BIA 678 — Big Data Technologies December 19, 2024

Big Data Project Report

Professor: Venu Guntupalli *Report By: Group G*

Team Members:

- Tharun Achanta
- Abhishek Kumar
- Leena Goyal
- Abhilasha Mehra

Contents

1 Project Details - NYC Citi Bike Hourly Demand Prediction

1.1 Problem Statement

The Citi Bike program struggles with an ongoing imbalance in bike and dock availability across its network, leading to frequent user frustration when stations lack available bikes or open docks—especially during peak commuting hours. This imbalance disrupts the user experience, reduces satisfaction, and challenges the program's operational efficiency.

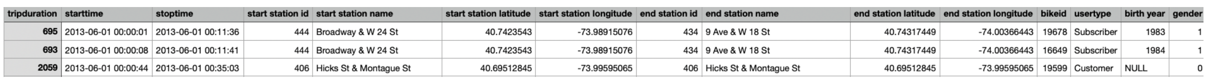
1.2 Objective

Our project aims to design a machine learning solution that accurately predicts bike and dock demand at each station. By forecasting demand based on factors such as time of day, weather, and local events, our solution helps Citi Bike optimize resource allocation and planning. The ultimate goal is to improve operational efficiency and provide a more reliable experience for riders.

2 Dataset

The datasets used for this project are as follows:

2.1 Citi Bike Historical Data



tripduration	starttime	stoptime	start station id	start station name	start station latitude	start station longitude	end station id	end station name	end station latitude	end station longitude	bikeid	usertype	birth year	gender
695	2013-06-01 00:00:01	2013-06-01 00:11:36	444	Broadway & W 24 St	40.7423543	-73.98915076	434	9 Ave & W 18 St	40.74317449	-74.00366443	19678	Subscriber	1983	1
693	2013-06-01 00:00:08	2013-06-01 00:11:41	444	Broadway & W 24 St	40.7423543	-73.98915076	434	9 Ave & W 18 St	40.74317449	-74.00366443	16649	Subscriber	1984	1
2059	2013-06-01 00:00:44	2013-06-01 00:35:03	406	Hicks St & Montague St	40.69512845	-73.99595065	406	Hicks St & Montague St	40.69512845	-73.99595065	19599	Customer	NULL	0

Figure 1: Caption describing the image.

The Citi Bike historical data provides detailed information about bike-sharing trips, including:

- Start time and stop time of each trip.
- Start and stop station names and IDs.
- Usage metrics, such as trip duration and bike IDs.

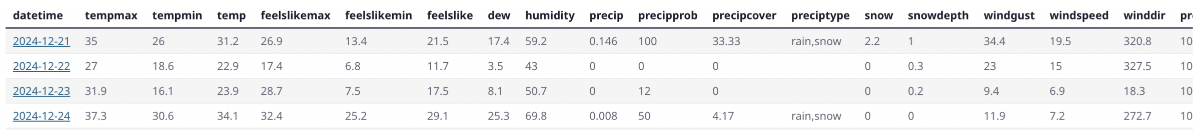
This dataset forms the foundation for demand prediction and allows us to analyze usage patterns over time. The data was sourced from *Citi Bike's system data portal* (`https://citibikenyc.com/system-data`).

2.2 Weather Data

Weather data is crucial for understanding external factors influencing bike demand. This dataset includes hourly weather information, such as:

- Temperature.
- Humidity.
- Precipitation.
- Windspeed.
- Visibility.

The weather data was sourced from *Visual Crossing's weather platform* (`https://www.visualcrossing.com/`), and it was integrated with the Citi Bike data to enhance feature engineering for predictive modeling.



datetime	tempmax	tempmin	temp	feelslikemax	feelslikemin	feelslike	dew	humidity	precip	precipprob	precipcover	preciptype	snow	snowdepth	windgust	windspeed	winddir	pr
2024-12-21	35	26	31.2	26.9	13.4	21.5	17.4	59.2	0.146	100	33.33	rain,snow	2.2	1	34.4	19.5	320.8	10
2024-12-22	27	18.6	22.9	17.4	6.8	11.7	3.5	43	0	0	0		0	0.3	23	15	327.5	10
2024-12-23	31.9	16.1	23.9	28.7	7.5	17.5	8.1	50.7	0	12	0		0	0.2	9.4	6.9	18.3	10
2024-12-24	37.3	30.6	34.1	32.4	25.2	29.1	25.3	69.8	0.008	50	4.17	rain,snow	0	0	11.9	7.2	272.7	10

Figure 2: Caption describing the image.

3 Pipeline-1 Loading Raw Data:

In this part of the project, we worked with approximately 320 CSV files containing bike-sharing data from January 2013 to December 2024. These files were uploaded to Google Cloud Storage (GCS), and we checked for schema changes across the datasets to make them consistent.

3.1 Data Upload to GCS

The files were named according to the year and month (e.g., `201301.csv`, `201302.csv`). They were stored in the GCS bucket at the following path:

`gs://bucket121024/csv`

3.2 Detecting Schema Changes

We used Apache Spark to detect schema changes programmatically. Below are the steps we followed:

3.2.1 Generate File Paths

We created a list of file paths for each year and month using Python. The paths pointed to the CSV files in the GCS bucket.

```python
# Generate file paths dynamically
years = range(2013, 2025)
months = range(1, 13)
dates = [f"{year}{str(month).zfill(2)}" for year in years for month in months]
```

3.2.2 Load and Analyze Schemas

Each file was loaded into a Spark DataFrame, and the schema was extracted using a Python function.

```python
# Function to extract schema
def get_schema(df):
    return [f"{field.name}:{field.dataType}" for field in df.schema.fields]

# Load files and analyze schemas
for date in dates:
    file_path = os.path.join(gcs_path, f"{date}*.csv")
    try:
        df = spark.read.option("header", "true").csv(file_path)
        schemas[date] = get_schema(df)
    except Exception as e:
        print(f"No data or error processing {date}: {str(e)}")
```

3.2.3 Compare Schemas Month by Month

We compared schemas month by month to find differences. Any schema change was logged for further processing.

```python
# Compare schemas month-over-month
previous_schema = None
for date, schema in sorted(schemas.items()):
    if previous_schema and schema != previous_schema:
        print(f"Schema change detected in {date}")
        print(f"Previous Schema ({previous_date}): {previous_schema}")
        print(f"Current Schema ({date}): {schema}")
    previous_schema = schema
    previous_date = date
```

3.3 Summary of Results

We found schema changes at specific months. These changes were logged to help standardize the data. Below is an example of the logged schema changes:

```
-------> Schema change detected in 201610
-------> Schema change detected in 201704
-------> Schema change detected in 202001
-------> Schema change detected in 202407
-------> Schema change detected in 202408
-------> Schema change detected in 202409
```

3.4 Performance

The schema comparison process took around **1 minute and 25 seconds**. This task was distributed across **5 workers** to handle large datasets efficiently.

