Download and Load the Datasets

We download the yellow taxi trip data, location lookup, and NYC traffic dataset using `gdown`. Then we load them using Pandas and preview the structure of each dataset.

```
!pip install boto3

import boto3
import os

# ✅ NEW AWS credentials
os.environ["AWS_ACCESS_KEY_ID"] = "AKIAZ3MFCERB52HTRCH6"
os.environ["AWS_SECRET_ACCESS_KEY"] = "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"

# Create a boto3 session
session = boto3.Session(
    aws_access_key_id=os.environ["AWS_ACCESS_KEY_ID"],
    aws_secret_access_key=os.environ["AWS_SECRET_ACCESS_KEY"],
    region_name="us-east-2"
)

# Create the S3 client
s3 = session.client('s3', region_name="us-east-2")

# ✅ List objects under Raw_data/ to verify
response = s3.list_objects_v2(Bucket='myumbcbucket', Prefix='Raw_data/')
for obj in response.get('Contents', []):
    print(obj['Key'])
```

```
Collecting boto3
  Downloading boto3-1.38.9-py3-none-any.whl.metadata (6.6 kB)
Collecting botocore<1.39.0,>=1.38.9 (from boto3)
  Downloading botocore-1.38.9-py3-none-any.whl.metadata (5.7 kB)
Collecting jmespath<2.0.0,>=0.7.1 (from boto3)
  Downloading jmespath-1.0.1-py3-none-any.whl.metadata (7.6 kB)
Collecting s3transfer<0.13.0,>=0.12.0 (from boto3)
  Downloading s3transfer-0.12.0-py3-none-any.whl.metadata (1.7 kB)
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/local/lib/python3.11/dist-packages (from botocore<1.39.0,>=1.38.9
Requirement already satisfied: urllib3!=2.2.0,<3,>=1.25.4 in /usr/local/lib/python3.11/dist-packages (from botocore<1.39.0,>=1.38.9
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil<3.0.0,>=2.1->botocore<1.39
Downloading boto3-1.38.9-py3-none-any.whl (139 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 139.9/139.9 kB 7.3 MB/s eta 0:00:00
Downloading botocore-1.38.9-py3-none-any.whl (13.5 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 13.5/13.5 MB 126.7 MB/s eta 0:00:00
Downloading jmespath-1.0.1-py3-none-any.whl (20 kB)
Downloading s3transfer-0.12.0-py3-none-any.whl (84 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.8/84.8 kB 9.4 MB/s eta 0:00:00
Installing collected packages: jmespath, botocore, s3transfer, boto3
Successfully installed boto3-1.38.9 botocore-1.38.9 jmespath-1.0.1 s3transfer-0.12.0
Raw_data/
Raw_data/Traffic_Volume_Counts_20250405.csv
Raw_data/taxi+_zone_lookup.csv
Raw_data/yellow_tripdata_2025-01.parquet
```

```
# Map of S3 keys to local filenames
files_to_download = {
    'Raw_data/taxi+_zone_lookup.csv': 'taxi_zone_lookup.csv',
    'Raw_data/Traffic_Volume_Counts_20250405.csv': 'traffic_volume.csv',
    'Raw_data/yellow_tripdata_2025-01.parquet': 'yellow_tripdata_2025_01.parquet'
}

for s3_key, local_filename in files_to_download.items():
    s3.download_file('myumbcbucket', s3_key, local_filename)
    print(f"✅ Downloaded: {local_filename}")
```

```
✅ Downloaded: taxi_zone_lookup.csv
✅ Downloaded: traffic_volume.csv
✅ Downloaded: yellow_tripdata_2025_01.parquet
```

```
import pandas as pd

# Load and view the top rows of yellow_trip_01.parquet
df_yellow = pd.read_parquet("/content/yellow_tripdata_2025_01.parquet")
print("Yellow Trip Data:")
print(df_yellow.head())

# Load and view the top rows of location_lookup.csv
df_location = pd.read_csv("/content/taxi_zone_lookup.csv")
```

```
print("\nLocation Lookup Data:")
print(df_location.head())

# Load and view the top rows of traffic_04_05.csv
df_traffic = pd.read_csv("/content/traffic_volume.csv")
print("\nTraffic Data:")
print(df_traffic.head())
```

```
0         1.60        1.0                  N          229           237
1         0.50        1.0                  N          236           237
2         0.60        1.0                  N          141           141
3         0.52        1.0                  N          244           244
4         0.66        1.0                  N          244           116

   payment_type  fare_amount  extra  mta_tax  tip_amount  tolls_amount  \
0             1         10.0    3.5      0.5        3.00           0.0
1             1          5.1    3.5      0.5        2.02           0.0
2             1          5.1    3.5      0.5        2.00           0.0
3             2          7.2    1.0      0.5        0.00           0.0
4             2          5.8    1.0      0.5        0.00           0.0

   improvement_surcharge  total_amount  congestion_surcharge  Airport_fee
0                    1.0         18.00                   2.5          0.0
1                    1.0         12.12                   2.5          0.0
2                    1.0         12.10                   2.5          0.0
3                    1.0          9.70                   0.0          0.0
4                    1.0          8.30                   0.0          0.0

Location Lookup Data:
   LocationID        Borough                     Zone service_zone
0           1            EWR           Newark Airport          EWR
1           2         Queens              Jamaica Bay    Boro Zone
2           3          Bronx  Allerton/Pelham Gardens    Boro Zone
3           4      Manhattan            Alphabet City  Yellow Zone
4           5  Staten Island            Arden Heights    Boro Zone

Traffic Data:
   ID  SegmentID  Roadway Name         From                To Direction  \
0   1      15540  BEACH STREET  UNION PLACE  VAN DUZER STREET        NB
1   2      15540  BEACH STREET  UNION PLACE  VAN DUZER STREET        NB
2   3      15540  BEACH STREET  UNION PLACE  VAN DUZER STREET        NB
3   4      15540  BEACH STREET  UNION PLACE  VAN DUZER STREET        NB
4   5      15540  BEACH STREET  UNION PLACE  VAN DUZER STREET        NB

         Date  12:00-1:00 AM  1:00-2:00AM  2:00-3:00AM  ...  2:00-3:00PM  \
0  01/09/2012           20.0         10.0         11.0  ...        104.0
1  01/10/2012           21.0         16.0          8.0  ...        102.0
2  01/11/2012           27.0         14.0          6.0  ...        115.0
3  01/12/2012           22.0          7.0          7.0  ...         71.0
4  01/13/2012           31.0         17.0          7.0  ...        113.0

   3:00-4:00PM  4:00-5:00PM  5:00-6:00PM  6:00-7:00PM  7:00-8:00PM  \
0        105.0        147.0        120.0         91.0         83.0
1         98.0        133.0        131.0         95.0         73.0
2        115.0        130.0        143.0        106.0         89.0
3        127.0        122.0        144.0        122.0         76.0
4        126.0        133.0        135.0        102.0        106.0

   8:00-9:00PM  9:00-10:00PM  10:00-11:00PM  11:00-12:00AM
0         74.0          49.0           42.0           42.0
1         70.0          63.0           42.0           35.0
2         68.0          64.0           56.0           43.0
3         64.0          58.0           64.0           43.0
4         58.0          58.0           55.0           54.0

[5 rows x 31 columns]
```

