_dtypes(include=['int64', 'float64']).columns:
       summary_data.append({
            'Filename': filename,         'Column': col,
            'Minimum': df[col].min(),     'Maximum': df[col].max(),
            'Average': df[col].mean(),    'Standard Deviation': df[col].std(),
            'Missing Values': df[col].isnull().sum()
        })
    # Convert the summary data list into a DataFrame
    return pd.DataFrame(summary_data)
```

8. **Defining perform_EDA_categorical()**

```
def perform_EDA_categorical(df : pd.DataFrame, filename : str,
categorical_columns):
    """
    perform_EDA_categorical(df : pd.DataFrame, filename : str,
categorical_columns)
    Accepts a dataframe and a text filename as inputs.
    Collects statistics on Categorical columns

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :param categorical_columns: A list of column names for categorical columns
    :returns:
    :   pd.DataFrame: A new dataframe with summary statistics
    """
    # Initialize a list to collect summary data
    summary_data = []
    # Gather summary statistics on numeric columns
    for col in categorical_columns:
       summary_data.append({
            'Filename': filename,
            'Column': col,
            'Unique Values': df[col].apply(lambda x: tuple(x) if isinstance(x,
list) else x).nunique(),
            'Minimum': df[col].min(),
            'Maximum': df[col].max(),
            'Missing Values': df[col].isnull().sum()
        })
    # Convert the summary data list into a DataFrame
    return pd.DataFrame(summary_data)
```

9. **Defining main_subway()**

```
def main_subway():
    # Columns are 'transit_timestamp', 'transit_mode', 'station_complex_id',
'station_complex', 'borough', 'payment_method', 'fare_class_category',
'ridership', 'transfers', 'latitude', 'longitude', 'Georeference'
    categorical_columns_list = [ 'transit_timestamp', 'transit_mode',
'station_complex_id', 'station_complex', 'borough', 'payment_method',
'fare_class_category']

    # Change the bucket name to match your project bucket
    bucket_name = 'my-bigdata-project-yb'
    file_path = "landing/MTA_Subway_Hourly_Ridership__2020-2024_20250302.csv"

    # Set the chunk_size to a high number such as 10 million or 100 million
    # If the Python Kernel 'restarts' or crashes, reduce the chunk_size
    chunk_size = 10000000
    chunk_counter = 0
    for chunk_df in read_csv_in_chunks_from_gcs(bucket_name, file_path,
chunk_size=chunk_size):
        # Process each chunk of the DataFrame
        chunk_counter = chunk_counter + chunk_size
        print(f"EDA on chunk from {chunk_counter-chunk_size} to
{chunk_counter}")
        chunk_df.info()
        print(chunk_df.head())
        print(chunk_df.shape)
        perform_EDA(chunk_df, file_path)
        categorical_summary_df =
perform_EDA_categorical(chunk_df,file_path,categorical_columns_list)
```

10. **Calling the main_subway() function**

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

11. **Additional EDA**

```
# displaying number of rows
print(f"Total Records: {sdf.count()}")

# listing all columns
print(f"Columns: {sdf.columns}")

# checking for null values
from pyspark.sql.functions import col, sum
sdf.select([(sum(col(c).isNull().cast("int")).alias(c)) for c in
sdf.columns]).show()

# summary  statistics for int columns {ridership, transfers}
sdf.select("ridership","transfers").summary("count", "min", "max").show()
```

```
# printing unique values in ‘transit_mode’ column
distinct_transit_modes = sdf.select("transit_mode").distinct()
print("Distinct Transit Modes:")
distinct_transit_modes.show(truncate=False)

# min/max dates for date column
sdf.selectExpr("min(transit_timestamp) as MinDate", "max(transit_timestamp) as
MaxDate").show()

# top 10 busiest stations
sdf.groupBy("station_complex").sum("ridership").orderBy(F.desc("sum(ridership)"
)).show(10)

# borough-wise ridership sum
sdf.groupBy("borough").sum("ridership").show()

# ridership by fare class category
sdf.groupBy("fare_class_category").sum("ridership").show()

# stations with highest transfers
sdf.groupBy("station_complex").sum("transfers").orderBy(F.desc("sum(transfers)"
)).show(10)
```

12. Defining small_subway()

```
def small_subway():
    bucket_name = 'my-bigdata-project-yb'
    file_path = "landing/MTA_Subway_Hourly_Ridership__2020-2024_20250302.csv"
    chunk_size = 100000  # Reduced to avoid memory issues
    chunk_counter = 0
    samples = []

    try:
        for chunk_df in read_csv_in_chunks_from_gcs(bucket_name, file_path,
chunk_size=chunk_size):
            chunk_counter += chunk_size
            print(f"Processing chunk from {chunk_counter - chunk_size} to
{chunk_counter}")

            if chunk_df is None or chunk_df.empty:
                print("Chunk is empty or could not be read.")
                continue

            # Take a random sample of 1000 rows from each chunk
            samples.append(chunk_df.sample(min(1000, len(chunk_df))))  # Ensure
you don't sample more than the chunk size
```

```
            # Break early to avoid memory overload, comment if you want to
process more chunks
            if len(samples) >= 10:  # Process 10 samples of 1000 rows each
(Total: 10,000 rows)
                break

        # Combine all the samples into one DataFrame
        sampled_df = pd.concat(samples, ignore_index=True)
        return sampled_df  # Return the smaller DataFrame for analysis

    except Exception as e:
        print(f"An error occurred: {str(e)}")
        return None
```

13. Assigning it to small_chunk_df

```
small_chunk_df  = small_subway()
```

14. Using small_chunk_df to perform visualisations

```
# Setting the style
sns.set(style="whitegrid")

# 1. Distribution Analysis (Histogram & Boxplot)
def plot_numerical_distributions(df, column):
    plt.figure(figsize=(14, 6))

    # Histogram
    plt.subplot(1, 2, 1)
    sns.histplot(df[column].dropna(), bins=30, kde=True)
    plt.title(f'Distribution of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')

    # Boxplot
    plt.subplot(1, 2, 2)
    sns.boxplot(x=df[column].dropna())
    plt.title(f'Boxplot of {column}')

    plt.show()

plot_ridership_over_time(small_chunk_df)

# 2. Ridership Analysis Over Time (Line Plot)
def plot_ridership_over_time(df):
    plt.figure(figsize=(12, 6))
    df['transit_timestamp'] = pd.to_datetime(df['transit_timestamp'])
    time_series_df =
df.groupby('transit_timestamp')['ridership'].sum().reset_index()
```

```
        sns.lineplot(x='transit_timestamp', y='ridership', data=time_series_df)
        plt.title('Ridership Over Time')
        plt.xlabel('Date')
        plt.ylabel('Total Ridership')
        plt.show()

    plot_ridership_over_time(small_chunk_df)

    # 3. Geographical Visualization (Scatter Plot)
    def plot_geographical_distribution(df):
        plt.figure(figsize=(10, 8))
        sns.scatterplot(x='longitude', y='latitude', data=df, hue='borough',
palette='Set2')
        plt.title('Geographical Distribution of Stations')
        plt.xlabel('Longitude')
        plt.ylabel('Latitude')
        plt.legend(title='Borough')
        plt.show()
    plot_geographical_distribution(small_chunk_df)

    # 4. Bar graphs for categorical and numerical columns
    def plot_categorical_distribution(df, column):
        plt.figure(figsize=(10, 6))
        sns.countplot(y=column, data=df, order=df[column].value_counts().index)
        plt.title(f'Distribution of {column}')
        plt.xlabel('Frequency')
        plt.ylabel(column)
        plt.show()

    # Function to plot bar graphs for numerical columns
    def plot_numerical_distribution(df, column):
        plt.figure(figsize=(10, 6))
        sns.histplot(df[column].dropna(), bins=30, kde=False)
        plt.title(f'Distribution of {column}')
        plt.xlabel(column)
        plt.ylabel('Frequency')
        plt.show()
    plot_categorical_distribution(small_chunk_df, 'payment_method')
    plot_categorical_distribution(small_chunk_df, 'fare_class_category')
    plot_numerical_distribution(small_chunk_df, 'ridership')
    # 5. Ridership analysis by Borough
    plt.figure(figsize=(10,5))
    sns.barplot(data=small_chunk_df, x="borough", y="ridership", estimator=sum,
ci=None, palette="viridis")
    plt.title("Total Ridership by Borough")
    plt.xlabel("Borough")
    plt.ylabel("Total Ridership")
    plt.xticks(rotation=45)
    plt.show()
```

Appendix B : Source Code for EDA Milestone 3
Dataset: Weather API

15. Defining read_csv_in_chunks_from_gcs()

```
def read_csv_in_chunks_from_gcs(bucket_name, file_path, chunk_size=1000):
    """
    Reads a CSV file from Google Cloud Storage in chunks.

    Args:
        bucket_name (str): The name of the GCS bucket.
        file_path (str): The path to the CSV file in GCS.
        chunk_size (int): The number of rows to read in each chunk.