4 Pipeline-1 Schema Standardization and Data Integration

We standardized column names across different datasets to make the data consistent. Then, we combined all the datasets into one DataFrame.

4.1 Standardizing Column Names

The datasets had inconsistent column names because of schema changes over time. We created a `target_schema` with all the required column names and used a dictionary to rename columns in each dataset.

4.1.1 Steps Involved

- **Renaming Columns:** We used a dictionary to map old column names (e.g., `Start Time`, `started_at`) to new ones (`starttime`). Columns were renamed using `withColumnRenamed`.
- **Adding Missing Columns:** Any column missing in a dataset was added with a default value of `NULL`.
- **Reordering Columns:** Columns were reordered to match the `target_schema`.

```
for original_col, target_col in column_mapping.items():
    if original_col in df.columns:
        df = df.withColumnRenamed(original_col, target_col)

for col in target_schema:
    if col not in df.columns:
        df = df.withColumn(col, F.lit(None))

df = df.select(target_schema)
```

4.2 Combining DataFrames

After standardizing the datasets, we combined them into one DataFrame.

```
standardized_dfs = [standardize_schema(file, target_schema) for file in dates]
standardized_dfs = [df for df in standardized_dfs if df is not None]

if standardized_dfs:
    final_df = standardized_dfs[0]
    for df in standardized_dfs[1:]:
        final_df = final_df.unionByName(df)
    final_df.show()
else:
    print("No valid data found.")
```

5 Pipeline-1 Results

5.1 Row Count and Null Value Analysis

The final DataFrame contained **268,834,357 rows**. A null value analysis showed missing values in some columns, as summarized below:

Column Name	Null Values Count
bikeid	153,726,709
birthyear	158,708,540
starttime	0
startstationname	50,286
...	...

Table 1: Null Values Count for Each Column

5.2 Saving the Processed Data

The final DataFrame was saved to GCS in `parquet` format:

```
output_path = "gs://bucket121024/pipeline1/final_output.parquet"
final_df.write.mode("overwrite").parquet(output_path)
```

This standardized dataset will be used in the next steps of the project.

6 Pipeline-2: Cleaning, Merging Data, and Feature Engineering

In this pipeline, we cleaned, merged, and standardized data from the output of Pipeline-1. This process included handling inconsistent date formats, parsing dates into a unified format, and preparing the data for feature engineering.

6.1 Input Data Schema

The input data for Pipeline-2 was read from the output of Pipeline-1, saved in `parquet` format. Below is the schema of the input data:

```
root
 |-- bikeid: string (nullable = true)
 |-- birthyear: string (nullable = true)
 |-- endlatitude: string (nullable = true)
 |-- endlongitude: string (nullable = true)
 |-- endstationid: string (nullable = true)
 |-- endstationname: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- membercasual: string (nullable = true)
 |-- rideabletype: string (nullable = true)
 |-- rideabletypeduplicatecolumnname1: string (nullable = true)
 |-- rideid: string (nullable = true)
 |-- startlatitude: string (nullable = true)
 |-- startlongitude: string (nullable = true)
 |-- startstationid: string (nullable = true)
 |-- startstationname: string (nullable = true)
 |-- starttime: string (nullable = true)
 |-- stoptime: string (nullable = true)
 |-- tripduration: string (nullable = true)
 |-- unnamed:0: integer (nullable = true)
 |-- usertype: string (nullable = true)
```

6.2 Standardizing Date Formats

The column `starttime` contained inconsistent date formats, making it difficult to perform time-based analysis. To address this, we standardized the column to the format `"yyyy-MM-dd HH:mm:ss"` using the following steps:

6.2.1 Define Date Formats

Three commonly observed formats were identified:

- `format_1`: `MM/dd/yyyy H:mm:ss` (e.g., ”2/1/2015 0:00:00”).
- `format_2`: `yyyy-MM-dd HH:mm:ss[.SSSS]` (e.g., ”2016-09-01 00:00:02”).
- `format_3`: `M/d/yyyy H:mm` (e.g., ”2/1/2015 0:00”).

6.2.2 Standardize Dates

Using Spark’s `when` and `otherwise` functions, the `starttime` column was first standardized to include leading zeros for months, days, and hours. The column was then parsed using the defined formats in the order of priority. Rows that failed parsing were flagged.

```python
data = data.withColumn(
    "starttime_standardized",
    F.when(
        F.col("starttime").rlike(r"^\d{1,2}/\d{1,2}/\d{4} \d{1,2}:\d{2}:\d{2}$"),
        F.concat(
            F.split(F.split(F.col("starttime"), "/")[2], " ")[0],
            F.lit("-"),
            F.lpad(F.split(F.col("starttime"), "/")[0], 2, "0"),
            F.lit("-"),
            F.lpad(F.split(F.col("starttime"), "/")[1], 2, "0"),
            F.lit(" "),
            F.split(F.col("starttime"), " ")[1]
        )
    ).otherwise(F.col("starttime"))
)

data = data.withColumn(
    "starttime_parsed",
    F.when(
        F.to_timestamp(F.col("starttime_standardized"), format_2).isNotNull(),
        F.to_timestamp(F.col("starttime_standardized"), format_2)
    ).when(
        F.to_timestamp(F.col("starttime_standardized"), format_1).isNotNull(),
        F.to_timestamp(F.col("starttime_standardized"), format_1)
    ).otherwise(
        F.to_timestamp(F.col("starttime_standardized"), format_3)
    )
)
```

6.3 Result of Standardization

The cleaned and standardized `starttime` column is now consistent across all rows. This allows for time-based feature engineering and analysis in subsequent stages.

7 Pipeline-2 Feature Engineering

This section describes the process of generating new features from the standardized `starttime_parsed` column to facilitate time-based demand prediction and station-level analysis.

7.1 Generating Time-Based Features

From the `starttime_parsed` column, additional time-based features were extracted to capture patterns in bike-sharing demand across different temporal dimensions.