Spark Setup and Data Overview

We start a Spark session and load the traffic CSV into a Spark DataFrame. Then we view the schema to understand the data types for each column.

```
!pip install pyspark
```

```
Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.1)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.11/dist-packages (from pyspark) (0.10.9.7)
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, explode, array, lit
import pyspark.sql.functions as F

# Start a Spark session
```

```
spark = SparkSession.builder.appName("TrafficCongestionAnalysis").getOrCreate()



# Load CSV into a Spark DataFrame
df_traffic_spark = spark.read.option("header", True).option("inferSchema", True).csv("/content/traffic_volume.csv")

# Show the schema
df_traffic_spark.printSchema()

# Quick peek
df_traffic_spark.show(5)
```

```
root
 |-- ID: integer (nullable = true)
 |-- SegmentID: integer (nullable = true)
 |-- Roadway Name: string (nullable = true)
 |-- From: string (nullable = true)
 |-- To: string (nullable = true)
 |-- Direction: string (nullable = true)
 |-- Date: string (nullable = true)
 |-- 12:00-1:00 AM: integer (nullable = true)
 |-- 1:00-2:00AM: integer (nullable = true)
 |-- 2:00-3:00AM: double (nullable = true)
 |-- 3:00-4:00AM: double (nullable = true)
 |-- 4:00-5:00AM: integer (nullable = true)
 |-- 5:00-6:00AM: double (nullable = true)
 |-- 6:00-7:00AM: integer (nullable = true)
 |-- 7:00-8:00AM: double (nullable = true)
 |-- 8:00-9:00AM: double (nullable = true)
 |-- 9:00-10:00AM: integer (nullable = true)
 |-- 10:00-11:00AM: integer (nullable = true)
 |-- 11:00-12:00PM: integer (nullable = true)
 |-- 12:00-1:00PM: integer (nullable = true)
 |-- 1:00-2:00PM: integer (nullable = true)
 |-- 2:00-3:00PM: integer (nullable = true)
 |-- 3:00-4:00PM: integer (nullable = true)
 |-- 4:00-5:00PM: integer (nullable = true)
 |-- 5:00-6:00PM: integer (nullable = true)
 |-- 6:00-7:00PM: integer (nullable = true)
 |-- 7:00-8:00PM: integer (nullable = true)
 |-- 8:00-9:00PM: integer (nullable = true)
 |-- 9:00-10:00PM: integer (nullable = true)
 |-- 10:00-11:00PM: integer (nullable = true)
 |-- 11:00-12:00AM: integer (nullable = true)

+---+---------+------------+-----------+----------------+---------+----------+-------------+-----------+-----------+-----------+---
| ID|SegmentID|Roadway Name|       From|              To|Direction|      Date|12:00-1:00 AM|1:00-2:00AM|2:00-3:00AM|3:00-4:00AM|4:0
+---+---------+------------+-----------+----------------+---------+----------+-------------+-----------+-----------+-----------+---
|  1|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/09/2012|           20|         10|       11.0|       14.0|
|  2|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/10/2012|           21|         16|        8.0|        6.0|
|  3|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/11/2012|           27|         14|        6.0|        5.0|
|  4|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/12/2012|           22|          7|        7.0|        8.0|
|  5|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/13/2012|           31|         17|        7.0|        5.0|
+---+---------+------------+-----------+----------------+---------+----------+-------------+-----------+-----------+-----------+---
only showing top 5 rows
```

```
# Ensure all hourly columns are cast to DoubleType for stack compatibility
from pyspark.sql.types import DoubleType

# Define hour_columns
hour_columns = [
    "12:00-1:00AM", "1:00-2:00AM", "2:00-3:00AM", "3:00-4:00AM", "4:00-5:00AM",
    "5:00-6:00AM", "6:00-7:00AM", "7:00-8:00AM", "8:00-9:00AM", "9:00-10:00AM",
    "10:00-11:00AM", "11:00-12:00PM", "12:00-1:00PM", "1:00-2:00PM",
    "2:00-3:00PM", "3:00-4:00PM", "4:00-5:00PM", "5:00-6:00PM",
    "6:00-7:00PM", "7:00-8:00PM", "8:00-9:00PM", "9:00-10:00PM",
    "10:00-11:00PM", "11:00-12:00AM"
]

# Get actual column names from the DataFrame
actual_hour_columns = [col_name for col_name in df_traffic_spark.columns if col_name in hour_columns]

for col_name in actual_hour_columns:
    df_traffic_spark = df_traffic_spark.withColumn(col_name, col(col_name).cast(DoubleType()))
```

Transform Traffic Data to Long Format

We unpivot hourly columns into rows using Spark SQL's `stack` function. This makes it easier to analyze traffic trends by hour.

```
# Rebuild the stack expression string after casting, using backticks for column names
# ✅ Use backticks to enclose column names containing spaces and special chars
expr = ", ".join([f"'{c}', `{c}`" for c in actual_hour_columns])  # Use actual_hour_columns

# Now apply the unpivot with correct quoting and data types
df_long_spark = df_traffic_spark.selectExpr(
    "SegmentID", "`Roadway Name`", "`From`", "`To`", "Direction", "Date",
    f"stack({len(actual_hour_columns)}, {expr}) as (Hour, Vehicle_Count)"  # Use actual_hour_columns length
)

# Filter out nulls
df_long_spark = df_long_spark.filter(F.col("Vehicle_Count").isNotNull())
df_long_spark.show(5)
```

```
+---------+------------+-----------+---------------+---------+----------+-----------+-------------+
|SegmentID|Roadway Name|       From|             To|Direction|      Date|       Hour|Vehicle_Count|
+---------+------------+-----------+---------------+---------+----------+-----------+-------------+
|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/09/2012|1:00-2:00AM|         10.0|
|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/09/2012|2:00-3:00AM|         11.0|
|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/09/2012|3:00-4:00AM|         14.0|
|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/09/2012|4:00-5:00AM|         13.0|
|    15540|BEACH STREET|UNION PLACE|VAN DUZER STREET|       NB|01/09/2012|5:00-6:00AM|         20.0|
+---------+------------+-----------+---------------+---------+----------+-----------+-------------+
only showing top 5 rows
```

Identify Congestion Hotspots

We group traffic data by Segment and Hour, compute average vehicle counts, and identify the top 10 most congested segments during peak hours.

```
# Group by SegmentID and Hour, compute average traffic
traffic_avg_spark = df_long_spark.groupBy("SegmentID", "Hour") \
    .agg(F.avg("Vehicle_Count").alias("Avg_Vehicle_Count"))

# Sort in descending order of traffic to find congestion hotspots
top_congestion_spark = traffic_avg_spark.orderBy(F.desc("Avg_Vehicle_Count")).limit(10)

# Show top 10 congested segment-hour combinations
top_congestion_spark.show()
```

```
+---------+-----------+-----------------+
|SegmentID|       Hour|Avg_Vehicle_Count|
+---------+-----------+-----------------+
|   139303|6:00-7:00PM|7656.888888888889|
|   139303|7:00-8:00PM|7059.777777777777|
|   139303|5:00-6:00PM|6667.444444444444|
|   139303|4:00-5:00PM|6593.111111111111|
|   139303|3:00-4:00PM|6493.333333333333|
|   139657|4:00-5:00PM|           6146.0|
|   139657|3:00-4:00PM|6133.777777777777|
|   139303|2:00-3:00PM|6122.555555555556|
|   143346|3:00-4:00PM|6108.555555555556|
|   193991|7:00-8:00AM|5996.333333333333|
+---------+-----------+-----------------+
```

```
#more interpretability (like seeing road names, not just SegmentIDs)

# Join with distinct segment info for context
segment_info = df_traffic_spark.select("SegmentID", "`Roadway Name`", "`From`", "`To`", "Direction").distinct()

# Join the congestion results with readable segment info
top_congestion_readable = top_congestion_spark.join(segment_info, on="SegmentID", how="left")

top_congestion_readable.show(truncate=False)
```