    Yields:
        pd.DataFrame: A chunk of the CSV file as a pandas DataFrame.
    """
    # Set up the Google CLoud Storage client connection
    storage_client = storage.Client()
    # Reference the bucket
    bucket = storage_client.bucket(bucket_name)
    # Get the blob for the file path
    blob = bucket.blob(file_path)
    # Check to see if the file is compressed or not
    if file_path.endswith(".gz"):
        compression = 'gzip'
    else:
        compression = None

    # Open the blob as a file stream 'f'
    # with blob.open("r", encoding='utf-8') as f:
    with blob.open("r") as f:
        # Create a 'reader' that reads in one chunk at a time
        # reader = pd.read_csv(f, chunksize=chunk_size, encoding='utf-8',
low_memory=False, compression=compression)
        reader = pd.read_csv(f, chunksize=chunk_size, low_memory=False,
compression=compression)
        # Loop through the chunks and return each one
        for chunk in reader:
            yield chunk
```

16. Defining perform_EDA()

```
def perform_EDA(df : pd.DataFrame, filename : str):
    """
    perform_EDA(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on the data and outputs to console.
    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :returns:
    """
    print(f"{filename} Number of records:")
    print(df.count())
    number_of_duplicate_records = df.duplicated().sum()
    # old way number_of_duplicate_records = len(df)-len(df.drop_duplicates())
    print(f"{filename} Number of duplicate records:
{number_of_duplicate_records}" )
    print(f"{filename} Info")
    print(df.info())
    print(f"{filename} Describe")
    print(df.describe())
    print(f"{filename} Columns with null values")
    print(df.columns[df.isnull().any()].tolist())
    rows_with_null_values = df.isnull().any(axis=1).sum()
    print(f"{filename} Number of Rows with null values:
{rows_with_null_values}" )
    integer_column_list = df.select_dtypes(include='int64').columns
    print(f"{filename} Integer data type columns: {integer_column_list}")
    float_column_list = df.select_dtypes(include='float64').columns
    print(f"{filename} Float data type columns: {float_column_list}")
```

17. Defining perform_EDA_numeric()

```
def perform_EDA_numeric(df : pd.DataFrame, filename : str):
    """
    perform_EDA_numeric(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on numeric columns and saves the output in a
dataframe.

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :returns:
    :   pd.DataFrame: A new dataframe with summary statistics
    """
    # Initialize a list to collect summary data
    summary_data = []
    # Gather summary statistics on numeric columns
    for col in df.select_dtypes(include=['int64', 'float64']).columns:
       summary_data.append({
            'Filename': filename,         'Column': col,
            'Minimum': df[col].min(),     'Maximum': df[col].max(),
            'Average': df[col].mean(),    'Standard Deviation': df[col].std(),
            'Missing Values': df[col].isnull().sum()
```

```
        })
    # Convert the summary data list into a DataFrame
    return pd.DataFrame(summary_data)
```

18. Defining perform_EDA_categorical()

```
def perform_EDA_categorical(df : pd.DataFrame, filename : str,
categorical_columns):
    """
    perform_EDA_categorical(df : pd.DataFrame, filename : str,
categorical_columns)
    Accepts a dataframe and a text filename as inputs.
    Collects statistics on Categorical columns

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :param categorical_columns: A list of column names for categorical columns
    :returns:
    :   pd.DataFrame: A new dataframe with summary statistics
    """
    # Initialize a list to collect summary data
    summary_data = []
    # Gather summary statistics on numeric columns
    for col in categorical_columns:
       summary_data.append({
            'Filename': filename,
            'Column': col,
            'Unique Values': df[col].apply(lambda x: tuple(x) if isinstance(x,
list) else x).nunique(),
            'Minimum': df[col].min(),
            'Maximum': df[col].max(),
            'Missing Values': df[col].isnull().sum()
        })
    # Convert the summary data list into a DataFrame
    return pd.DataFrame(summary_data)
```

19. Defining main_weather()

```
def main_weather():
    # Columns we are interested in
    relevant_columns = ['apparent_temperature', 'precipitation', 'rain',
'showers', 'snow_fall', 'snow_depth', 'weather_code']

    bucket_name = 'weather-api-mta-yb'
    file_path = "landing/weather_data.csv"

    # Set the chunk_size
    chunk_size = 1000000  # Reduced to avoid crashing
    try:
        for chunk_df in read_csv_in_chunks_from_gcs(bucket_name, file_path,
chunk_size=chunk_size):
            if chunk_df is None or chunk_df.empty:
                print("Chunk is empty or could not be read.")
                continue

            print("Chunk loaded successfully!")

            available_columns = [col for col in relevant_columns if col in
chunk_df.columns]
            if not available_columns:
                print("No relevant columns found in this chunk.")
                continue

            chunk_df = chunk_df[available_columns]

            # Display the relevant columns
            print(chunk_df.head())
            print(chunk_df.info())
            print(f"Chunk shape: {chunk_df.shape}")

            return chunk_df

    except Exception as e:
        print(f"An error occurred: {str(e)}")
        return None

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

Appendix C : The Data Cleaning Code Milestone 3

20. Defining read_csv_in_chunks_from_gcs() for Weather API data

```
def read_csv_in_chunks_from_gcs(bucket_name, file_path, chunk_size=1000):
    """
    Reads a CSV file from Google Cloud Storage in chunks.

    Args:
        bucket_name (str): The name of the GCS bucket.
        file_path (str): The path to the CSV file in GCS.
        chunk_size (int): The number of rows to read in each chunk.

    Yields:
        pd.DataFrame: A chunk of the CSV file as a pandas DataFrame.
    """
    # Set up the Google CLoud Storage client connection
    storage_client = storage.Client()
    # Reference the bucket
    bucket = storage_client.bucket(bucket_name)
    # Get the blob for the file path
    blob = bucket.blob(file_path)
    # Check to see if the file is compressed or not
    if file_path.endswith(".gz"):
        compression = 'gzip'
    else:
        compression = None

    # Open the blob as a file stream 'f'
    # with blob.open("r", encoding='utf-8') as f:
    with blob.open("r") as f:
        # Create a 'reader' that reads in one chunk at a time
        # reader = pd.read_csv(f, chunksize=chunk_size, encoding='utf-8',
low_memory=False, compression=compression)
        reader = pd.read_csv(f, chunksize=chunk_size, low_memory=False,
compression=compression)
        # Loop through the chunks and return each one
        for chunk in reader:
            yield chunk
```

21. Defining clean_data()

```
def clean_data(chunk_df):
    # Drop unnecessary columns if they exist
    unnecessary_columns = ['Georeference']  # Add other unnecessary columns if
present
    chunk_df = chunk_df.drop(columns=unnecessary_columns, errors='ignore')
    # Remove rows where critical columns are missing (if needed)
    required_columns = ['apparent_temperature', 'precipitation', 'rain',
                        'showers', 'snowfall', 'snow_depth', 'weather_code']

    # Drop rows where all required columns are missing
    chunk_df = chunk_df.dropna(subset=required_columns, how='all')

    # Convert relevant columns to numeric, replace invalid values with NaN
    for column in required_columns:
        if column in chunk_df.columns:
            chunk_df[column] = pd.to_numeric(chunk_df[column], errors='coerce')

    # Fill remaining missing values with 0 (assuming missing data means 0 in
the weather context)
    chunk_df = chunk_df.fillna(0)

    # Rename columns to standardize format
    chunk_df.columns = chunk_df.columns.str.lower().str.replace(' ', '_')

    return chunk_df
```

22. Defining main_weather_clean()

```
def main_weather_clean():
    # GCS Bucket and File Paths
    bucket_name = 'weather-api-mta-yb'
    input_folder = "landing/"
    output_folder = "cleaned/"
    input_file_path = f"{input_folder}weather_data.csv"

    # Set chunk size to avoid memory overload
    chunk_size = 100000  # Reduced for safer processing
    chunk_counter = 0

    try:
        for chunk_df in read_csv_in_chunks_from_gcs(bucket_name,
input_file_path, chunk_size=chunk_size):
            chunk_counter += 1
            print(f"\nProcessing & Cleaning Chunk {chunk_counter}")

            # Clean the chunk
            cleaned_df = clean_data(chunk_df)

            # Upload cleaned chunk directly to GCS as Parquet
            new_filename =
f"{output_folder}cleaned_weather_chunk_{chunk_counter}.parquet"
```

```
            client = storage.Client()
            bucket = client.bucket(bucket_name)
            new_blob = bucket.blob(new_filename)

            # Save DataFrame to Parquet in-memory and upload directly
            parquet_buffer = io.BytesIO()
            cleaned_df.to_parquet(parquet_buffer, index=False)
            parquet_buffer.seek(0)
            new_blob.upload_from_file(parquet_buffer,
content_type='application/octet-stream')

            print(f"✅ Uploaded chunk {chunk_counter} to GCS as
{new_filename}")

            # Clear memory after each chunk
            del cleaned_df
            gc.collect()

    except Exception as e:
        print(f"An error occurred: {str(e)}")

main_weather_clean()
```

23. Defining clean_data() for MTA Subway Data

```
def clean_data(chunk_df):
    # Step 1: Drop unnecessary columns
    unnecessary_columns = ['Georeference']
    chunk_df = chunk_df.drop(columns=unnecessary_columns, errors='ignore')

    # Step 2: Handle missing data (Drop rows with missing values)
    chunk_df = chunk_df.dropna(subset=['transit_timestamp', 'transit_mode',
'station_complex_id',
                                       'station_complex', 'borough',
'payment_method',
                                       'fare_class_category', 'ridership',
'transfers'])

    # Step 3: Fill missing numerical data with 0
    numerical_columns = ['ridership', 'transfers']
    chunk_df[numerical_columns] = chunk_df[numerical_columns].fillna(0)

# Convert 'transit_timestamp' to datetime format with a specified format
    try:
        chunk_df['transit_timestamp'] =
pd.to_datetime(chunk_df['transit_timestamp'], format='%Y-%m-%d %H:%M:%S',
errors='coerce')
```

```
    except:
        chunk_df['transit_timestamp'] =
pd.to_datetime(chunk_df['transit_timestamp'], errors='coerce')

    # Convert 'station_complex_id' to numeric, convert errors to NaN
    chunk_df['station_complex_id'] =
pd.to_numeric(chunk_df['station_complex_id'], errors='coerce')

    # Remove rows with NaN values generated by invalid data conversions
    chunk_df = chunk_df.dropna(subset=['transit_timestamp',
'station_complex_id'])

    # Convert data types for numerical columns
    chunk_df['station_complex_id'] = chunk_df['station_complex_id'].astype(int)
    chunk_df['ridership'] = chunk_df['ridership'].astype(float)
    chunk_df['transfers'] = chunk_df['transfers'].astype(float)

    # Rename columns
    chunk_df.columns = chunk_df.columns.str.lower().str.replace(' ', '_')

    return chunk_df
```