7.1.1 Steps to Extract Time-Based Features

The following features were derived:

- **`date`:** Extracted the date (year, month, and day) using `to_date`.
- **`hour`:** Captures the hour of the day using `hour`, which helps model hourly demand variations.
- **`day_of_week`:** Extracted using `dayofweek`, where days are numbered from Sunday (1) to Saturday (7), enabling weekday versus weekend analysis.
- **`month`:** Captures the month of the year using `month`, allowing analysis of seasonal demand patterns.
- **`year`:** Extracted using `year` for understanding demand trends over time.

```
# Generate time-based features
data = data.withColumn("date", F.to_date("starttime_parsed"))
data = data.withColumn("hour", F.hour("starttime_parsed"))
data = data.withColumn("day_of_week", F.dayofweek("starttime_parsed"))
data = data.withColumn("month", F.month("starttime_parsed"))
data = data.withColumn("year", F.year("starttime_parsed"))
```

8 Pipeline-2 Engineering the Target Variable: Demand

This section explains the process of creating the target variable (`demand`) for the demand prediction model. The target variable represents the hourly demand for bike-sharing services, aggregated across various temporal and spatial dimensions.

8.1 Target Variable Definition

The `demand` variable was engineered by counting the number of trips starting from each station within a specific time period. The aggregation was performed using the following key dimensions:

- `startstationname`: Identifies the station where the trip started.
- `day_of_week`: Captures weekly patterns, such as differences between weekdays and weekends.
- `month` and `year`: Tracks seasonal and yearly trends.
- `hour`: Aggregates demand for each hour of the day, highlighting hourly patterns.
- `date`: Maintains daily granularity to support temporal trend analysis.

8.2 Aggregation Logic

The `groupBy` operation in PySpark was utilized to compute the count of trips (`demand`) for each combination of these dimensions. This process ensured that the target variable accurately reflected the temporal and spatial distribution of demand.

```
# Aggregate demand data
demand_data = data.groupBy("startstationname", "day_of_week", "month", "date", "hour", "year") \
                   .agg(F.count("*").alias("demand"))
```

8.3 Handling Missing Data

To ensure the reliability of the engineered target variable, rows with `NULL` values were removed. This step was crucial for maintaining the integrity of the dataset used for model training.

```
# Drop rows with NULL values
demand_data = demand_data.dropna()
```

8.4 Resulting Target Variable

The resulting `demand` variable represents the aggregated count of trips for each station and time combination. This target variable serves as the foundation for building predictive models in subsequent pipelines.

9 Pipeline-2 Outlier Removal

This section describes the process of identifying and removing outliers from the `demand` data to ensure the quality and reliability of the target variable for modeling.

9.1 Descriptive Statistics

The initial analysis of the `demand` variable provided insights into its distribution:

- **Mean:** 5.0965
- **Standard Deviation:** 6.6911

These statistics were used to compute Z-scores for detecting outliers.

9.2 Z-Score Calculation

The Z-score was calculated for each `demand` value using the formula:

$$Z = \frac{(x - \mu)}{\sigma}$$

where:

- x: Individual `demand` value.
- μ: Mean demand (5.0965).
- σ: Standard deviation of demand (6.6911).

```
df_with_zscore = demand_data.withColumn(
    "z_score", (F.col("demand") - mean_demand) / stddev_demand
)
```

9.3 Outlier Detection and Removal

Rows with Z-scores greater than 3 or less than -3 were considered outliers and filtered out. This corresponds to removing demand values that are more than three standard deviations away from the mean.

```
outliers = df_with_zscore.filter(
    (F.col("z_score") > 3) | (F.col("z_score") < -3)
)

# Filter data to remove outliers
demand_data = df_with_zscore.filter(
    (F.col("z_score") <= 3) & (F.col("z_score") >= -3)
)
```

9.4 Final Dataset After Outlier Removal

The cleaned dataset was finalized by dropping the `z_score` column, as it is no longer required. The filtered dataset ensures that extreme values do not negatively impact model training and predictions.

```
# Drop the z_score column if it's no longer needed
demand_data = demand_data.drop("z_score")
```

9.5 Key Outcomes

- Identified and removed rows with extreme demand values (outliers).
- Ensured the reliability and quality of the dataset for downstream modeling tasks.

This step improves the robustness of the demand prediction model by removing noise and extreme values from the data.

10 Pipeline-2 Merging Weather Data

In this step, weather data was merged with the demand data to include external environmental factors that influence bike-sharing demand. Weather conditions such as temperature, humidity, and precipitation are crucial features for understanding and predicting demand patterns.

10.1 Loading and Processing Weather Data

The weather data was sourced from a CSV file containing hourly weather records for New York City. The following steps were performed to prepare the weather data:

10.1.1 Loading the Data

The weather data was loaded into a PySpark DataFrame with inferred schemas.

```python
# Load Weather Data
weather_path = "gs://bucket121024/w_data/nyc_w_data_full.csv"
weather_data = spark.read.csv(weather_path, header=True, inferSchema=True)
```

10.1.2 Feature Engineering on Weather Data

Several temporal features were extracted to align the weather data with the demand data:

- `date`: Extracted using `to_date` to represent the calendar date.
- `hour`: Extracted using `hour` to capture hourly variations in weather conditions.
- `day_of_week`: Extracted using `dayofweek` to capture weekly patterns.
- `month` and `year`: Extracted for seasonal and yearly trends.

```python
# Process Weather Data
weather_data = weather_data.withColumn("date", F.to_date("datetime", "yyyy-MM-dd"))
weather_data = weather_data.withColumn("hour", F.hour(F.col("datetime")))
weather_data = weather_data.withColumn("day_of_week", F.dayofweek(F.col("date")))
weather_data = weather_data.withColumn("month", F.month(F.col("date")))
weather_data = weather_data.withColumn("year", F.year("date"))
```

10.1.3 Dropping Irrelevant Columns

Irrelevant columns such as `name`, `feelslike`, `dew`, and others were removed to reduce the size of the dataset and focus on relevant weather variables.

```python
# Drop irrelevant columns
weather_data = weather_data.drop(*[
    'name', 'feelslike', 'dew', 'precipprob', 'preciptype', 'snow', 'snowdepth',
    'windgust', 'winddir', 'sealevelpressure', 'cloudcover', 'solarradiation',
    'solarenergy', 'uvindex', 'severerisk', 'icon', 'stations'
])
```