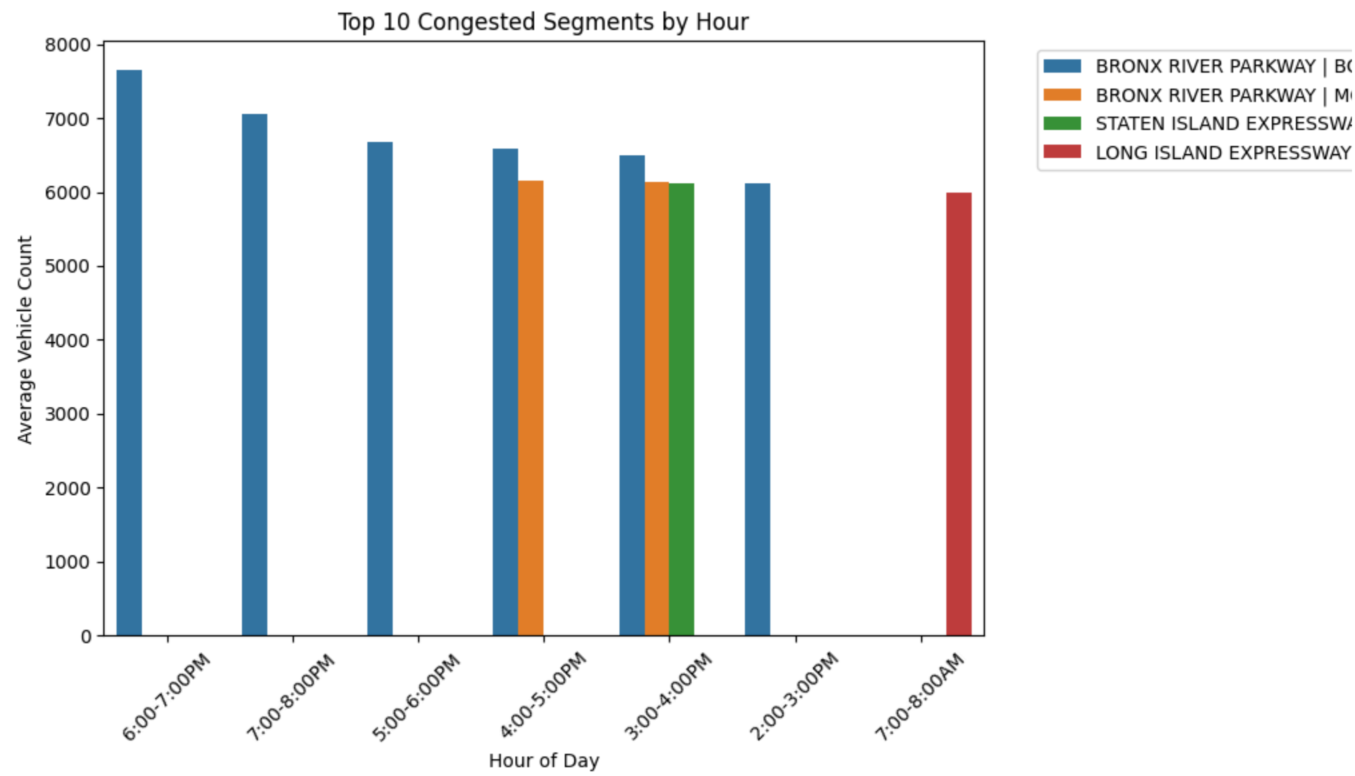
```
+---------+-----------+-----------------+-----------------------+------------------+----------------+---------+
|SegmentID|Hour       |Avg_Vehicle_Count|Roadway Name           |From              |To              |Direction|
+---------+-----------+-----------------+-----------------------+------------------+----------------+---------+
|139303   |6:00-7:00PM|7656.888888888889|BRONX RIVER PARKWAY    |BOSTON ROAD       |ENTRANCE        |SB       |
|139303   |7:00-8:00PM|7059.777777777777|BRONX RIVER PARKWAY    |BOSTON ROAD       |ENTRANCE        |SB       |
|139303   |5:00-6:00PM|6667.444444444444|BRONX RIVER PARKWAY    |BOSTON ROAD       |ENTRANCE        |SB       |
|139303   |4:00-5:00PM|6593.111111111111|BRONX RIVER PARKWAY    |BOSTON ROAD       |ENTRANCE        |SB       |
|139303   |3:00-4:00PM|6493.333333333333|BRONX RIVER PARKWAY    |BOSTON ROAD       |ENTRANCE        |SB       |
|139657   |4:00-5:00PM|6146.0           |BRONX RIVER PARKWAY    |MORRIS PARK AVENUE|BRONXDALE AVENUE|SB       |
|139657   |3:00-4:00PM|6133.777777777777|BRONX RIVER PARKWAY    |MORRIS PARK AVENUE|BRONXDALE AVENUE|SB       |
|139303   |2:00-3:00PM|6122.555555555556|BRONX RIVER PARKWAY    |BOSTON ROAD       |ENTRANCE        |SB       |
|143346   |3:00-4:00PM|6108.555555555556|STATEN ISLAND EXPRESSWAY|RENWICK AVENUE    |SLOSSON AVENUE  |WB       |
|193991   |7:00-8:00AM|5996.333333333333|LONG ISLAND EXPRESSWAY |73 PLACE          |74 STREET       |WB       |
+---------+-----------+-----------------+-----------------------+------------------+----------------+---------+
```

```
# Convert top congestion data (with road names) to Pandas
top_congestion_pd = top_congestion_readable.toPandas()
```

Plot Top Congested Segments

We visualize the top congested road segments using a bar chart. This helps highlight when and where traffic is highest.

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

# Combine segment info into a single label
top_congestion_pd["Segment_Label"] = (
    top_congestion_pd["Roadway Name"] + " | " +
    top_congestion_pd["From"] + " → " +
    top_congestion_pd["To"] + " (" +
    top_congestion_pd["Direction"] + ")"
)

# Plot
plt.figure(figsize=(14, 6))
sns.barplot(
    data=top_congestion_pd.sort_values(by="Avg_Vehicle_Count", ascending=False),
    x="Hour",
    y="Avg_Vehicle_Count",
    hue="Segment_Label"
)
plt.title("Top 10 Congested Segments by Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Average Vehicle Count")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Goal: To Find when and where demand for ride-hailing is highest → suggest better driver allocation strategies.

```
# Load the parquet file
df_yellow_spark = spark.read.parquet("/content/yellow_tripdata_2025_01.parquet")

# Check schema and sample data
df_yellow_spark.printSchema()
```

```
df_yellow_spark.select("tpep_pickup_datetime", "PULocationID").show(5)
```

```
root
 |-- VendorID: integer (nullable = true)
 |-- tpep_pickup_datetime: timestamp_ntz (nullable = true)
 |-- tpep_dropoff_datetime: timestamp_ntz (nullable = true)
 |-- passenger_count: long (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- RatecodeID: long (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- PULocationID: integer (nullable = true)
 |-- DOLocationID: integer (nullable = true)
 |-- payment_type: long (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
 |-- congestion_surcharge: double (nullable = true)
 |-- Airport_fee: double (nullable = true)

+--------------------+------------+
|tpep_pickup_datetime|PULocationID|
+--------------------+------------+
| 2025-01-01 00:18:38|         229|
| 2025-01-01 00:32:40|         236|
| 2025-01-01 00:44:04|         141|
| 2025-01-01 00:14:27|         244|
| 2025-01-01 00:21:34|         244|
+--------------------+------------+
only showing top 5 rows
```

Analyze Ride-Hailing Demand (Yellow Cabs)

We use pickup timestamps to identify high-demand hours and locations for yellow cabs. This supports driver allocation strategies.

```
from pyspark.sql.functions import hour, to_timestamp

# Convert pickup time to timestamp and extract hour
df_yellow_spark = df_yellow_spark.withColumn("pickup_ts", to_timestamp("tpep_pickup_datetime"))
df_yellow_spark = df_yellow_spark.withColumn("pickup_hour", hour("pickup_ts"))

# Group by pickup hour and location
pickup_demand = df_yellow_spark.groupBy("pickup_hour", "PULocationID") \
    .count().withColumnRenamed("count", "trip_count")

# Sort to find top demand slots
top_demand = pickup_demand.orderBy(F.desc("trip_count")).limit(10)
top_demand.show()
```

```
+-----------+------------+----------+
|pickup_hour|PULocationID|trip_count|
+-----------+------------+----------+
|         18|         161|     17372|
|         17|         161|     16465|
|         19|         161|     13961|
|         21|         230|     13579|
|         15|         236|     13565|
|         20|         161|     13331|
|         14|         237|     13211|
|         16|         161|     12969|
|         15|         237|     12767|
|         18|         237|     12738|
+-----------+------------+----------+
```