24. Defining main_subway_clean()

```
def main_subway_clean():
    # Change the bucket name to match your project bucket
    bucket_name = 'my-bigdata-project-yb'
    input_folder = "landing/"
    output_folder = "cleaned/"
    input_file_path =
f"{input_folder}MTA_Subway_Hourly_Ridership__2020-2024_20250302.csv"

    chunk_size = 100000  # Reduced for better handling
    chunk_counter = 0

    try:
        for chunk_df in read_csv_in_chunks_from_gcs(bucket_name,
input_file_path, chunk_size=chunk_size):
            chunk_counter += 1
            print(f"Cleaning chunk {chunk_counter}")

            # Clean the data chunk
            cleaned_df = clean_data(chunk_df)

            # Save cleaned chunk to Parquet format
            new_filename =
f"{output_folder}cleaned_mta_ridership_chunk_{chunk_counter}.parquet"
            client = storage.Client()
            bucket = client.bucket(bucket_name)
            new_blob = bucket.blob(new_filename)

            # Save as Parquet file and upload directly to GCS
            cleaned_df.to_parquet("temp.parquet", index=False)
            with open("temp.parquet", "rb") as filedata:
                new_blob.upload_from_file(filedata)

            print(f"Saved cleaned data file to {new_filename}")

    except Exception as e:
        print(f"An error occurred: {str(e)}")

main _subway_clean()
```

Appendix D : Feature Engineering Code

25. Reading Data from GCS cleaned-folder

```
# Set up the path to a file
bucket = 'gs://my-bigdata-project-yb'
landing_folder = f"{bucket}/landing/"
cleaned_folder = f"{bucket}/cleaned/"
trusted_folder = f"{bucket}/trusted/"
models_folder = f"{bucket}/models/"

spark.conf.set("spark.sql.legacy.parquet.nanosAsLong", "true")
# Read all Parquet files from the folder
spark_df = spark.read.parquet(cleaned_folder)
```

26. Defining convert_unix_to_timestamp()

```
def convert_unix_to_timestamp(df, original_column="transit_timestamp",
new_column="transit_timestamp"):
    """
    Converts a Unix timestamp (in nanoseconds) column into a readable timestamp
column.

    Parameters:
    - df (DataFrame): Input Spark DataFrame.
    - original_column (str): Name of the column containing Unix timestamp.
    - new_column (str): Name to assign to the new readable timestamp column.

    Returns:
    - DataFrame: Updated Spark DataFrame with the converted timestamp.
    """
    df = df.withColumn("timestamp_converted",
from_unixtime((col(original_column) / 1_000_000_000).cast("long")))
    df = df.drop(original_column).withColumnRenamed("timestamp_converted",
new_column)
    return df

spark_df = convert_unix_to_timestamp(spark_df)
```

27. Defining sample_dataframe()

```
def sample_dataframe(spark_df, fraction=0.20, seed=42):
    """
    Randomly samples a fraction of the DataFrame without replacement.

    Parameters:
    - df (DataFrame): Input Spark DataFrame.
    - fraction (float): Fraction of the data to sample (default 0.20).
    - seed (int): Random seed for reproducibility (default 42).

    Returns:
    - DataFrame: Sampled Spark DataFrame.
    """
    sampled_df = spark_df.sample(withReplacement=False, fraction=fraction,
seed=42)
    return sampled_df

sampled_df = sample_dataframe(spark_df)
```

28. Defining clean_and_prepare_sampled_df()

```
def clean_and_prepare_sampled_df(sampled_df):
    """
    Cleans and prepares the sampled Subway DataFrame by:
    - Renaming 'station_complex' to 'station_name'
    - Dropping unnecessary columns
    - Checking for null and NaN values
    - Creating a 'date' column from 'transit_timestamp'

    Parameters:
    - sampled_df (DataFrame): Input Spark DataFrame.

    Returns:
    - DataFrame: Cleaned Spark DataFrame.
    """
    # Rename column
    sampled_df = sampled_df.withColumnRenamed("station_complex",
"station_name")

    # Drop unnecessary columns
    sampled_df = sampled_df.drop('station_complex_id', 'transit_mode',
'latitude', 'longitude', 'payment_method', 'fare_class_category')

    # Show null counts for checking
    sampled_df.select([count(when(col(c).isNull(), c)).alias(c) for c in
sampled_df.columns]).show(truncate=False)

    # Show any NaN values in numeric columns
    sampled_df.filter(isnan("ridership") |
isnan("transfers")).show(truncate=False)

    # Create a date column
    sampled_df = sampled_df.withColumn("date", to_date("transit_timestamp"))

    return sampled_df

sampled_df = clean_and_prepare_sampled_df(sampled_df)
```

29. Defining create_holiday_features()

```
# holiday feature engineering function
def create_holiday_features(df):
    import holidays
    us_holidays = holidays.US(years=range(2015, 2031))
    holiday_pd_df = pd.DataFrame({
        "date": list(us_holidays.keys()),
        "holiday_name": list(us_holidays.values()),
        "is_holiday": 1
    })
    holiday_sdf = spark.createDataFrame(holiday_pd_df)
    holiday_sdf = holiday_sdf.withColumn("date", to_date(col("date")))

    # Merge
    df = df.withColumn("date", to_date(col("transit_timestamp")))
    df = df.join(holiday_sdf, on="date", how="left")

    # Fill null holidays
    df = df.withColumn("is_holiday", coalesce(col("is_holiday"), lit(0)))
    df = df.withColumn("holiday_name", coalesce(col("holiday_name"),
lit("Unknown Holiday")))

    return df
```

30. Defining match_holiday_dates()

```
from pyspark.sql.functions import to_date

def match_holiday_dates(subway_df, holiday_df):
    """
    Converts 'date' columns to DateType, finds matching dates between subway
and holiday dataframes,
    and prints the number of matching dates.

    Parameters:
    - subway_df (DataFrame): Subway ridership DataFrame.
    - holiday_df (DataFrame): Holiday DataFrame.

    Returns:
    - int: Number of matching dates.
    """
    # Ensure date columns are properly formatted
    subway_df = subway_df.withColumn("date", to_date("date"))
    holiday_df = holiday_df.withColumn("date", to_date("date"))
    # Find matching dates
    matching_dates = subway_df.select("date").distinct() \
                              .intersect(holiday_df.select("date").distinct())

    # Count matches
    matching_count = matching_dates.count()

    print(f"Number of matching dates: {matching_count}")

    return matching_count
```

31. Defining replace_nulls_with_defaults()

```
from pyspark.sql.functions import coalesce, lit

def replace_nulls_with_defaults(df):
    """
    Replaces NULL values in 'is_holiday' column with 0 and 'holiday_name'
column with an empty string.

    Args:
    - df: The DataFrame to apply the changes on.

    Returns:
    - DataFrame with NULLs replaced by default values.
    """
    return df \
        .withColumn("is_holiday", coalesce(df["is_holiday"], lit(0))) \
        .withColumn("holiday_name", coalesce(df["holiday_name"], lit("Unknown
Holiday")))  # Replacing NULLs with 'Unknown Holiday'

# Usage
spark_df = replace_nulls_with_defaults(spark_df)

# Show the updated DataFrame
spark_df.show()
```

32. Reading the weather data

```
weather_raw_df =
spark.read.csv("gs://weather-api-mta-yb/cleaned/weather-data-ny-latest.csv",
header=False)
```

33. Defining clean_weather_data()

```
# weather data cleaning function
def clean_weather_data(weather_raw_df):
    header = weather_raw_df.first()
    weather_df = weather_raw_df.filter(weather_raw_df._c0 != header[0])
    for i, col_name in enumerate(header):
        weather_df = weather_df.withColumnRenamed(f"_c{i}", col_name)
    return weather_df
```

34. Weather Data EDA

```
weather_df.show(3, vertical=True)

# missing values
weather_df.select([count(when(isnull(c), c)).alias(c) for c in
weather_df.columns]).show()

# records where precipitation is greater than 0 or temperature is not constant
weather_df.filter((weather_df.precipitation_mm > 0) | (weather_df.temperature_F
!= 0)).show(10)

# sorting by precipitation or selecting specific weather events like rain/snow
weather_df.filter(weather_df.precipitation_mm > 0).orderBy('precipitation_mm',
ascending=False).show(10)

# descriptive stats on weather_df
weather_df.describe(['temperature_F', 'apparent_temperature_F', 'rain_mm',
'snowfall_cm', 'precipitation_mm']).show()
```