10.2 Merging Weather Data with Demand Data

The weather data was joined with the demand data on shared temporal features (`date`, `day_of_week`, `month`, `hour`, `year`). A `left` join was used to ensure that all demand data rows are retained, even if corresponding weather records are missing.

```python
# Merge weather data with demand data
merged_data = demand_data.join(weather_data, ["date", "day_of_week", "month", "hour", "year"], "left")
merged_data.printSchema()
```

10.3 Schema of the Merged Dataset

The schema of the merged dataset includes demand-related and weather-related variables:

```
root
 |-- date: date (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- year: integer (nullable = true)
 |-- startstationname: string (nullable = true)
 |-- demand: long (nullable = false)
 |-- datetime: timestamp (nullable = true)
 |-- temp: double (nullable = true)
 |-- humidity: double (nullable = true)
 |-- precip: double (nullable = true)
 |-- windspeed: double (nullable = true)
 |-- visibility: double (nullable = true)
 |-- conditions: string (nullable = true)
```

10.4 Key Features in the Merged Dataset

- **Demand Data:** Includes station names (`startstationname`) and hourly demand (`demand`).
- **Weather Data:** Includes features like `temp`, `humidity`, `precip`, and `visibility`.
- **Temporal Data:** Retains temporal dimensions such as `date`, `hour`, `day_of_week`, `month`, and `year`.

10.5 Outcome

The merged dataset combines hourly demand and weather data, providing a comprehensive dataset for predictive modeling. This step integrates external factors to enhance the accuracy of demand predictions.

11 Pipeline-2: Additional Feature Engineering

This section focuses on creating new features to improve the demand prediction model by adding temporal, categorical, and interaction-based features.

11.1 Derived Features

11.1.1 Weekend Indicator

A new column, `is_weekend`, was added to mark weekends. It is set to 1 for Saturday and Sunday, and 0 for other days.

```python
merged_data = merged_data.withColumn(
    "is_weekend", F.when(F.col("day_of_week").isin(1, 7), 1).otherwise(0)
)
```

11.1.2 COVID-19 Lockdown Indicator

To track demand during lockdowns, a column `is_lockdown` was added. It is 1 for dates between March 15, 2020, and June 30, 2020, and 0 otherwise.

```python
merged_data = merged_data.withColumn(
    "is_lockdown",
    F.when((F.col("date") >= "2020-03-15") & (F.col("date") <= "2020-06-30"), 1).otherwise(0)
)
```

11.2 Handling Categorical Features

11.2.1 Condition Indexing

The `conditions` column (e.g., "Clear", "Rainy") was converted into numerical values using `StringIndexer`.

```python
conditions_indexer = StringIndexer(inputCol="conditions", outputCol="conditions_indexed")
merged_data = conditions_indexer.fit(merged_data).transform(merged_data)
```

11.2.2 Station Name Indexing

Similarly, `startstationname` was indexed for use in modeling.

```
station_indexer = StringIndexer(inputCol="startstationname", outputCol="startstationname_indexed")
merged_data = station_indexer.fit(merged_data).transform(merged_data)
```

11.3 Selected Features

The final set of features for modeling includes:

- **Time-based Features:** `day_of_week`, `month`, `hour`, `year`.
- **Weather Features:** `temp`, `humidity`, `precip`, `windspeed`, `visibility`.
- **Derived Features:** `is_weekend`, `is_lockdown`.
- **Categorical Features:** `conditions_indexed`, `startstationname_indexed`.

11.4 Outcome

These features enhance the dataset by adding relevant temporal, weather, and categorical information, helping the model better understand demand patterns.

12 Pipeline-2: Temporal and Interaction Features

12.1 Lagged and Rolling Features

12.1.1 Lagged Demand

The previous time step's demand was added as `lag_demand_1`. This was done using the `lag` function.

```
from pyspark.sql.window import Window
window = Window.partitionBy("startstationname").orderBy("datetime")
merged_data = merged_data.withColumn("lag_demand_1", F.lag("demand", 1).over(window))
```

12.1.2 Rolling Average Demand

A rolling average of demand over the last three time steps was added as `rolling_avg_demand`.

```
merged_data = merged_data.withColumn(
    "rolling_avg_demand", F.avg("demand").over(window.rowsBetween(-3, 0))
)
```

12.1.3 Handling Missing Values

Missing values in lagged and rolling features were replaced with 0.

```
merged_data = merged_data.na.fill({"lag_demand_1": 0, "rolling_avg_demand": 0})
```

12.2 Categorized Time Periods

The `hour` column was divided into buckets (`morning`, `afternoon`, `evening`, `night`) and indexed for modeling.

```
merged_data = merged_data.withColumn(
    "hour_bucket", F.when(F.col("hour").between(6, 11), "morning")
                    .when(F.col("hour").between(12, 17), "afternoon")
                    .when(F.col("hour").between(18, 23), "evening")
                    .otherwise("night"))
```

12.3 Interaction Features

Interaction terms were added to capture how weather affects demand differently on weekends:

```python
merged_data = merged_data.withColumn("temp_is_weekend", F.col("temp") * F.col("is_weekend"))
merged_data = merged_data.withColumn("humidity_is_weekend", F.col("humidity") * F.col("is_weekend"))
```

12.4 Outcome

Adding these features helps the model better capture demand trends over time, interactions, and historical patterns.

13 Pipeline-2: Train-Test Split

13.1 Ordered Splitting

The dataset was split into training (80%) and testing (20%) sets in chronological order to mimic real-world scenarios.

```python
from pyspark.sql import Window
from pyspark.sql.functions import row_number, col

# Order the data by datetime
window_spec = Window.orderBy("datetime")
ordered_data = merged_data.withColumn("row_num", row_number().over(window_spec))

# Calculate split point
total_rows = ordered_data.count()
split_point = int(total_rows * 0.8)

# Split the data
train_data = ordered_data.filter(col("row_num") <= split_point).drop("row_num")
test_data = ordered_data.filter(col("row_num") > split_point).drop("row_num")
```

13.2 Data Verification

The training set contains 80% of rows, while the testing set holds 20%. Counts were verified to ensure accuracy.

```python
print(f"Training Data Count: {train_data.count()}")
print(f"Testing Data Count: {test_data.count()}")
```

13.3 Saving the Splits

The datasets were saved to Google Cloud Storage in `parquet` format for easy access.

```python
train_data.write.mode("overwrite").parquet("gs://bucket/train_data.parquet")
test_data.write.mode("overwrite").parquet("gs://bucket/test_data.parquet")
```

13.4 Outcome

The data is now split into training and testing sets. Using temporal order ensures the model is tested on future data, simulating real-world scenarios.