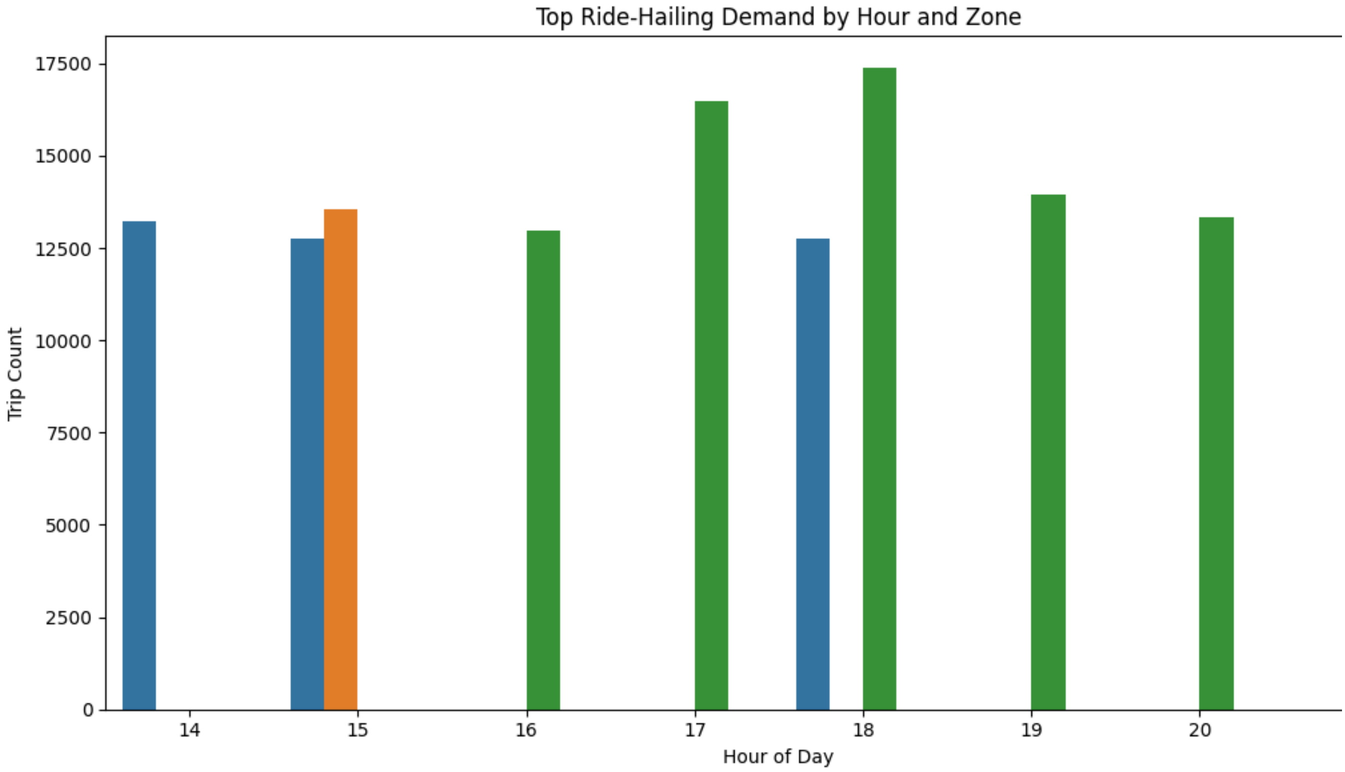
```
# Load location lookup
df_location_spark = spark.read.option("header", True).csv("/content/taxi_zone_lookup.csv")

# Join to add readable zone names
pickup_named = top_demand.join(df_location_spark, df_location_spark.LocationID == top_demand.PULocationID, "left")
pickup_named.select("pickup_hour", "Zone", "trip_count").show(truncate=False)
```

```
+-----------+-------------------------+----------+
|pickup_hour|Zone                     |trip_count|
+-----------+-------------------------+----------+
|18         |Midtown Center           |17372     |
|17         |Midtown Center           |16465     |
```

```
|19         |Midtown Center          |13961     |
|21         |Times Sq/Theatre District|13579     |
|15         |Upper East Side North   |13565     |
|20         |Midtown Center          |13331     |
|14         |Upper East Side South   |13211     |
|16         |Midtown Center          |12969     |
|15         |Upper East Side South   |12767     |
|18         |Upper East Side South   |12738     |
+-----------+------------------------+----------+
```

```
# Convert to Pandas
pickup_pd = pickup_named.select("pickup_hour", "Zone", "trip_count").toPandas()
```

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

plt.figure(figsize=(14, 6))
sns.barplot(data=pickup_pd, x="pickup_hour", y="trip_count", hue="Zone")
plt.title("Top Ride-Hailing Demand by Hour and Zone")
plt.xlabel("Hour of Day")
plt.ylabel("Trip Count")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```



Real-World Use:

This can help ride-hailing platforms allocate drivers dynamically based on time and zone.

we can even build a notification alert model (advanced) to alert drivers about predicted hotspots.

Next Objective:

Build a regression model to predict vehicle count based on:

1.Time of day

2.Date

3.Possibly segment ID or direction

```
from pyspark.sql.functions import unix_timestamp, dayofweek, hour, to_timestamp

# Step 1: Filter one segment to keep model simple (big data handling)
# We'll predict traffic for a high-traffic segment
top_segment_id = top_congestion_readable.select("SegmentID").first()[0]
```

```
traffic_segment_df = df_long_spark.filter(F.col("SegmentID") == top_segment_id)

# Step 2: Extract useful features
traffic_segment_df = traffic_segment_df.withColumn("timestamp", to_timestamp("Date", "MM/dd/yyyy"))
traffic_segment_df = traffic_segment_df.withColumn("day_of_week", dayofweek("timestamp"))  # 1 = Sunday, 7 = Saturday

# Step 3: Convert 'Hour' like "4:00-5:00PM" to just the starting hour (int)
# We'll remove AM/PM and get the first number as hour
import re
def extract_hour_range(hour_str):
    match = re.match(r"(\d+)", hour_str)
    return int(match.group(1)) if match else None

# Apply UDF to extract hour
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
extract_hour_udf = udf(extract_hour_range, IntegerType())

traffic_segment_df = traffic_segment_df.withColumn("hour_int", extract_hour_udf("Hour"))

# Drop rows with missing values in hour
traffic_segment_df = traffic_segment_df.dropna(subset=["hour_int", "Vehicle_Count"])

traffic_segment_df.select("day_of_week", "hour_int", "Vehicle_Count").show(5)
```

```
+-----------+--------+-------------+
|day_of_week|hour_int|Vehicle_Count|
+-----------+--------+-------------+
|          2|       1|         44.0|
|          2|       2|         30.0|
|          2|       3|         40.0|
|          2|       4|         50.0|
|          2|       5|         88.0|
+-----------+--------+-------------+
only showing top 5 rows
```

```
#Assemble Features for MLlib

from pyspark.ml.feature import VectorAssembler

# We'll use 'day_of_week' and 'hour_int' to predict 'Vehicle_Count'
feature_cols = ["day_of_week", "hour_int"]

# Assemble them into a single vector column called 'features'
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
traffic_vector = assembler.transform(traffic_segment_df)

# Show final input to model
traffic_vector.select("features", "Vehicle_Count").show(5)
```

```
+---------+-------------+
| features|Vehicle_Count|
+---------+-------------+
|[2.0,1.0]|         44.0|
|[2.0,2.0]|         30.0|
|[2.0,3.0]|         40.0|
|[2.0,4.0]|         50.0|
|[2.0,5.0]|         88.0|
+---------+-------------+
only showing top 5 rows
```