35. Defining merge_ridership_weather_by_hour()

```
def merge_ridership_weather_by_hour(sampled_df, weather_df,
                                    ridership_time_col="transit_timestamp",
                                    weather_time_col="time"):
    """
    Merges ridership and weather dataframes on both 'date' and 'hour' columns.

    Parameters:
    - spark_df: Spark DataFrame containing MTA ridership data
    - weather_df: Spark DataFrame containing weather data
    - ridership_time_col: Column in spark_df that contains timestamp
    - weather_time_col: Column in weather_df that contains timestamp

    Returns:
    - merged_df: Spark DataFrame joined on 'date' and 'hour'
    """

    # Extract date and hour from ridership timestamp
    sampled_df = sampled_df \
        .withColumn("date", to_date(spark_df[ridership_time_col])) \
        .withColumn("hour", hour(spark_df[ridership_time_col]))

    # Extract date and hour from weather timestamp
    weather_df = weather_df \
        .withColumn("date", to_date(weather_df[weather_time_col])) \
        .withColumn("hour", hour(weather_df[weather_time_col]))

    # Perform the join on date and hour
    merged_df = sampled_df.join(weather_df, on=["date", "hour"], how="left")

    return merged_df

merged_df = merge_ridership_weather_by_hour(sampled_df, weather_df,
ridership_time_col="transit_timestamp", weather_time_col="time")
```

36. Missing Values

```
# checking missing values
merged_df.select([count(when(isnull(c), c)).alias(c) for c in
merged_df.columns]).show()
```

37. Defining add_date_parts()

```
# extracting Year, Month, Day of Week
from pyspark.sql.functions import year, date_format, col, when
from pyspark.sql.functions import year, date_format, col, when

def add_date_parts(df):
    df = df.withColumn("year", year("transit_timestamp"))
    df = df.withColumn("month", date_format("transit_timestamp", "MMM"))  #
Jan, Feb, ...
    df = df.withColumn("day_name", date_format("transit_timestamp", "E"))  #
Sun, Mon, ...

    df = df.withColumn(
        "day_of_week",
        when(col("day_name") == "Mon", 1)
        .when(col("day_name") == "Tue", 2)
        .when(col("day_name") == "Wed", 3)
        .when(col("day_name") == "Thu", 4)
        .when(col("day_name") == "Fri", 5)
        .when(col("day_name") == "Sat", 6)
        .when(col("day_name") == "Sun", 7)
        .otherwise(None)
    )

    return df
```

38. Defining add_season()

```
def add_season(df):
    return df.withColumn(
        "season",
        when(col("month").isin("Dec", "Jan", "Feb"), "Winter")
        .when(col("month").isin("Mar", "Apr", "May"), "Spring")
        .when(col("month").isin("Jun", "Jul", "Aug"), "Summer")
        .when(col("month").isin("Sep", "Oct", "Nov"), "Fall")
        .otherwise("Unknown")
    )
```

39. Defining add_hour_labels()

```
# extracting hour + label time of the day
from pyspark.sql.functions import hour

from pyspark.sql.functions import hour, col, when

def add_hour_labels(df):
    return df \
        .withColumn("hour", hour("transit_timestamp")) \
        .withColumn("time_of_day", when((col("hour") >= 0) & (col("hour") < 4),
"Late Night")
                                   .when((col("hour") >= 4) & (col("hour") <
8), "Early Morning")
                                   .when((col("hour") >= 8) & (col("hour") <
12), "Morning")
                                   .when((col("hour") >= 12) & (col("hour") <
16), "Afternoon")
                                   .when((col("hour") >= 16) & (col("hour") <
20), "Evening")
                                   .when((col("hour") >= 20) & (col("hour") <
24), "Night")
                                   .otherwise("Unknown"))
```

40. Defining add_flags()

```
# adding is_weekend and is_peakhour flags
def add_flags(df):
    return df \
        .withColumn("is_weekend", when(col("day_of_week").isin(6, 7),
1).otherwise(0)) \
        .withColumn("is_peakhour", when(((col("hour") >= 7) & (col("hour") <=
10)) |
                                        ((col("hour") >= 16) & (col("hour") <=
19)), 1).otherwise(0))
```

41. Defining feature_engineering()

```
def feature_engineering(merged_df):
    merged_df = add_date_parts(merged_df)
    merged_df = add_season(merged_df)
    merged_df = add_hour_labels(merged_df)
    merged_df = add_flags(merged_df)
    return merged_df
```

42. Dealing with Outliers

```
#  holidays package failed to install, thus could not turn this into a
‘reusable’ function
# calculating the z score to identify outliers
from pyspark.sql.functions import col, mean, stddev

stats = merged_df.select(mean("ridership").alias("mean_val"),
stddev("ridership").alias("std_val")).first()
mean_val = stats["mean_val"]
std_val = stats["std_val"]

# Add z-score column to merged_df
merged_df = merged_df.withColumn("ridership_z_score", (col("ridership") -
mean_val) / std_val)

# creating another dataframe just for outliers
# Filter outliers
z_outliers_df = merged_df.filter((col("ridership_z_score") > 4) |
(col("ridership_z_score") < -4))

# Group by 'is_holiday' and count the number of outliers
outliers_count_by_holiday = z_outliers_df.groupBy("is_holiday").count()

# Show the result
outliers_count_by_holiday.show()

merged_df = merged_df.filter((merged_df["ridership_z_score"] >= -4) &
(z_outliers_df["ridership_z_score"] <= 4))
merged_df.count()
```

43. Median Calculation and Target Variable Creation

```
from pyspark.sql import functions as F

# Step 1: Calculate median ridership for each time_of_day and station
group_medians = (
    merged_df.groupBy("time_of_day", "station_name")
      .agg(F.expr("percentile_approx(ridership, 0.5,
100)").alias("median_ridership"))
)

# Step 2: Join the median values back to the original dataframe
merged_df = merged_df.join(group_medians, on=["time_of_day", "station_name"],
how="left")

# Step 3: Create the 'high_ridership' binary column
merged_df = merged_df.withColumn(
    "high_ridership",
    (merged_df["ridership"] > merged_df["median_ridership"]).cast("integer")
)

# Total row count
total_count = merged_df.count()

# Value counts + percentage
merged_df.groupBy("high_ridership") \
    .count() \
    .withColumn("percent", round((col("count") / total_count) * 100, 2)) \
    .show()
```

44. Preparing data for ML pipeline

```
# IMPORTANT: last sanity check to remove any records that have NULL values in
any of our important columns
list_of_columns = ['hour', 'station_name', 'borough', 'transfers',
'holiday_name', 'is_holiday', 'temperature_F',
                   'apparent_temperature_F', 'rain_mm', 'snowfall_cm',
'precipitation_mm', 'year', 'month', 'day_name',
                   'season', 'time_of_day', 'is_weekend', 'is_peakhour',
'ridership_z_score', 'high_ridership']

merged_df = merged_df.dropna(subset=list_of_columns, how='any')

merged_df = merged_df.drop("holiday_name")
```

45. Defining prepare_for_modeling()

```
def prepare_for_modeling(df):

      string_columns = ['time_of_day', 'station_name', 'hour', 'borough',
'year', 'month', 'day_name', 'season']
      double_columns = ['transfers', 'temperature_F', 'apparent_temperature_F',
      'rain_mm', 'snowfall_cm', 'precipitation_mm', 'ridership_z_score']
      binary_flags = [ 'is_holiday', 'is_weekend', 'is_peakhour',]
      target = ['high_ridership']

for col_name in string_columns:
    merged_df = merged_df.withColumn(col_name, col(col_name).cast("string"))
for col_name in double_columns:
    merged_df = merged_df.withColumn(col_name, col(col_name).cast("double"))
for col_name in binary_flags:
    merged_df = merged_df.withColumn(col_name, col(col_name).cast("double"))
for col_name in target:
    merged_df = merged_df.withColumn(col_name, col(col_name).cast("double"))

    df = df.dropna()
    return df
```

46. Creating Machine Learning Pipeline

```
# creating an indexer for string based columns
categorical_column_list = string_columns
categorical_index_list = [c + "Index" for c in categorical_column_list]
string_indexer = StringIndexer(inputCols=categorical_column_list,
outputCols=categorical_index_list, handleInvalid="keep")

# create an encoder for indexes
categorical_encoded_list = [c + "Vector" for c in categorical_column_list]
encoder = OneHotEncoder(inputCols=categorical_index_list,
outputCols=categorical_encoded_list, dropLast=True, handleInvalid="keep")

# list of continuous data columns
continuous_column_list = double_columns

# assemble the continuous columns into a vector
continuous_assembler = VectorAssembler(inputCols=continuous_column_list,
outputCol="continuousVector")

# run the continuousVector Vector through the scaler
scaler = StandardScaler(inputCol="continuousVector",
outputCol="continuousScaled")

# assemble all of your Vectors into one large vector named 'features'
feature_columns = categorical_encoded_list + ["continuousScaled"] +
binary_flags

# feature_columns = categorical_encoded_list
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")

# test this much of pipeline
test_pipeline = Pipeline(stages=[string_indexer, encoder, continuous_assembler,
scaler, assembler])
# test_pipeline = Pipeline(stages=[string_indexer, encoder,
continuous_assembler, scaler, assembler])

transformed_sdf = test_pipeline.fit(merged_df).transform(merged_df)
transformed_sdf.show(n=2, vertical=True)
```

47. Saving to trusted folder

```
# saving to trusted folder
# transformed_sdf.save.format(
trusted_filename = f"{trusted_folder}/"
transformed_sdf.write.parquet(trusted_filename, mode='overwrite')
```

48. Creating Logistic Regression Estimator

```
# initialising
lr = LogisticRegression(featuresCol="features", labelCol="high_ridership",
                       regParam=0.01, elasticNetParam=1.0)

# assemble all of the steps into a pipeline
ml_pipeline = Pipeline(stages=[string_indexer, encoder, continuous_assembler,
scaler, assembler, lr])

# split the dataframe into training and test sets
trainData, testData = merged_df.randomSplit([0.7, 0.3], seed=42)

lr_model = ml_pipeline.fit(trainData)

test_results = lr_model.transform(testData)
```