14 Pipeline-3: Model Training and Evaluation

In this pipeline, we prepared the data for training, scaled the features, and trained a regression model to predict bike-sharing demand. This section details the steps performed, including data loading, feature scaling, and model training using PySpark's machine learning library.

14.1 Setup and Configuration

The training and testing datasets generated in Pipeline-2 were loaded from Google Cloud Storage (GCS). The Spark cluster consisted of 5 workers, each with 4 cores. To optimize parallel processing, the training and testing datasets were repartitioned:

```python
from pyspark.sql import SparkSession

spark = (SparkSession.builder
    .appName("Models")
    .config("spark.sql.shuffle.partitions", 20)
    .getOrCreate())

train = "gs://bucket121024/pipeline2/train_data.parquet"
test = "gs://bucket121024/pipeline2/test_data.parquet"

# Load and repartition data
train_data = spark.read.parquet(train)
test_data = spark.read.parquet(test)
train_data = train_data.repartition(80)  # 80 partitions for training data
test_data = test_data.repartition(40)   # 40 partitions for testing data
```

14.2 Feature Selection and Scaling

14.2.1 Selected Features

The following features were used for modeling:

- **Time-based Features:** `day_of_week`, `month`, `hour`, `year`.
- **Weather Features:** `temp`, `humidity`, `precip`, `windspeed`, `visibility`.
- **Derived Features:** `is_weekend`, `is_lockdown`, `humidity_is_weekend`, `temp_is_weekend`.
- **Categorical Features:** `startstationname_indexed`, `hour_bucket_indexed`.
- **Lagged and Rolling Features:** `rolling_avg_demand`, `lag_demand_1`.

```python
feature_cols = [
    "day_of_week", "month", "hour", "year", "temp", "humidity",
    "precip", "windspeed", "visibility", "is_weekend", "is_lockdown",
    "startstationname_indexed", "humidity_is_weekend", "temp_is_weekend",
    "hour_bucket_indexed", "rolling_avg_demand", "lag_demand_1"
]
```

14.2.2 Feature Scaling

To ensure uniform scaling of features and improve model convergence, the feature set was scaled using `StandardScaler`:

```python
from pyspark.ml.feature import VectorAssembler, StandardScaler

# Assemble features and scale
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
scaler = StandardScaler(inputCol="features", outputCol="scaledFeatures")
```

14.3 Outcome

At this stage, the dataset is prepared with scaled features and repartitioned for efficient model training. The next steps involve model selection, hyperparameter tuning, and evaluation.

15 Feature Relations - Visualizations

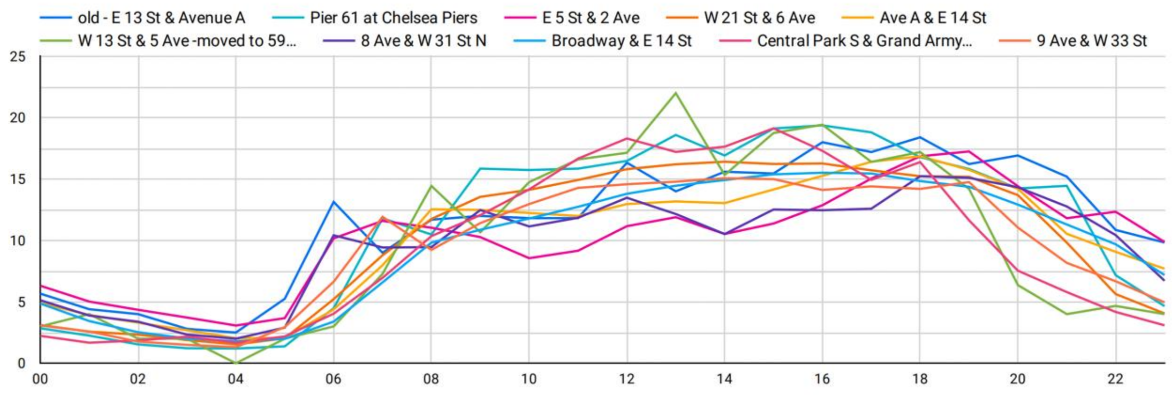


Figure 3: Average demand per hour per station.

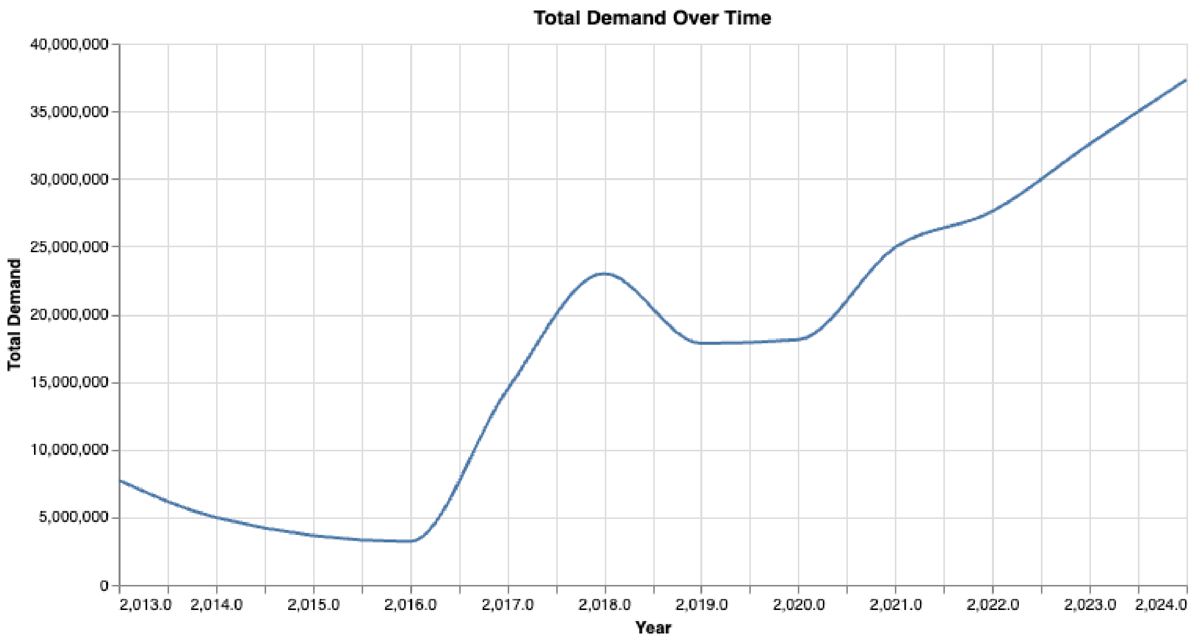


Figure 4: Total Demand over years.