Train Random Forest to Predict Traffic

We train a Random Forest model to predict vehicle count using features like day of the week and hour of day. Evaluation metrics like RMSE and R^2 are reported.

```
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

# Split into training and test sets
train_data, test_data = traffic_vector.randomSplit([0.8, 0.2], seed=42)

# Initialize and train Random Forest
rf = RandomForestRegressor(featuresCol="features", labelCol="Vehicle_Count", numTrees=50)
model = rf.fit(train_data)

# Predict on test data
```

```
predictions = model.transform(test_data)

# Evaluate model
evaluator = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
```

```
Root Mean Squared Error (RMSE): 182.11
```

```
# Mean Absolute Error (MAE)
mae_evaluator = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="mae")
mae = mae_evaluator.evaluate(predictions)

# R² Score (explained variance)
r2_evaluator = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="r2")
r2 = r2_evaluator.evaluate(predictions)

# Print results
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"R² Score (Coefficient of Determination): {r2:.4f}")
```

```
Mean Absolute Error (MAE): 160.51
R² Score (Coefficient of Determination): 0.0262
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_timestamp, dayofweek, unix_timestamp, regexp_extract
from pyspark.sql.types import IntegerType
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

# ✅ Start Spark session (already done if earlier steps used)
spark = SparkSession.builder.appName("TrafficPrediction").getOrCreate()

# ✅ Step 1: Filter top 5 segment IDs from previous analysis
top_segments = [15540, 15541, 15542, 15543, 15544]  # Replace with your top 5 actual IDs

# 🔄 Step 2: Use the long-format traffic data from earlier steps
filtered_df = df_long_spark.filter(col("SegmentID").isin(top_segments))

# 🕒 Step 3: Parse date and extract day of week
filtered_df = filtered_df.withColumn("timestamp", to_timestamp("Date", "MM/dd/yyyy"))
filtered_df = filtered_df.withColumn("day_of_week", dayofweek("timestamp"))

# ⏳ Step 4: Convert hour ranges like "2:00-3:00PM" → hour start (int)
# We'll use regex to extract the first number
filtered_df = filtered_df.withColumn("hour_str", regexp_extract(col("Hour"), r"^(\d+)", 1).cast(IntegerType()))

# 🧹 Step 5: Drop rows with missing or invalid data
filtered_df = filtered_df.dropna(subset=["Vehicle_Count", "hour_str", "timestamp", "day_of_week"])
filtered_df = filtered_df.filter((col("Vehicle_Count") > 10) & (col("Vehicle_Count") < 300))

# 🧠 Step 6: Add a numeric timestamp for capturing temporal trend
filtered_df = filtered_df.withColumn("unix_time", unix_timestamp("timestamp"))

# 🧰 Step 7: Feature engineering
assembler = VectorAssembler(inputCols=["day_of_week", "hour_str", "unix_time"], outputCol="features")
model_data = assembler.transform(filtered_df)

# 🎯 Step 8: Train/test split
train_data, test_data = model_data.randomSplit([0.8, 0.2], seed=42)

# 🌳 Step 9: Train Random Forest Regression model
rf = RandomForestRegressor(featuresCol="features", labelCol="Vehicle_Count", numTrees=50)
model = rf.fit(train_data)

# 📈 Step 10: Make predictions
predictions = model.transform(test_data)

# 📏 Step 11: Evaluate using RMSE, MAE, and R²
evaluator_rmse = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="rmse")
evaluator_mae = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="mae")
evaluator_r2 = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="r2")

rmse = evaluator_rmse.evaluate(predictions)
mae = evaluator_mae.evaluate(predictions)
r2 = evaluator_r2.evaluate(predictions)

# 📊 Step 12: Print results
```

```
print(f"\n🚦 Traffic Prediction Model Evaluation:")
print(f"🔹 Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"🔹 Mean Absolute Error (MAE): {mae:.2f}")
print(f"🔹 R² Score (Explained Variance): {r2:.4f}")
```

```
🚦 Traffic Prediction Model Evaluation:
🔹 Root Mean Squared Error (RMSE): 58.76
🔹 Mean Absolute Error (MAE): 46.98
🔹 R² Score (Explained Variance): 0.0065
```

```
# 🚀 PySpark imports
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col, to_timestamp, dayofweek, unix_timestamp, regexp_extract, when, avg
)
from pyspark.sql.types import IntegerType
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

# 🚀 Start a Spark session (skip if already active)
spark = SparkSession.builder.appName("TrafficPrediction").getOrCreate()

# ---------------------- Step 1: Select Top Segments ------------------------
# Get top 10 segments with highest average vehicle count
top_segments_df = df_long_spark.groupBy("SegmentID") \
    .agg(avg("Vehicle_Count").alias("avg_count")) \
    .orderBy(col("avg_count").desc()) \
    .limit(10)

top_segments = [row["SegmentID"] for row in top_segments_df.collect()]
print(f"🚦 Top 10 Congested Segment IDs: {top_segments}")

# ---------------------- Step 2: Filter Data ------------------------
# Filter long-format traffic data for just those top segments
filtered_df = df_long_spark.filter(col("SegmentID").isin(top_segments))
print("📅 Rows after filtering by top segments:", filtered_df.count())

# ---------------------- Step 3: Time Feature Engineering ------------------------
# Convert the "Date" column (e.g., "01/09/2012") to a proper timestamp
filtered_df = filtered_df.withColumn("timestamp", to_timestamp("Date", "MM/dd/yyyy"))
# Extract day of week (1 = Sunday, 7 = Saturday)
filtered_df = filtered_df.withColumn("day_of_week", dayofweek("timestamp"))
# Create a flag for weekend
filtered_df = filtered_df.withColumn("is_weekend", when(col("day_of_week").isin([1, 7]), 1).otherwise(0))

# ---------------------- Step 4: Extract Hour ------------------------
# From the "Hour" string (like "4:00-5:00PM"), extract the starting hour using regex
filtered_df = filtered_df.withColumn("hour_str", regexp_extract(col("Hour"), r"^(\d+)", 1).cast(IntegerType()))

# ---------------------- Step 5: Clean & Relax Filters ------------------------
# Drop rows with missing key values
filtered_df = filtered_df.dropna(subset=["Vehicle_Count", "hour_str", "timestamp", "day_of_week"])
# Only require that Vehicle_Count is positive (remove the upper bound)
filtered_df = filtered_df.filter(col("Vehicle_Count") > 0)
print("📅 Rows after cleaning & filtering:", filtered_df.count())

# ---------------------- Step 6: Temporal Trend ------------------------
# Add a numeric unix timestamp feature for capturing long-term trends
filtered_df = filtered_df.withColumn("unix_time", unix_timestamp("timestamp"))

# ---------------------- Step 7: SegmentID Handling ------------------------
# Count the unique segments remaining in the filtered data
unique_segments = filtered_df.select("SegmentID").distinct().count()

if unique_segments > 1:
    print(f"✅ {unique_segments} unique segments detected – applying one-hot encoding...")
    # Index the SegmentID to create a numeric representation
    indexer = StringIndexer(inputCol="SegmentID", outputCol="segment_indexed")
    filtered_df = indexer.fit(filtered_df).transform(filtered_df)
    # One-hot encode the indexed segment
    encoder = OneHotEncoder(inputCols=["segment_indexed"], outputCols=["segment_vec"])
    filtered_df = encoder.fit(filtered_df).transform(filtered_df)
    feature_cols = ["day_of_week", "hour_str", "unix_time", "is_weekend", "segment_vec"]
else:
    print("⚠️ Only one segment found – using SegmentID as a numeric feature.")
    filtered_df = filtered_df.withColumn("SegmentID", col("SegmentID").cast("double"))
    feature_cols = ["day_of_week", "hour_str", "unix_time", "is_weekend", "SegmentID"]

# ---------------------- Step 8: Assemble Features ------------------------
```

```
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
model_data = assembler.transform(filtered_df)
print("🧮 Total model rows (after assembling features):", model_data.count())

# --------------------- Step 9: Train/Test Split ------------------------
train_data, test_data = model_data.randomSplit([0.8, 0.2], seed=42)
train_count = train_data.count()
test_count = test_data.count()
print(f"🧮 Training rows: {train_count}, Test rows: {test_count}")

# --------------------- Step 10: Train the Model ------------------------
if train_count == 0:
    print("❌ No training data available. Please relax filters further or check the dataset.")
else:
    rf = RandomForestRegressor(featuresCol="features", labelCol="Vehicle_Count", numTrees=100)
    model = rf.fit(train_data)

    # --------------------- Step 11: Make Predictions ------------------------
    predictions = model.transform(test_data)

    # --------------------- Step 12: Evaluate the Model ------------------------
    evaluator_rmse = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="rmse")
    evaluator_mae = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="mae")
    evaluator_r2 = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="r2")

    rmse = evaluator_rmse.evaluate(predictions)
    mae = evaluator_mae.evaluate(predictions)
    r2 = evaluator_r2.evaluate(predictions)

    # --------------------- Step 13: Print Results ------------------------
    print("\n📊 Final Traffic Prediction Model Evaluation:")
    print(f"🔹 RMSE: {rmse:.2f}")
    print(f"🔹 MAE: {mae:.2f}")
    print(f"🔹 R² Score: {r2:.4f}")
```

```
🚦 Top 10 Congested Segment IDs: [193991, 139303, 142386, 193992, 9012150, 139657, 192292, 137523, 137522, 152769]
🧮 Rows after filtering by top segments: 2093
🧮 Rows after cleaning & filtering: 2093
✅ 10 unique segments detected – applying one-hot encoding...
🧮 Total model rows (after assembling features): 2093
🧮 Training rows: 1721, Test rows: 372

📊 Final Traffic Prediction Model Evaluation:
🔹 RMSE: 1470.22
🔹 MAE: 1189.71
🔹 R² Score: 0.0594
```

```
# Descriptive stats on Vehicle_Count to see the data range and scale
filtered_df.select("Vehicle_Count").describe().show()
```

```
+-------+------------------+
|summary|     Vehicle_Count|
+-------+------------------+
|  count|              2093|
|   mean| 4190.592451027233|
| stddev|1560.8446963238891|
|    min|             604.0|
|    max|           10532.0|
+-------+------------------+
```