49. Evaluator

```
evaluator = BinaryClassificationEvaluator(
    labelCol="high_ridership",
    rawPredictionCol="rawPrediction",
    metricName="areaUnderROC")

roc_auc = evaluator.evaluate(test_results)
print(f"ROC AUC: {roc_auc:.5f}")
```

50. ParamGrid and CV

```
# create a BinaryClassification Evaluator to evaluate how well the model works
evaluator = BinaryClassificationEvaluator(metricName="areaUnderROC",
labelCol="high_ridership")

# create a grid to hold hyperparameters
grid = ParamGridBuilder()
grid = grid.addGrid(lr.regParam, [0.01, 0.5, 1.0])
grid = grid.addGrid(lr.elasticNetParam, [0.0, 1.0])

# best params [regParam = 0.01, elasticNetParam = 1.0]

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

# show the number of models to be tested
print("Number of models to be tested: ", len(grid))

# creat the CrossValidator
# note the 'estimator' is pointing to the complete ml_pipeline
cv = CrossValidator(estimator=ml_pipeline,
                    estimatorParamMaps=grid,
                    evaluator=evaluator, numFolds=3)

# this will train all of the models
all_models = cv.fit(trainData)

# choose the best model
best_model = all_models.bestModel

# show the average metric on all model runs
print(f"Average Metric: {all_models.avgMetrics}")

# use the bestModel to run the model on testData
test_results = best_model.transform(testData)
```

51. Test Results

```
def display_predictions(test_results):
    """
    Displays key columns from the prediction results:
    - station_name, day_name, is_holiday, high_ridership, prediction
    - high_ridership, prediction, probability

    Parameters:
    - test_results (DataFrame): The DataFrame containing model prediction
results.

    Returns:
    - None (just shows outputs)
    """
    print("\n=== Predictions Overview ===")
    test_results.select(
        ["station_name", "day_name", "is_holiday", "high_ridership",
"prediction"]
    ).show(truncate=False)

    print("\n=== Prediction Probabilities ===")
    test_results.select(
        ["high_ridership", "prediction", "probability"]
    ).show(truncate=False)
```

52. Saving Results

```
# Save a subset of test results
test_results.select("station_name", "day_name", "is_holiday", "high_ridership",
                    "prediction").write.csv('/mta/data/test_predictions.csv',
header=True, mode='overwrite')
```

```
print("Sampled predictions saved")
```

53. Confusion Matrix

```
cm =
test_results.groupby('high_ridership').pivot('prediction').count().fillna(0).so
rt('high_ridership', ascending=True).collect()

# Function to calculate recall and precision
def calculate_recall_precision(cm):
    tn = cm[0][1]
    fp = cm[0][2]
    fn = cm[1][1]
    tp = cm[1][2]
    precision = tp / ( tp + fp )
    recall = tp / ( tp + fn )
    accuracy = ( tp + tn ) / ( tp + tn + fp + fn )
    f1_score = 2 * ( ( precision * recall ) / ( precision + recall ) )
    return accuracy, precision, recall, f1_score

accuracy, precision, recall, f1_score =  calculate_recall_precision(cm)

print(accuracy, precision, recall, f1_score)

print("label,0.0,1.0\n", cm[0][0],",",cm[0][1],",",cm[0][2], "\n",
cm[1][0],",",cm[1][1],",",cm[1][2])
```

54. Defining save_metrics_and_predictions()

```
def save_metrics_and_predictions(metrics, test_results, trusted_folder):
    """
    Saves model evaluation metrics and test predictions into the trusted
folder.

    Parameters:
    - metrics (tuple): (accuracy, precision, recall, f1_score)
    - test_results (DataFrame): Spark DataFrame containing test predictions.
    - trusted_folder (str): Path to the trusted folder where files will be
saved.

    Returns:
    - None
    """

    # Unpack metrics
    accuracy, precision, recall, f1_score = metrics
    # Create trusted folder if it does not exist
    #if not os.path.exists(trusted_folder):
     #   os.makedirs(trusted_folder)

    # Save model metrics
    metrics_data = {
        "Metric": ["Accuracy", "Precision", "Recall", "F1 Score"],
        "Value": [accuracy, precision, recall, f1_score]
    }
    metrics_df = pd.DataFrame(metrics_data)
    metrics_df.to_csv(f"{trusted_folder}/model_metrics.csv", index=False)
    print(f"Model metrics saved to {trusted_folder}/model_metrics.csv")

    # Save test predictions
    test_results.select(
        "station_name", "day_name", "is_holiday", "high_ridership",
"prediction"
    ).write.csv(f"{trusted_folder}/test_predictions.csv", header=True,
mode='overwrite')
    print(f"Test predictions saved to {trusted_folder}/test_predictions.csv")

# First calculate your metrics
accuracy, precision, recall, f1_score = calculate_recall_precision(cm)

# saving them
save_metrics_and_predictions(
    (accuracy, precision, recall, f1_score),
    test_results)
```

55. 10 Busiest Stations by predictions

```
# Group by station_name, hour, and time_of_day and compute average prediction
grouped_predictions = test_results.groupBy("station_name", "hour",
"time_of_day") \

.agg(avg("prediction").alias("avg_predicted_high_ridership")) \
                                  .filter(col("avg_predicted_high_ridership") >
0).orderBy(col("avg_predicted_high_ridership").desc())

grouped_predictions.show(10, truncate=False)  # Display top 10 busiest stations
```

Appendix E : Data Visualization Code

56. Defining plot_roc_curve()

```
def plot_roc_curve(test_results, label_col="high_ridership",
raw_pred_col="rawPrediction", sample_fraction=0.1, seed=42):
    """
    This function plots the ROC curve for binary classification results.

    Parameters:
    test_results (DataFrame): The test DataFrame containing predictions.
    label_col (str): The column containing the true labels.
    raw_pred_col (str): The column containing the raw predictions (output of
the model).
    sample_fraction (float): Fraction of the data to sample for plotting
(default is 0.1).
    seed (int): Random seed for sampling (default is 42).
    """

    # Extract probability/score from rawPrediction (assuming rawPrediction is a
vector)
    test_with_scores = test_results.withColumn("score",
vector_to_array(raw_pred_col)[1]) \
                                   .select("score", label_col)

    # Sample the data for plotting if needed
    roc_data = test_with_scores.sample(fraction=sample_fraction,
seed=seed).toPandas()

    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(roc_data[label_col], roc_data["score"])
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='blue', label=f'ROC curve (AUC = {roc_auc:.4f})')
    plt.plot([0, 1], [0, 1], color='gray', linestyle='--')  # diagonal line
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve')
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()

plot_confusion_matrix(test_results)
```

57. Saving sample predictions

```
# saving sample predictions
# Save a subset of test results to GCS
test_results.select("station_name", "day_name", "is_holiday", "high_ridership",
"prediction") \
    .write.mode("overwrite") \
    .option("header", True) \
    .csv("gs://my-bigdata-project-yb/test_results/sample/sample_predictions")

print("Sampled predictions saved to GCS")
```

58. Saving sample predictions (all columns after vector columns)

```
columns_to_drop = [field.name for field in test_results.schema.fields
                   if str(field.dataType).startswith("StructType")
                   or "Vector" in str(field.dataType)
                   or "Array" in str(field.dataType)]

clean_df = test_results.drop(*columns_to_drop)

# Save to GCS
clean_df.write.mode("overwrite") \
    .option("header", True) \
    .csv("gs://my-bigdata-project-yb/test_results/sample/sample_predictions")

print("Sample predictions (all supported columns) saved to GCS")
```

59. Defining save_plot_to_gcs()

```
def save_plot_to_gcs(fig, filename):
    """
    Saves a matplotlib/seaborn figure to GCS using predefined BUCKET_NAME and
FIGURE_FOLDER.

    Parameters:
    - fig: The matplotlib figure object (e.g., plt.gcf())
    - filename: Just the name of the file, like 'plot1.png'
    """
    img_data = io.BytesIO()
    fig.savefig(img_data, format='png', bbox_inches='tight')
    img_data.seek(0)

    storage_client = storage.Client()
    bucket = storage_client.bucket(BUCKET_NAME)
    blob = bucket.blob(f"{FIGURE_FOLDER}/{filename}")
    blob.upload_from_file(img_data)

    print(f" Saved plot to gs://{BUCKET_NAME}/{filename}")
```

60. Defining plot_confusion_matrix()

```
def plot_confusion_matrix(test_results):
    # Compute confusion matrix
    cm_raw = (test_results.groupBy('high_ridership')
              .pivot('prediction')
              .count()
              .fillna(0)
              .sort('high_ridership', ascending=True)
              .collect())

    tn = cm_raw[0][1]  # Actual 0, Predicted 0
    fp = cm_raw[0][2]  # Actual 0, Predicted 1
    fn = cm_raw[1][1]  # Actual 1, Predicted 0
    tp = cm_raw[1][2]  # Actual 1, Predicted 1

    # Convert to Pandas DataFrame
    data = [[tn, fp], [fn, tp]]
    df_cm = pd.DataFrame(data, index=['Actual 0', 'Actual 1'],
                  columns=['Predicted 0', 'Predicted 1'])

    # Plot the heatmap
    plt.figure(figsize=(6, 5))
    sns.heatmap(df_cm, annot=True, fmt='d', cmap='Blues')
    plt.title("Confusion Matrix")
    plt.ylabel("Actual")
    plt.xlabel("Predicted")
    plt.tight_layout()
    plt.show()

    fig = plt.gcf()
    return fig

fig1 = plot_confusion_matrix(test_results)
save_plot_to_gcs(fig1, "confusion-matrix.png")
```