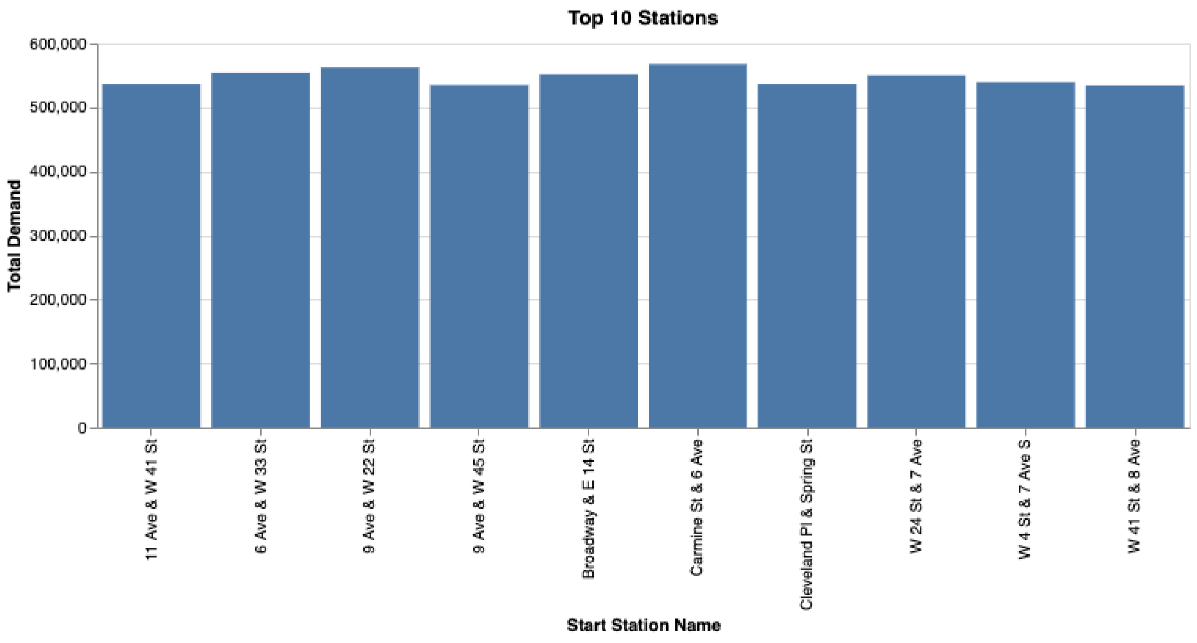


Figure 5: Total Demand over top 10 stations.

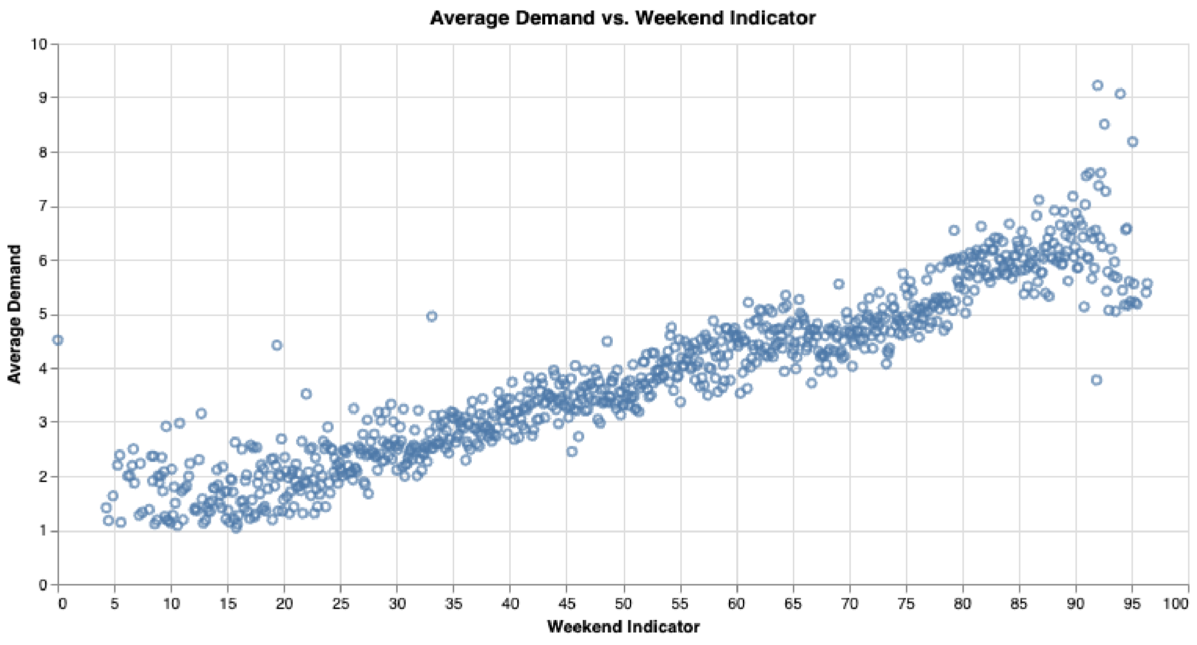


Figure 6: avg Demand over weekend*temparature.