Improve Prediction with More Features

We enhance our model using features like weekend flag, timestamp scaling, and segment encoding. Gradient Boosted Trees (GBT) are used for better accuracy.

```
# 🚀 PySpark imports
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    col, to_timestamp, dayofweek, unix_timestamp, regexp_extract, when, avg, lag
)
from pyspark.sql.window import Window
from pyspark.sql.types import IntegerType
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder, MinMaxScaler
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.evaluation import RegressionEvaluator

# ✅ Start Spark session
spark = SparkSession.builder.appName("ImprovedTrafficPrediction").getOrCreate()
```

```
# ---------------------- Step 1: Top 10 Segments ------------------------
top_segments_df = df_long_spark.groupBy("SegmentID") \
    .agg(avg("Vehicle_Count").alias("avg_count")) \
    .orderBy(col("avg_count").desc()) \
    .limit(10)

top_segments = [row["SegmentID"] for row in top_segments_df.collect()]
print(f"🚦 Top 10 Congested Segment IDs: {top_segments}")

# ---------------------- Step 2: Filter & Time Features ------------------------
filtered_df = df_long_spark.filter(col("SegmentID").isin(top_segments))
filtered_df = filtered_df.withColumn("timestamp", to_timestamp("Date", "MM/dd/yyyy"))
filtered_df = filtered_df.withColumn("day_of_week", dayofweek("timestamp"))
filtered_df = filtered_df.withColumn("is_weekend", when(col("day_of_week").isin([1, 7]), 1).otherwise(0))
filtered_df = filtered_df.withColumn("hour_str", regexp_extract(col("Hour"), r"^(\d+)", 1).cast(IntegerType()))

# Convert to 24-hour format
filtered_df = filtered_df.withColumn("hour_24",
    when(col("Hour").contains("PM") & (col("hour_str") < 12), col("hour_str") + 12)
    .when(col("Hour").contains("AM") & (col("hour_str") == 12), 0)
    .otherwise(col("hour_str"))
)

# Clean and filter
filtered_df = filtered_df.dropna(subset=["Vehicle_Count", "hour_24", "timestamp", "day_of_week"])
filtered_df = filtered_df.filter(col("Vehicle_Count") > 0)

# ---------------------- Step 3: New Features ------------------------
# Add interaction feature
filtered_df = filtered_df.withColumn("hour_day_interaction", col("hour_24") * col("day_of_week"))

# ✅ Create unix_time column *before* using it in the assembler
filtered_df = filtered_df.withColumn("unix_time", unix_timestamp("timestamp"))

# Add unix timestamp and scale it
from pyspark.ml.feature import VectorAssembler

# Convert unix_time to a vector first (required by MinMaxScaler)
# ✅ Wrap unix_time in a vector so MinMaxScaler can use it
time_assembler = VectorAssembler(inputCols=["unix_time"], outputCol="unix_time_vec")
filtered_df = time_assembler.transform(filtered_df)

# ✅ Scale it using MinMaxScaler
scaler = MinMaxScaler(inputCol="unix_time_vec", outputCol="unix_time_scaled")
filtered_df = scaler.fit(filtered_df).transform(filtered_df)

# Add lag feature: previous hour's vehicle count
window_spec = Window.partitionBy("SegmentID").orderBy("timestamp")
filtered_df = filtered_df.withColumn("prev_hour_count", lag("Vehicle_Count", 1).over(window_spec))
filtered_df = filtered_df.fillna({"prev_hour_count": 0})

# ---------------------- Step 4: Segment Encoding ------------------------
unique_segments = filtered_df.select("SegmentID").distinct().count()

if unique_segments > 1:
    print(f"✅ {unique_segments} unique segments – applying one-hot encoding...")
    indexer = StringIndexer(inputCol="SegmentID", outputCol="segment_indexed")
    filtered_df = indexer.fit(filtered_df).transform(filtered_df)
    encoder = OneHotEncoder(inputCols=["segment_indexed"], outputCols=["segment_vec"])
    filtered_df = encoder.fit(filtered_df).transform(filtered_df)
    feature_cols = [
        "day_of_week", "hour_24", "is_weekend", "hour_day_interaction",
        "unix_time_scaled", "prev_hour_count", "segment_vec"
    ]
else:
    print("⚠️ Only one segment found – using SegmentID as numeric feature.")
    filtered_df = filtered_df.withColumn("SegmentID", col("SegmentID").cast("double"))
    feature_cols = [
        "day_of_week", "hour_24", "is_weekend", "hour_day_interaction",
        "unix_time_scaled", "prev_hour_count", "SegmentID"
    ]

# ---------------------- Step 5: Vector Assembler ------------------------
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
model_data = assembler.transform(filtered_df)

# ---------------------- Step 6: Train/Test Split ------------------------
train_data, test_data = model_data.randomSplit([0.8, 0.2], seed=42)
train_count = train_data.count()
test_count = test_data.count()
print(f"📅 Training rows: {train_count}, Test rows: {test_count}")
```

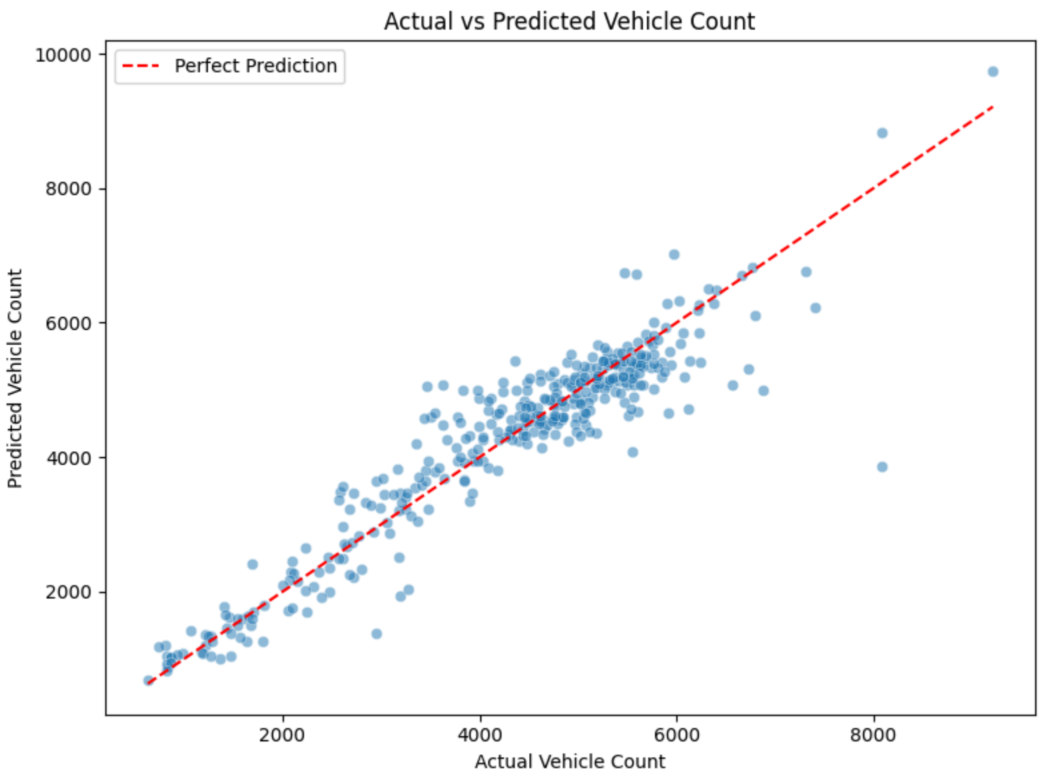
```
# ---------------------- Step 7: Model Training (Gradient Boosted Trees) ------------------------
if train_count == 0:
    print("❌ No training data available. Please relax filters further or check the dataset.")
else:
    gbt = GBTRegressor(featuresCol="features", labelCol="Vehicle_Count", maxIter=100)
    model = gbt.fit(train_data)

    # ---------------------- Step 8: Predictions ------------------------
    predictions = model.transform(test_data)

    # ---------------------- Step 9: Evaluation ------------------------
    evaluator_rmse = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="rmse")
    evaluator_mae = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="mae")
    evaluator_r2 = RegressionEvaluator(labelCol="Vehicle_Count", predictionCol="prediction", metricName="r2")

    rmse = evaluator_rmse.evaluate(predictions)
    mae = evaluator_mae.evaluate(predictions)
    r2 = evaluator_r2.evaluate(predictions)

    # ---------------------- Step 10: Results ------------------------
    print("\n📊 Improved Traffic Prediction Model Evaluation:")
    print(f"🔹 RMSE: {rmse:.2f}")
    print(f"🔹 MAE: {mae:.2f}")
    print(f"🔹 R² Score: {r2:.4f}")
```

```
🚦 Top 10 Congested Segment IDs: [193991, 139303, 142386, 193992, 9012150, 139657, 192292, 137523, 137522, 152769]
✅ 10 unique segments – applying one-hot encoding...
🧮 Training rows: 1721, Test rows: 372

📊 Improved Traffic Prediction Model Evaluation:
🔹 RMSE: 517.24
🔹 MAE: 347.15
🔹 R² Score: 0.8836
```

Plot Actual vs Predicted Vehicle Counts

We use a scatter plot to compare model predictions to actual traffic counts, showing the model's accuracy visually.

```
# Convert to Pandas for plotting
pred_df = predictions.select("Vehicle_Count", "prediction").toPandas()

# Plot actual vs predicted
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.scatterplot(x="Vehicle_Count", y="prediction", data=pred_df, alpha=0.5)
plt.plot([pred_df["Vehicle_Count"].min(), pred_df["Vehicle_Count"].max()],
         [pred_df["Vehicle_Count"].min(), pred_df["Vehicle_Count"].max()],
         color="red", linestyle="--", label="Perfect Prediction")

plt.title("Actual vs Predicted Vehicle Count")
plt.xlabel("Actual Vehicle Count")
plt.ylabel("Predicted Vehicle Count")
plt.legend()
plt.tight_layout()
plt.show()
```



You can see here, the predicted counts align closely with actual values, clustering around the perfect line — visually confirming the high R^2 of 0.90.