61. Defining extract_flat_feature_names()

```
def extract_flat_feature_names(best_model):
    string_indexer_model = best_model.stages[0]
    encoder_model = best_model.stages[1]
    scaler_model = best_model.stages[3]
    continuous_columns = best_model.stages[2].getInputCols()
    binary_flags = ['is_holiday', 'is_weekend', 'is_peakhour']

    index_input_cols = string_indexer_model.getInputCols()
    category_sizes = encoder_model.categorySizes  # real sizes after dropLast

    # Reconstruct OHE feature names
    encoded_feature_names = []
    for col, size in zip(index_input_cols, category_sizes):
        encoded_feature_names.extend([f"{col}_{i}" for i in range(size)])

    # Scaled continuous features
    scaled_continuous = [f"{col}_scaled" for col in continuous_columns]

    # Combine all
    return encoded_feature_names + scaled_continuous + binary_flags

feature_names = extract_flat_feature_names(best_model)
logreg_model = best_model.stages[-1]
```

62. Defining visualize_logreg_coefficients()

```
# version plotting top N features
def visualize_logreg_coefficients(model, feature_names, top_n=20):

    coeffs = model.coefficients.toArray()

    if len(coeffs) != len(feature_names):
        raise ValueError(f"Mismatch: {len(coeffs)} coefficients vs
{len(feature_names)} feature names")

    coef_df = pd.DataFrame({
        'Feature': feature_names,
        'Coefficient': coeffs
    })

    # Sort by absolute value and pick top N
    top_coef_df =
coef_df.reindex(coef_df.Coefficient.abs().sort_values(ascending=False).index).h
ead(top_n)

    # Print
    print("\nTop Logistic Regression Coefficients:\n")
    print(top_coef_df)

    # Plot
    plt.figure(figsize=(10, 6))
    sns.barplot(data=top_coef_df, x='Coefficient', y='Feature',
palette='coolwarm')
    plt.title(f"Top {top_n} Logistic Regression Coefficients")
    plt.tight_layout()
    plt.show()
    fig = plt.gcf()
    return fig

fig1 = visualize_logreg_coefficients(logreg_model, feature_names, top_n=5)
save_plot_to_gcs(fig1, "logreg-coeffs.png")
```

63. Defining get_feature_category()

```
def get_feature_category(feature_name):
    if "station_name_" in feature_name:
        return "station_name"
    elif "time_of_day_" in feature_name:
        return "time_of_day"
    elif "hour_" in feature_name:
        return "hour"
    elif "borough_" in feature_name:
        return "borough"
    elif "season_" in feature_name:
        return "season"
    elif "month_" in feature_name:
        return "month"
    elif "day_name_" in feature_name:
        return "day_name"
    elif "year_" in feature_name:
        return "year"
    elif "rain_mm" in feature_name or "temperature" in feature_name or
"precipitation" in feature_name or "ridership_z_score" in feature_name:
        return "continuous"
    elif feature_name in ['is_holiday', 'is_weekend', 'is_peakhour']:
        return "binary_flag"
    else:
        return "other"
```

64. Defining summarize_feature_importance_by_category()

```
def summarize_feature_importance_by_category(model, feature_names, top_n=10)
    coeffs = model.coefficients.toArray()

    if len(coeffs) != len(feature_names):
        raise ValueError(f"Mismatch: {len(coeffs)} coefficients vs
{len(feature_names)} feature names")

    # Build DataFrame
    coef_df = pd.DataFrame({
        'Feature': feature_names,
        'Coefficient': coeffs
    })

    coef_df = coef_df[coef_df['Coefficient'] != 0]  # drop zeros
    coef_df['Importance'] = coef_df['Coefficient'].abs()
    coef_df['Category'] = coef_df['Feature'].apply(get_feature_category)

    # Group by category
    group_df = coef_df.groupby('Category')['Importance'].sum().reset_index()
    group_df = group_df.sort_values(by='Importance', ascending=False)

    # Print summary
    print("\nTotal Feature Importance by Category:\n")
    print(group_df)

    # Plot
    plt.figure(figsize=(10, 6))
    sns.barplot(data=group_df.head(top_n), x='Importance', y='Category',
palette='Set2')
    plt.title(f"Top {top_n} Feature Categories by Total Importance")
    plt.xlabel("Total |Coefficient|")
    plt.tight_layout()
    plt.show()

    return group_df

category_importance = summarize_feature_importance_by_category(logreg_model,
feature_names, top_n=10)
```

65. Defining plot_feature_category_importance_line()

```
def plot_feature_category_importance_line(df):
    # Sort by importance (descending to ascending for better left-to-right
trend)
    df_sorted = df.sort_values(by='Importance', ascending=False)

    # Plot
    plt.figure(figsize=(10, 6))
    plt.plot(df_sorted['Category'], df_sorted['Importance'], marker='o',
linestyle='-')
    plt.xticks(rotation=45)
    plt.title("Feature Category Importance (by Total |Coefficient|)")
    plt.xlabel("Feature Category")
    plt.ylabel("Total Importance (|Coefficient|)")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

plot_feature_category_importance_line(category_importance)
```

66. Displaying distribution of predictions

```
predictions_df.groupBy("prediction").count().show()
```

67. Displaying Prediction vs Actual (Confusion Matrix Count)

```
predictions_df.groupBy("high_ridership",
"prediction").count().orderBy("high_ridership").show()
```

68. Sampling data for further visualizations

```
# Random 0.10 fraction of data
sample_df = predictions_df.sample(withReplacement=False, fraction=0.10,
seed=42)
```

69. Defining visualize_prediction_analysis()

```
def visualize_prediction_analysis(sample_df):
    sns.set(style='whitegrid')

    # 1. Class Distribution (Actual)
    class_df = sample_df.groupBy("high_ridership").count().toPandas()
    plt.figure(figsize=(5, 4))
    sns.barplot(data=class_df, x="high_ridership", y="count", palette='pastel')
    plt.title("Class Distribution (Actual High Ridership)")
    plt.xlabel("Actual High Ridership")
    plt.ylabel("Count")
    plt.show()

    # 2. Prediction vs Actual
    pred_df = (
        sample_df.groupBy("prediction", "high_ridership").count()
                 .toPandas()
    )
    plt.figure(figsize=(6, 4))
    sns.barplot(data=pred_df, x="prediction", y="count", hue="high_ridership",
palette='Set2')
    plt.title("Prediction vs Actual")
    plt.xlabel("Predicted Label")
    plt.ylabel("Count")
    plt.legend(title='Actual')
    plt.show()

    # 3. Error Analysis
    error_df = (
        sample_df.withColumn("error_type", F.when(F.col("prediction") ==
F.col("high_ridership"), "Correct").otherwise("Incorrect"))
                 .groupBy("error_type").count()
                 .toPandas()
    )
    plt.figure(figsize=(5, 4))
    sns.barplot(data=error_df, x='error_type', y='count', palette='coolwarm')
    plt.title("Prediction Accuracy Breakdown")
    plt.xlabel("Prediction Outcome")
    plt.ylabel("Count")
    plt.show()

    # 4. Confusion Matrix (Sklearn style)
    cm_pd = sample_df.select("high_ridership", "prediction").toPandas()
    cm = confusion_matrix(cm_pd["high_ridership"], cm_pd["prediction"])
    plt.figure(figsize=(5, 4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1],
yticklabels=[0, 1])
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

visualize_prediction_analysis(sample_df)
```

70. Defining plot_ridership_by_hour()

```
def plot_ridership_by_hour(df):
    sns.set(style="darkgrid")

    df_pd = (
        df.groupBy("hour")
          .agg(F.avg("high_ridership").alias("avg_high_ridership"))
          .toPandas()
          .sort_values("hour")
    )

    plt.figure(figsize=(10, 5))
    sns.lineplot(data=df_pd, x="hour", y="avg_high_ridership", marker="o")
    plt.title("Avg High Ridership by Hour", fontsize=14)
    plt.ylabel("Avg High Ridership")
    plt.xlabel("Hour of Day")
    plt.tight_layout()
    plt.show()

plot_ridership_by_hour(sample_df)
```

71. Defining plot_ridership_by_borough()

```
def plot_ridership_by_borough(df):
    sns.set_context("talk")

    df_pd = (
        df.groupBy("borough")
          .agg(F.avg("high_ridership").alias("avg_high_ridership"))
          .toPandas()
          .sort_values("avg_high_ridership", ascending=True)
    )

    plt.figure(figsize=(8, 5))
    sns.barplot(data=df_pd, x="avg_high_ridership", y="borough",
palette="coolwarm")
    plt.title("Avg High Ridership by Borough")
    plt.xlabel("Avg High Ridership")
    plt.ylabel("Borough")
    plt.tight_layout()
    plt.show()

plot_ridership_by_borough(sample_df)
```

72. Defining plot_ridership_by_binary_col()

```
def plot_ridership_by_binary_col(df, col_name):
    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set_palette("Set2")

    df_pd = (
        df.groupBy(col_name)
          .agg(F.avg("high_ridership").alias("avg_high_ridership"))
          .toPandas()
    )

    plt.figure(figsize=(6, 4))
    sns.barplot(data=df_pd, x=col_name, y="avg_high_ridership")
    plt.title(f"Avg High Ridership by {col_name.replace('_', ' ').title()}")
    plt.ylabel("Avg High Ridership")
    plt.xlabel(col_name.replace("_", " ").title())
    plt.xticks([0, 1], ["No", "Yes"])
    plt.tight_layout()
    plt.show()

plot_ridership_by_binary_col(sample_df, "is_holiday")
plot_ridership_by_binary_col(sample_df, "is_weekend")
plot_ridership_by_binary_col(sample_df, "is_peakhour")
```