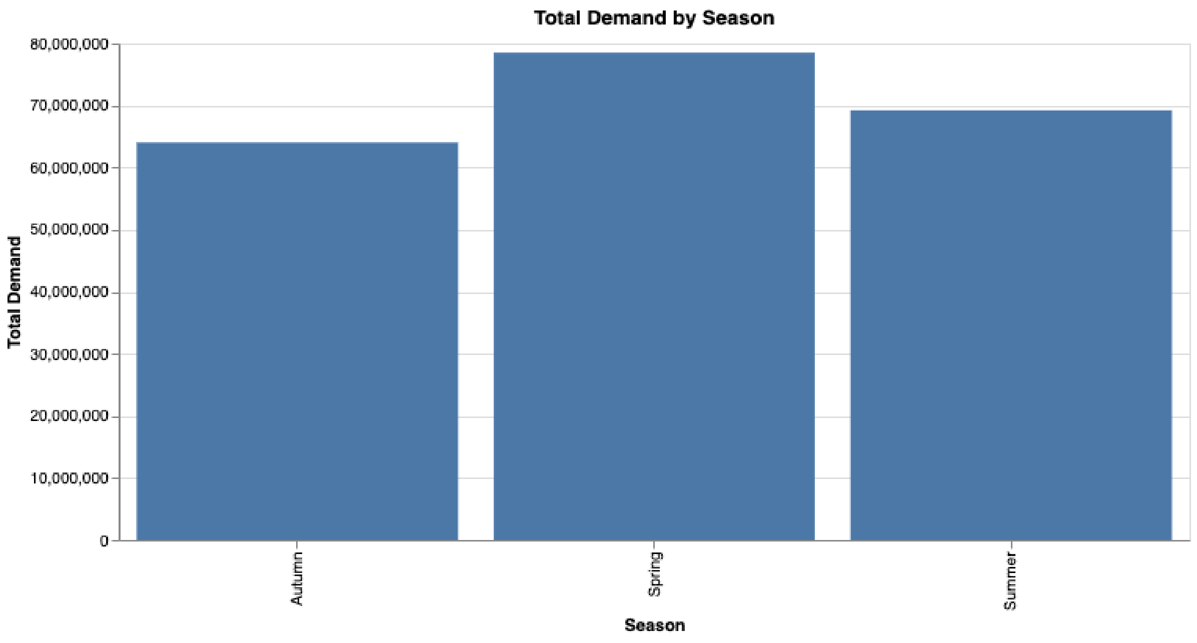


Figure 7: Demand by season.

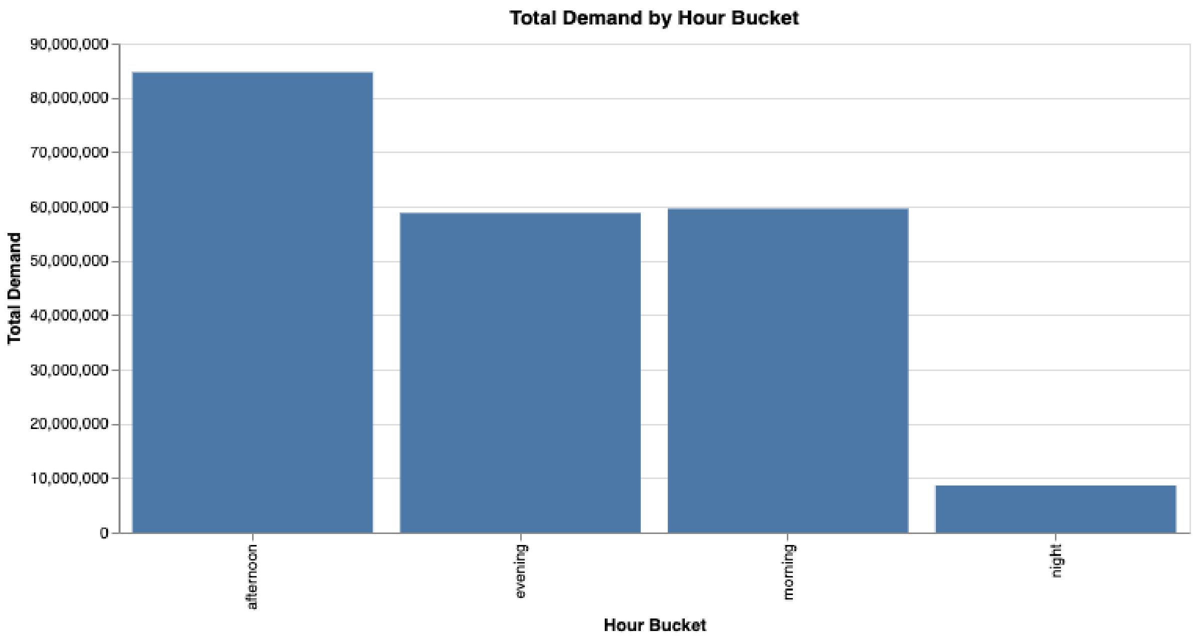


Figure 8: Demand by time of the day.

16 Model Training: Decision Tree Regressor

In this step, we trained a Decision Tree Regressor to predict bike-sharing demand using the preprocessed and scaled features. A Train-Validation Split (TVS) approach was employed for model tuning and evaluation.

16.1 Model and Pipeline Definition

The `DecisionTreeRegressor` model was used as the base regressor. The model was integrated into a pipeline consisting of:

1. **Feature Assembler:** Combines all selected features into a single vector (`features`).
2. **Scaler:** Standardizes the feature vector (`scaledFeatures`).
3. **Regressor:** The `DecisionTreeRegressor` predicts the demand (`demand`).

```python
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml import Pipeline

dt = DecisionTreeRegressor(featuresCol="scaledFeatures", labelCol="demand")

# Define pipeline
pipeline = Pipeline(stages=[assembler, scaler, dt])
```

16.2 Hyperparameter Tuning

The model was tuned using a `ParamGridBuilder` to evaluate combinations of hyperparameters:

- `maxDepth`: Maximum depth of the tree (`[3, 5]`).
- `minInstancesPerNode`: Minimum number of instances per node (`[1, 2]`).
- `maxBins`: Maximum number of bins for splitting (`[16]`).

```python
param_grid = (ParamGridBuilder()
              .addGrid(dt.maxDepth, [3, 5])
              .addGrid(dt.minInstancesPerNode, [1, 2])
              .addGrid(dt.maxBins, [16])
              .build())
```

A Train-Validation Split was used for hyperparameter tuning, with 80% of the training data used for model fitting and 20% for validation.

```python
from pyspark.ml.tuning import TrainValidationSplit

train_val_split = TrainValidationSplit(
    estimator=pipeline,
    estimatorParamMaps=param_grid,
    evaluator=evaluator,
    trainRatio=0.8
)
```

17 Model Training: Random Forest Regressor

In this section, we trained a Random Forest Regressor to predict bike-sharing demand. The Random Forest model is an ensemble learning method that improves predictive performance by aggregating the results of multiple decision trees.

17.1 Model and Pipeline Definition

The `RandomForestRegressor` was used as the base model, and it was integrated into a pipeline consisting of:

1. **Feature Assembler:** Combines all selected features into a single vector (`features`).
2. **Scaler:** Standardizes the feature vector (`scaledFeatures`).