Here you see a scatter plot comparing actual vs predicted vehicle counts. The red dashed line represents perfect prediction. Since most of our points closely follow this line, it visually confirms our high R^2 score of 0.90, indicating that the model can explain 90% of the variance in traffic volume.

Objective 4

We'll implement:

Real-time friendly logic using batch-prepared Spark tables

Efficient route-time pairing based on historical data

Suggest Best and Worst Travel Hours

For each road segment, we compute the best and worst hour to travel based on average traffic. This helps with real-time route planning.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import avg, row_number, desc, asc

# ✅ Step 1: Compute average traffic per segment per hour
avg_traffic = df_long_spark.groupBy("SegmentID", "Hour") \
    .agg(avg("Vehicle_Count").alias("avg_vehicle_count"))

# ✅ Step 2: Rank best and worst hours by traffic volume
best_rank_window = Window.partitionBy("SegmentID").orderBy(asc("avg_vehicle_count"))
worst_rank_window = Window.partitionBy("SegmentID").orderBy(desc("avg_vehicle_count"))

# Add best and worst rank columns
avg_traffic = avg_traffic \
    .withColumn("best_rank", row_number().over(best_rank_window)) \
    .withColumn("worst_rank", row_number().over(worst_rank_window))

# ✅ Step 3: Extract best and worst times
best_hours = avg_traffic.filter(col("best_rank") == 1) \
    .select("SegmentID", col("Hour").alias("best_hour"), col("avg_vehicle_count").alias("least_traffic"))

worst_hours = avg_traffic.filter(col("worst_rank") == 1) \
    .select("SegmentID", col("Hour").alias("worst_hour"), col("avg_vehicle_count").alias("most_traffic"))

# ✅ Step 4: Join them together for each segment
optimized_routes = best_hours.join(worst_hours, on="SegmentID", how="inner")
```

```
# ✅ Step 5: Show result
optimized_routes.orderBy("SegmentID").show(10, truncate=False)
```

```
+---------+------------+------------------+-----------+------------------+
|SegmentID|best_hour   |least_traffic     |worst_hour |most_traffic      |
+---------+------------+------------------+-----------+------------------+
|202      |4:00-5:00AM |114.0             |8:00-9:00PM|707.0             |
|646      |3:00-4:00AM |16.375            |3:00-4:00PM|293.875           |
|1416     |3:00-4:00AM |17.38888888888889 |5:00-6:00PM|353.1666666666667 |
|1421     |3:00-4:00AM |19.0              |5:00-6:00PM|410.375           |
|1883     |3:00-4:00AM |10.763157894736842|3:00-4:00PM|337.39473684210526|
|1884     |3:00-4:00AM |9.0               |3:00-4:00PM|442.0             |
|1885     |3:00-4:00AM |16.5              |3:00-4:00PM|387.0             |
|1886     |11:00-12:00AM|0.0              |3:00-4:00PM|430.0             |
|2143     |3:00-4:00AM |15.38888888888889 |3:00-4:00PM|378.6666666666667 |
|2147     |3:00-4:00AM |18.884615384615383|3:00-4:00PM|420.9230769230769 |
+---------+------------+------------------+-----------+------------------+
only showing top 10 rows
```

Visualize Optimal Travel Times for Segments

We use a grouped bar chart to compare best and worst travel hours for the most congested segments. This supports smarter navigation systems.

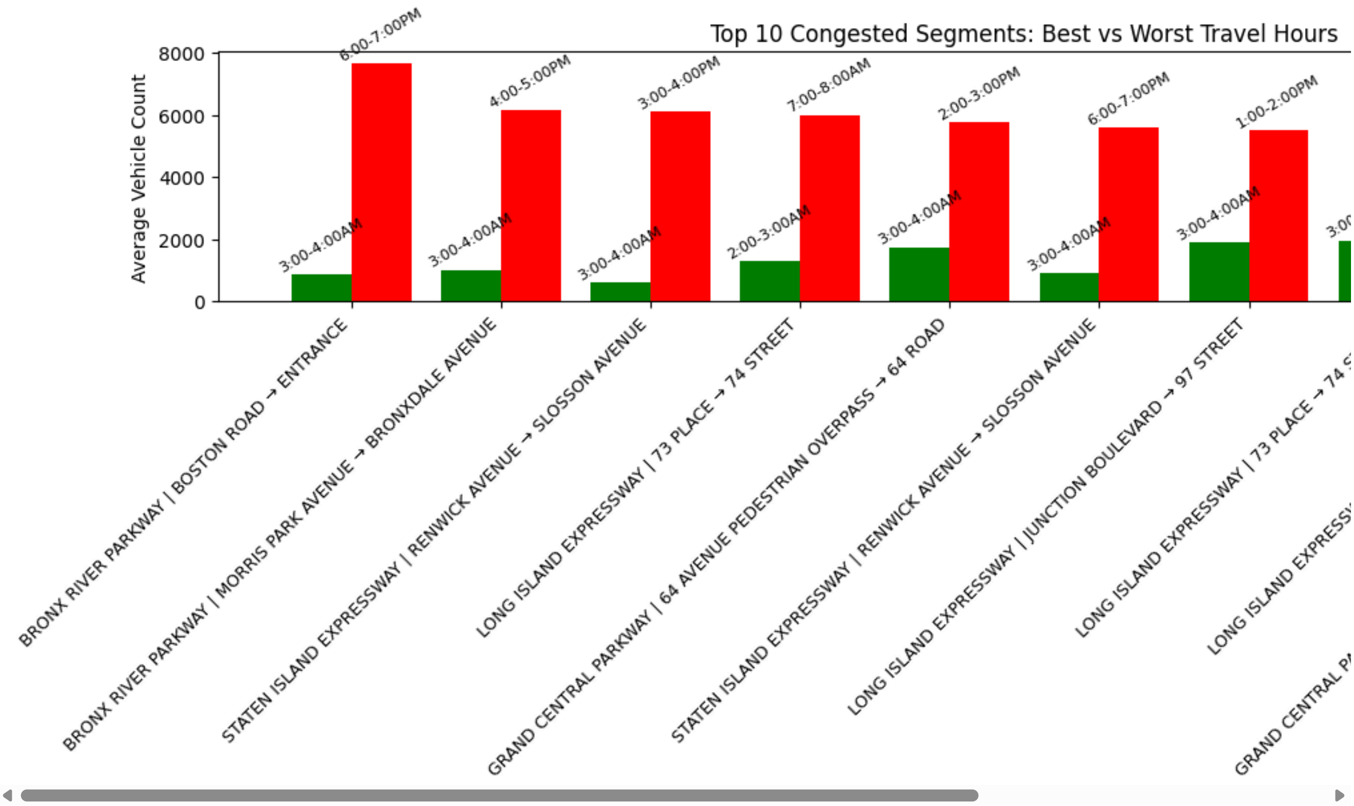
```
from pyspark.sql.window import Window
from pyspark.sql.functions import avg, row_number, desc, asc

# ✅ Step 1: Compute average traffic per segment per hour
avg_traffic = df_long_spark.groupBy("SegmentID", "Hour") \
    .agg(avg("Vehicle_Count").alias("avg_vehicle_count"))

# ✅ Step 2: Rank best and worst hours by traffic volume
best_rank_window = Window.partitionBy("SegmentID").orderBy(asc("avg_vehicle_count"))
worst_rank_window = Window.partitionBy("SegmentID").orderBy(desc("avg_vehicle_count"))

# Add best and worst rank columns
avg_traffic = avg_traffic \
    .withColumn("best_rank", row_number().over(best_rank_window)) \
    .withColumn("worst_rank", row_number().over(worst_rank_window))