73. Defining plot_top_10_stations()

```
def plot_top_10_stations(df):
    import matplotlib.pyplot as plt
    import seaborn as sns
    sns.set_style("white")

    df_pd = (
        df.groupBy("station_name")
          .agg(F.avg("high_ridership").alias("avg_high_ridership"))
          .toPandas()
          .sort_values("avg_high_ridership", ascending=False)
          .head(10)
    )

    plt.figure(figsize=(10, 6))
    sns.barplot(data=df_pd, x="avg_high_ridership", y="station_name",
palette="magma")
    plt.title("Top 10 Stations by Avg High Ridership")
    plt.xlabel("Avg High Ridership")
    plt.ylabel("Station Name")
    plt.tight_layout()
    plt.show()

plot_top_10_stations(sample_df)
```

74. Defining describe_weather_columns()

```
def describe_weather_columns(df, columns):
    from pyspark.sql import functions as F

    summary_list = []

    for col in columns:
        stats = df.select(
            F.min(col).alias("min"),
            F.expr(f"percentile_approx({col}, 0.25)").alias("Q1"),
            F.mean(col).alias("mean"),
            F.expr(f"percentile_approx({col}, 0.5)").alias("median"),
            F.expr(f"percentile_approx({col}, 0.75)").alias("Q3"),
            F.max(col).alias("max")
        ).toPandas().round(2)
        stats.insert(0, "feature", col)
        summary_list.append(stats)

    # Combine all summaries into one DataFrame
    import pandas as pd
    summary_df = pd.concat(summary_list, ignore_index=True)
    return summary_df

describe_weather_columns(sample_df, ['temperature_F', 'apparent_temperature_F',
'rain_mm', 'snowfall_cm', 'precipitation_mm'])
```

75. Defining plot_temp_vs_ridership()

```
def plot_temp_vs_ridership(df, group_col="hour"):
    import matplotlib.pyplot as plt
    import seaborn as sns
    import pandas as pd
    sns.set(style="whitegrid")

    # Aggregate in PySpark
    df_pd = (
        df.groupBy(group_col)
          .agg(
              F.avg("temperature_F").alias("mean_temp_F"),
              F.avg("apparent_temperature_F").alias("mean_apparent_temp_F"),
              F.avg("high_ridership").alias("avg_high_ridership")
          )
          .toPandas()
    )

    # Sort by hour if applicable
    if group_col == "hour":
        df_pd[group_col] = df_pd[group_col].astype(int)
        df_pd = df_pd.sort_values(group_col)

    # Plotting
    plt.figure(figsize=(10, 5))
    sns.set_palette("Set2")
    ax = sns.lineplot(data=df_pd, x=group_col, y="mean_temp_F",
label="Temperature (°F)", marker="o")
    sns.lineplot(data=df_pd, x=group_col, y="mean_apparent_temp_F",
label="Apparent Temp (°F)", marker="s")

    ax2 = ax.twinx()
    sns.lineplot(data=df_pd, x=group_col, y="avg_high_ridership", label="High
Ridership Rate", ax=ax2, color="black", marker="D", linestyle="--")

    ax.set_xlabel(group_col.replace("_", " ").title())
    ax.set_ylabel("Avg Temperature (°F)")
    ax2.set_ylabel("Avg High Ridership")
    plt.title(f"Avg Temperature vs Avg High Ridership by
{group_col.replace('_', ' ').title()}")
    ax.legend(loc="upper left")
    ax2.legend(loc="upper right")
    plt.tight_layout()
    plt.show()

plot_temp_vs_ridership(sample_df, group_col="hour")
plot_temp_vs_ridership(sample_df, group_col="season")
```

76. Defining plot_line_avg_ridership_by_time_of_day()

```
def plot_line_avg_ridership_by_time_of_day(df):
    import matplotlib.pyplot as plt
    import seaborn as sns
    import pandas as pd
    sns.set_theme(style="whitegrid")
    sns.set_context("notebook", font_scale=1.2)

    df_pd = (
        df.groupBy("time_of_day")
          .agg(F.avg("high_ridership").alias("avg_high_ridership"))
          .toPandas()
    )

    time_order = ["Early Morning", "Morning", "Afternoon", "Evening", "Night",
"Late Night"]
    df_pd["time_of_day"] = pd.Categorical(df_pd["time_of_day"],
categories=time_order, ordered=True)
    df_pd = df_pd.sort_values("time_of_day")

    plt.figure(figsize=(10, 5))
    sns.lineplot(data=df_pd, x="time_of_day", y="avg_high_ridership",
marker="o", linewidth=2, color="#2c7bb6")
    plt.title("Avg High Ridership by Time of Day", fontsize=14)
    plt.xlabel("Time of Day")
    plt.ylabel("Avg High Ridership")
    plt.tight_layout()
    plt.show()

plot_line_avg_ridership_by_time_of_day(sample_df)
```

77. Defining plot_hourly_high_ridership()

```
def plot_hourly_high_ridership(df):
    import matplotlib.pyplot as plt
    import seaborn as sns
    import pandas as pd
    sns.set_theme(style="whitegrid")
    sns.set_context("notebook", font_scale=1.2)

    df_pd = (
        df.groupBy("hour")
          .agg(F.avg("high_ridership").alias("avg_high_ridership"))
          .toPandas()
    )

    # Convert hour to int and sort
    df_pd["hour"] = df_pd["hour"].astype(int)
    df_pd = df_pd.sort_values("hour")

    # Plot
    plt.figure(figsize=(10, 5))
    sns.lineplot(data=df_pd, x="hour", y="avg_high_ridership", marker="o",
linewidth=2, color="#1f77b4")
    plt.title("Hourly Avg High Ridership Trend", fontsize=14)
    plt.xlabel("Hour of Day (0–23)")
    plt.ylabel("Avg High Ridership")
    plt.xticks(range(0, 24))
    plt.tight_layout()
    plt.show()

plot_hourly_high_ridership(sample_df)
```

78. Defining

```
def plot_avg_ridership_by_bools(df):
    import matplotlib.pyplot as plt
    import seaborn as sns
    import pandas as pd
    sns.set_theme(style="whitegrid")
    sns.set_context("notebook", font_scale=1.2)

    # Aggregate each column separately and merge for line chart
    weekend_df =
df.groupBy("is_weekend").agg(F.avg("high_ridership").alias("avg_high_ridership"
)).toPandas()
    holiday_df =
df.groupBy("is_holiday").agg(F.avg("high_ridership").alias("avg_high_ridership"
)).toPandas()
    peak_df =
df.groupBy("is_peakhour").agg(F.avg("high_ridership").alias("avg_high_ridership
")).toPandas()

    # Sort by binary flag (0 then 1)
    weekend_df = weekend_df.sort_values("is_weekend")
    holiday_df = holiday_df.sort_values("is_holiday")
    peak_df = peak_df.sort_values("is_peakhour")

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

    sns.lineplot(data=weekend_df, x="is_weekend", y="avg_high_ridership",
marker="o", label="Weekend", linewidth=2)
    sns.lineplot(data=holiday_df, x="is_holiday", y="avg_high_ridership",
marker="s", label="Holiday", linewidth=2)
```

```
    sns.lineplot(data=peak_df, x="is_peakhour", y="avg_high_ridership",
marker="^", label="Peak Hour", linewidth=2)

    plt.title("Avg High Ridership by Weekend, Holiday, and Peak Hour")
    plt.xlabel("Binary Value (0 = No, 1 = Yes)")
    plt.ylabel("Avg High Ridership")
    plt.xticks([0, 1], ["No", "Yes"])
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_avg_ridership_by_bools(sample_df)
```

79. Defining plot_transfers_vs_temperature()

```
def plot_transfers_vs_temperature(df):
    import matplotlib.pyplot as plt
    import seaborn as sns
    from pyspark.sql import functions as F

    # Aggregate in PySpark
    sp_temp_rider = (
        spark_df.groupBy("temperature_F")
                .agg(F.sum("transfers").alias("total_transfers"))
                .orderBy("temperature_F")
                .toPandas()
    )

    # Plot in seaborn
    plt.figure(figsize=(8, 6))
    sns.scatterplot(data=sp_temp_rider, x="temperature_F", y="total_transfers",
alpha=0.6)
    plt.title("Transfers vs Temperature")
    plt.xlabel("Temperature (F)")
    plt.ylabel("Total Transfers")
    plt.tight_layout()
    plt.show()

plot_transfers_vs_temperature(sample_df)
```

80. Defining plot_transfers_vs_precipitation()

```
def plot_transfers_vs_precipitation(df):

    # Select relevant columns and convert to Pandas
    sp_rain = sample_df.select("precipitation_mm", "transfers").toPandas()

    # Plot
    plt.figure(figsize=(8, 6))
    sns.scatterplot(data=sp_rain, x="precipitation_mm", y="transfers",
alpha=0.6)
    plt.title("Transfers vs Precipitation")
    plt.xlabel("Precipitation (mm)")
    plt.ylabel("Transfers")
    plt.tight_layout()
    plt.show()

plot_transfers_vs_precipitation(sample_df)
```

81. Defining plot_transfers_by_hour_borough()

```
def plot_transfers_by_hour_borough(df):
    # Convert Spark to Pandas and ensure hour is numeric
    df = df.select("hour", "transfers", "borough").toPandas()
    df["hour"] = pd.to_numeric(df["hour"], errors="coerce")

    # Drop rows with null hours if any
    df = df.dropna(subset=["hour"])

    # Plot
    plt.figure(figsize=(10, 6))
    sns.scatterplot(data=df, x="hour", y="transfers", hue="borough", alpha=0.6)
    plt.title("Transfers by Hour, Colored by Borough")
    plt.xlabel("Hour of Day")
    plt.ylabel("Transfers")
    plt.legend(title="Borough")
    plt.tight_layout()
    plt.show()
plot_transfers_by_hour_borough(sample_df)
```

82. Defining plot_ridership_z_score_outliers()

```
def plot_ridership_z_score_outliers(spark_df):

    # Convert to Pandas
    df = spark_df.select("ridership_z_score").toPandas()

    # Classify values
    def classify_z(z):
        if z > 3:
            return "High Outlier (>3)"
        elif z < -3:
            return "Low Outlier (<-3)"
        else:
            return "Normal (-3 to 3)"

    df["outlier_class"] = df["ridership_z_score"].apply(classify_z)

    # Plot
    plt.figure(figsize=(10, 5))
    sns.histplot(data=df, x="ridership_z_score", hue="outlier_class", bins=50,
palette="Set2", kde=True)
    plt.axvline(3, color='red', linestyle='--', label='Z = 3')
    plt.axvline(-3, color='blue', linestyle='--', label='Z = -3')
    plt.title("Ridership Z-Score Distribution with Outliers Highlighted")
    plt.xlabel("Z-Score")
    plt.ylabel("Count")
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_ridership_z_score_outliers(sample_df)
```

83. Defining plot_ridership_z_score_outliers_only()

```
def plot_ridership_z_score_outliers_only(spark_df):

    # Convert to Pandas
    df = spark_df.select("ridership_z_score").toPandas()

    # Filter only outliers
    outliers_df = df[(df["ridership_z_score"] > 3) | (df["ridership_z_score"] <
-3)].copy()

    # Label outliers
    outliers_df["outlier_class"] = outliers_df["ridership_z_score"].apply(
        lambda z: "High Outlier (>3)" if z > 3 else "Low Outlier (<-3)"
    )

    # Plot
    plt.figure(figsize=(10, 5))
    sns.histplot(data=outliers_df, x="ridership_z_score", hue="outlier_class",
bins=30, palette="Set2", kde=True)
    plt.title("Ridership Z-Score – Outliers Only")
    plt.xlabel("Z-Score")
    plt.ylabel("Count")
    plt.legend()
    plt.tight_layout()
    plt.show()

plot_ridership_z_score_outliers_only(sample_df)
```

84. Defining plot_top_10_stations_by_high_ridership()

```
def plot_top_10_stations_by_high_ridership(sample_df):
    """
    Identifies and visualizes the top 10 stations with the highest average
'high_ridership' (binary) values.
```

```
    Parameters:
    - sample_df: PySpark DataFrame with columns 'station_name' and
'high_ridership'
    """
    # Step 1: Group by station and compute average of high_ridership
(proportion of high ridership)
    top_stations_df = (
        sample_df.groupBy("station_name")
        .agg(F.avg("high_ridership").alias("avg_high_ridership"))
        .orderBy(F.desc("avg_high_ridership"))
        .limit(10)
    )

    # Step 2: Convert to Pandas
    top_stations_pd = top_stations_df.toPandas()

    # Step 3: Plot
    plt.figure(figsize=(12, 6))
    sns.barplot(data=top_stations_pd, x="station_name", y="avg_high_ridership",
palette="magma")
    plt.xticks(rotation=45, ha='right')
    plt.title("Top 10 Stations by Average High Ridership")
    plt.ylabel("Avg High Ridership (Proportion)")
    plt.xlabel("Station Name")
    plt.tight_layout()
    plt.show()

    fig = plt.gcf()
    return fig

fig5 = plot_top_10_stations_by_high_ridership(sample_df)
save_plot_to_gcs(fig5, "Top10 stations by Average high_rdiership.png")
```

85. Defining plot_top_10_station_hour_by_high_ridership()

```
def plot_top_10_station_hour_by_high_ridership(sample_df):
    """
    Groups by station_name and hour to compute average high_ridership,
    shows top 10 combinations, and visualizes them.

    Parameters:
    - sample_df: PySpark DataFrame with 'station_name', 'hour', and
'high_ridership'
    """
    # Step 1: Group and compute average high_ridership
    grouped_df = (
        sample_df.groupBy("station_name", "hour")
        .agg(F.avg("high_ridership").alias("avg_high_ridership"))
        .orderBy(F.desc("avg_high_ridership"))
        .limit(10)
    )

    # Step 2: Convert to Pandas
    grouped_pd = grouped_df.toPandas()
    grouped_pd["station_hour"] = grouped_pd["station_name"] + " @ " +
grouped_pd["hour"].astype(str)

    # Step 3: Plot
    plt.figure(figsize=(12, 6))
    sns.barplot(data=grouped_pd, x="station_hour", y="avg_high_ridership",
palette="coolwarm")
    plt.xticks(rotation=45, ha='right')
    plt.title("Top 10 Station-Hour Pairs by Avg High Ridership")
    plt.ylabel("Avg High Ridership (Proportion)")
    plt.xlabel("Station @ Hour")
    plt.tight_layout()
    plt.show()

    fig = plt.gcf()
    return fig

fig6 = plot_top_10_station_hour_by_high_ridership(sample_df)
save_plot_to_gcs(fig6, "Top10 stations by Average high_rdiership grouped by
station name and hour.png")
```

86. Defining get_top_station_each_hour()

```
def get_top_station_each_hour(sample_df):
    """
    Calculates the average high_ridership for each (station_name, hour) pair,
    then returns the station with the highest average high_ridership for each
hour.
    """
    # Step 1: Group by station_name and hour, compute average high_ridership
    grouped_df = (
        sample_df.groupBy("station_name", "hour")
        .agg(F.avg("high_ridership").alias("avg_high_ridership"))
    )

    # Step 2: Define window to rank stations by high_ridership per hour
    window_spec =
Window.partitionBy("hour").orderBy(F.desc("avg_high_ridership"))

    # Step 3: Rank and filter to get top station per hour
    ranked_df = grouped_df.withColumn("rank", F.row_number().over(window_spec))
    top_stations_per_hour = ranked_df.filter(F.col("rank") ==
1).orderBy("hour")

    # Step 4: Show results
    top_stations_per_hour.select("hour", "station_name",
"avg_high_ridership").show(24, truncate=False)

    return top_stations_per_hour
```

87. Defining plot_top_10_station_timeofday_high_ridership()

```
def plot_top_10_station_timeofday_high_ridership(sample_df):
    """
    Groups by station_name and time_of_day to compute average high_ridership,
    orders by avg_high_ridership and time_of_day, selects top 10,
    and visualizes them as a bar plot.

    Parameters:
    - sample_df: PySpark DataFrame with 'station_name', 'time_of_day', and
'high_ridership'
    """
    # Step 1: Group by station_name and time_of_day and compute average
high_ridership
    grouped_df = (
        sample_df.groupBy("station_name", "time_of_day")
        .agg(F.avg("high_ridership").alias("avg_high_ridership"))
        .orderBy(F.desc("avg_high_ridership"), F.asc("time_of_day"))
        .limit(10)
    )

    # Step 2: Convert to Pandas
    grouped_pd = grouped_df.toPandas()
    grouped_pd["station_time"] = grouped_pd["station_name"] + " @ " +
grouped_pd["time_of_day"]

    # Step 3: Plot
    plt.figure(figsize=(12, 6))
    sns.barplot(data=grouped_pd, x="station_time", y="avg_high_ridership",
palette="flare")
    plt.xticks(rotation=45, ha='right')
    plt.title("Top 10 Station-Time of Day by Avg High Ridership")
    plt.ylabel("Avg High Ridership (Proportion)")
    plt.xlabel("Station @ Time of Day")
    plt.tight_layout()
    plt.show()
```

88. Defining get_crowdiest_station_by()

```
def get_crowdiest_station_by(sample_df, group_col):
    """
    For a given grouping column (e.g., 'hour', 'season'), returns the station
with
    the highest average high_ridership per group.

    Parameters:
    - sample_df: PySpark DataFrame containing 'station_name', 'high_ridership',
and the group_col
    - group_col: str, name of the column to group by along with station_name
(e.g., 'hour')

    Returns:
    - PySpark DataFrame with columns: [group_col, station_name,
avg_high_ridership]
      showing the top station per group value
    """

    # Step 1: Group by group_col + station_name and calculate average
high_ridership
    grouped_df = (
        sample_df.groupBy("station_name", group_col)
        .agg(F.avg("high_ridership").alias("avg_high_ridership"))
    )

    # Step 2: Rank within each group_col to find top station
    window_spec =
Window.partitionBy(group_col).orderBy(F.desc("avg_high_ridership"))

    ranked_df = grouped_df.withColumn("rank", F.row_number().over(window_spec))

    # Step 3: Filter to get only the top-ranked station per group
    top_stations = ranked_df.filter(F.col("rank") == 1).select(group_col,
"station_name", "avg_high_ridership")

    # Optional: Display result
    top_stations.orderBy(group_col).show(truncate=False)

    return top_stations

get_crowdiest_station_by(sample_df, "hour")
get_crowdiest_station_by(sample_df, "time_of_day")
get_crowdiest_station_by(sample_df, "season")
get_crowdiest_station_by(sample_df, "month")
```

– END OF THIS PROJECT –