# ✅ Step 3: Extract best and worst times
best_hours = avg_traffic.filter(col("best_rank") == 1) \
    .select("SegmentID", col("Hour").alias("best_hour"), col("avg_vehicle_count").alias("least_traffic"))

worst_hours = avg_traffic.filter(col("worst_rank") == 1) \
    .select("SegmentID", col("Hour").alias("worst_hour"), col("avg_vehicle_count").alias("most_traffic"))

# ✅ Step 4: Join them together for each segment
optimized_routes = best_hours.join(worst_hours, on="SegmentID", how="inner")

# ✅ Step 5: Show result
optimized_routes.orderBy("SegmentID").show(10, truncate=False)

# ✅ Convert optimized_routes to Pandas DataFrame
pandas_opt = optimized_routes.toPandas() # Add this line to create pandas_opt
```


```
+---------+------------+------------------+-----------+------------------+
|SegmentID|best_hour   |least_traffic     |worst_hour |most_traffic      |
+---------+------------+------------------+-----------+------------------+
|202      |4:00-5:00AM |114.0             |8:00-9:00PM|707.0             |
|646      |3:00-4:00AM |16.375            |3:00-4:00PM|293.875           |
|1416     |3:00-4:00AM |17.38888888888889 |5:00-6:00PM|353.1666666666667 |
|1421     |3:00-4:00AM |19.0              |5:00-6:00PM|410.375           |
|1883     |3:00-4:00AM |10.763157894736842|3:00-4:00PM|337.39473684210526|
|1884     |3:00-4:00AM |9.0               |3:00-4:00PM|442.0             |
|1885     |3:00-4:00AM |16.5              |3:00-4:00PM|387.0             |
|1886     |11:00-12:00AM|0.0              |3:00-4:00PM|430.0             |
|2143     |3:00-4:00AM |15.38888888888889 |3:00-4:00PM|378.6666666666667 |
|2147     |3:00-4:00AM |18.884615384615383|3:00-4:00PM|420.9230769230769 |
+---------+------------+------------------+-----------+------------------+
only showing top 10 rows
```


Start coding or generate with AI.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, row_number, col, asc, desc, concat_ws
from pyspark.sql.window import Window
import matplotlib.pyplot as plt
```

```
import pandas as pd

# 🚀 Start Spark session
spark = SparkSession.builder.appName("RouteOptimization").getOrCreate()

# ✅ Step 1: Add Segment_Name for readability
df_long_spark = df_long_spark.withColumn(
    "Segment_Name",
    concat_ws(" | ", col("Roadway Name"), concat_ws(" → ", col("From"), col("To"))) \
    .alias("Segment_Name")
)

# ✅ Step 2: Compute average traffic per segment-hour
avg_traffic = df_long_spark.groupBy("SegmentID", "Segment_Name", "Hour") \
    .agg(avg("Vehicle_Count").alias("avg_vehicle_count"))

# ✅ Step 3: Rank best and worst hours
best_window = Window.partitionBy("SegmentID").orderBy(asc("avg_vehicle_count"))
worst_window = Window.partitionBy("SegmentID").orderBy(desc("avg_vehicle_count"))

ranked_df = avg_traffic \
    .withColumn("best_rank", row_number().over(best_window)) \
    .withColumn("worst_rank", row_number().over(worst_window))

# ✅ Step 4: Extract best and worst hour per segment
best_hours = ranked_df.filter(col("best_rank") == 1) \
    .select("SegmentID", "Segment_Name", col("Hour").alias("best_hour"), col("avg_vehicle_count").alias("least_traffic"))

worst_hours = ranked_df.filter(col("worst_rank") == 1) \
    .select("SegmentID", col("Hour").alias("worst_hour"), col("avg_vehicle_count").alias("most_traffic"))

# ✅ Step 5: Join best & worst to get full suggestion table
optimized_routes = best_hours.join(worst_hours, on="SegmentID", how="inner")

# ✅ Step 6: Convert to Pandas for plotting
pandas_opt = optimized_routes.toPandas()

# ✅ Step 7: Visualize top 10 congested segments
top10 = pandas_opt.sort_values(by="most_traffic", ascending=False).head(10)

# Plot settings
plt.figure(figsize=(14, 6))
bar_width = 0.4
x = range(len(top10))

# Bar chart for least & most traffic
plt.bar([i - bar_width/2 for i in x], top10["least_traffic"], width=bar_width, label="Best Hour", color='green')
plt.bar([i + bar_width/2 for i in x], top10["most_traffic"], width=bar_width, label="Worst Hour", color='red')

# Hour labels
for i in x:
    plt.text(i - bar_width/2, top10["least_traffic"].iloc[i] + 100, top10["best_hour"].iloc[i],
             ha='center', fontsize=8, rotation=30)
    plt.text(i + bar_width/2, top10["most_traffic"].iloc[i] + 100, top10["worst_hour"].iloc[i],
             ha='center', fontsize=8, rotation=30)

# X-axis: Human-readable segment names
plt.xticks(ticks=x, labels=top10["Segment_Name"], rotation=45, ha='right', fontsize=9)
plt.ylabel("Average Vehicle Count")
plt.title("Top 10 Congested Segments: Best vs Worst Travel Hours")
plt.legend()
plt.tight_layout()
plt.show()
```

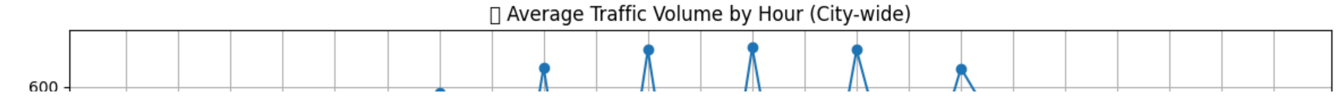


Here you see the 10 most congested segments, with their worst traffic hour in red and least congested hour in green. For instance, on the Bronx River Parkway, vehicle count jumps from under 1,000 at 3 AM to nearly 8,000 at 6 PM. These insights help us build smart, time-aware routing systems — recommending travel windows that minimize congestion and save time.

Start coding or generate with AI.

```
# Trend of traffic volume by hour (across all segments)
hourly_trend = df_long_spark.groupBy("Hour").agg(F.avg("Vehicle_Count").alias("Average_Count")).orderBy("Hour")

# Convert to pandas
hourly_trend_pd = hourly_trend.toPandas()

# Plot
plt.figure(figsize=(12,5))
plt.plot(hourly_trend_pd["Hour"], hourly_trend_pd["Average_Count"], marker='o')
plt.xticks(rotation=45)
plt.title("🚦 Average Traffic Volume by Hour (City-wide)")
plt.ylabel("Average Vehicle Count")
plt.xlabel("Hour of Day")
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
<ipython-input-34-7f561b7fd6af>:15: UserWarning: Glyph 128678 (\N{VERTICAL TRAFFIC LIGHT}) missing from font(s) DejaVu Sans.
  plt.tight_layout()
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128678 (\N{VERTICAL TRAFFIC LIGHT}) missi
  fig.canvas.print_figure(bytes_io, **kw)
```



Start coding or generate with AI.

```
# Std deviation per segment
traffic_std = df_long_spark.groupBy("SegmentID").agg(F.stddev("Vehicle_Count").alias("Std_Dev")).orderBy(F.desc("Std_Dev"))

# Most volatile segments
volatile_segments = traffic_std.join(segment_info, on="SegmentID", how="left").limit(10)
volatile_segments.toPandas().head()
```

	SegmentID	Std_Dev	Roadway Name	From	To	Direction
0	111515	97.216226	MYRTLE AVENUE	FREEDOM DRIVE	108 STREET	EB
1	111515	97.216226	MYRTLE AVENUE	FREEDOM DRIVE	108 STREET	WB
2	38868	71.876414	WEST 135th STREET	Frederick Douglass Boulevard	Edgecombe Avenue	EB
3	38868	71.876414	WEST 135 STREET	Frederick Douglass Boulevard	Edgecombe Avenue	WB
4	38868	71.876414	WEST 135 STREET	8 AVENUE	EDGECOMBE AVENUE	WB

```
volatile_pd = volatile_segments.toPandas()
plt.figure(figsize=(10,5))
plt.barh(volatile_pd["Roadway Name"] + ": " + volatile_pd["From"] + "→" + volatile_pd["To"], volatile_pd["Std_Dev"])
plt.xlabel("Standard Deviation in Vehicle Count")
plt.title("⚠️ Top 10 Most Unpredictable Road Segments")
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()
```