3. **Regressor:** The `RandomForestRegressor` predicts the demand (`demand`).

```python
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml import Pipeline

# Initialize Random Forest Regressor
rf = RandomForestRegressor(featuresCol="scaledFeatures", labelCol="demand")

# Define pipeline
rf_pipeline = Pipeline(stages=[assembler, scaler, rf])
```

17.2 Hyperparameter Tuning

Hyperparameter tuning was performed using `ParamGridBuilder` to evaluate combinations of the following parameters:

- `numTrees`: Number of trees in the forest (`[50, 100]`).
- `maxDepth`: Maximum depth of each tree (`[5, 10]`).

```python
# Define hyperparameter grid
param_grid = (ParamGridBuilder()
              .addGrid(rf.numTrees, [50, 100])
              .addGrid(rf.maxDepth, [5, 10])
              .build())
```

A Train-Validation Split approach was used for hyperparameter tuning, with 80% of the training data used for model fitting and 20% for validation.

```python
# TrainValidationSplit for tuning
train_val_split = TrainValidationSplit(estimator=rf_pipeline,
                                       estimatorParamMaps=param_grid,
                                       evaluator=evaluator,
                                       trainRatio=0.8)
```

18 Model Training: Linear Regression

This section describes the training and evaluation of a Linear Regression model to predict bike-sharing demand. Linear Regression serves as a baseline model for regression tasks.

18.1 Model and Pipeline Definition

The `LinearRegression` model was used, integrated into a pipeline consisting of:

1. **Feature Assembler:** Combines selected features into a single vector (`features`).
2. **Scaler:** Standardizes the feature vector (`scaledFeatures`).
3. **Regressor:** The `LinearRegression` model predicts the demand (`demand`).

```python
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline

lr = LinearRegression(featuresCol="scaledFeatures", labelCol="demand")

# Define pipeline
lr_pipeline = Pipeline(stages=[assembler, scaler, lr])
```

18.2 Hyperparameter Tuning

Hyperparameter tuning was conducted using a `ParamGridBuilder` with the following parameters:

- `regParam`: Regularization parameter (`[0.1, 0.3]`).
- `elasticNetParam`: Elastic Net mixing parameter (`[0.0, 0.5]`).

```
param_grid = (ParamGridBuilder()
               .addGrid(lr.regParam, [0.1, 0.3])
               .addGrid(lr.elasticNetParam, [0.0, 0.5])
               .build())
```

A Train-Validation Split was used for hyperparameter tuning, with 80% of the training data used for model fitting and 20% for validation.

```
train_val_split = TrainValidationSplit(estimator=lr_pipeline,
                                       estimatorParamMaps=param_grid,
                                       evaluator=rmse_evaluator,
                                       trainRatio=0.8)
```

19 Model Training: Gradient-Boosted Tree (GBT)

This section describes the training and evaluation of a Gradient-Boosted Tree (GBT) Regressor to predict bike-sharing demand. GBT models are powerful for regression tasks due to their ability to handle non-linear relationships.

19.1 Model and Pipeline Definition

The `GBTRegressor` model was used, integrated into a pipeline consisting of:

1. **Feature Assembler:** Combines selected features into a single vector (`features`).
2. **Scaler:** Standardizes the feature vector (`scaledFeatures`).
3. **Regressor:** The `GBTRegressor` model predicts the demand (`demand`).

```
from pyspark.ml.regression import GBTRegressor
from pyspark.ml import Pipeline

gbt = GBTRegressor(featuresCol="scaledFeatures", labelCol="demand")

# Define pipeline
gbt_pipeline = Pipeline(stages=[assembler, scaler, gbt])
```

19.2 Hyperparameter Tuning

Hyperparameter tuning was conducted using a `ParamGridBuilder` with the following parameters:

- `maxDepth`: Maximum depth of the tree (`[3, 5]`).
- `maxIter`: Number of iterations (`[10, 20]`).
- `stepSize`: Step size for learning rate (`[0.05, 0.1]`).

```
param_grid = (ParamGridBuilder()
               .addGrid(gbt.maxDepth, [3, 5])
               .addGrid(gbt.maxIter, [10, 20])
               .addGrid(gbt.stepSize, [0.05, 0.1])
               .build())
```

A Train-Validation Split was used for hyperparameter tuning, with 80% of the training data used for model fitting and 20% for validation.

```python
from pyspark.ml.tuning import TrainValidationSplit
from pyspark.ml.evaluation import RegressionEvaluator

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

train_val_split = TrainValidationSplit(estimator=gbt_pipeline,
                                       estimatorParamMaps=param_grid,
                                       evaluator=evaluator,
                                       trainRatio=0.8)
```

19.3 Evaluation Metrics

The best models was evaluated on the test data using metrics such as RMSE, R^2, MAE, and MSE.

```python
# Evaluate metrics
model = train_val_split.fit(train_data)
best_model = model.bestModel

predictions = best_model.transform(test_data)

rmse = evaluator.evaluate(predictions)  # RMSE
r2 = RegressionEvaluator(labelCol="demand",
                             predictionCol="prediction",
                             metricName="r2").evaluate(predictions)  # R²
mae = RegressionEvaluator(labelCol="demand",
                              predictionCol="prediction",
                              metricName="mae").evaluate(predictions)  # MAE
mse = RegressionEvaluator(labelCol="demand",
                              predictionCol="prediction",
                              metricName="mse").evaluate(predictions)  # MSE

# Print evaluation metrics
print(f"Regressor - Best RMSE: {rmse}")
print(f"Regressor - Best R2: {r2}")
print(f"Regressor - Best MAE: {mae}")
print(f"Regressor - Best MSE: {mse}")
```

20 Scaling Strategies and Metrics

To evaluate the performance and scalability of the models, we trained all four models using different configurations of cluster sizes and data proportions. The following scaling strategies were employed:

20.1 Cluster Configurations

Each model was trained on clusters with varying numbers of workers to assess the impact of computational resources on model performance. The configurations included:

- **3-worker cluster**
- **4-worker cluster**
- **5-worker cluster**

20.2 Data Proportions

For each cluster configuration, the models were trained using different percentages of the total dataset to evaluate the relationship between data volume and computational efficiency. The data proportions used were:

- **60% of the dataset**
- **80% of the dataset**
- **100% of the dataset**

20.3 Performance Metrics

The scalability and performance were evaluated using the following metrics:

- **Training Time:** The time required to train each model under different configurations.
- **Prediction Accuracy:** Measured using metrics specific to each model type, such as RMSE, R^2, MAE, or MSE.

20.4 Results with 3 Workers

Table 2: Model Performance with 3 Workers

Model	**Data (%)**	**RMSE**	**R^2**	**MAE**	**Wall Time**
Decision Tree	60%	2.3178	0.7288	1.4936	6 min 2 s
	80%	2.3177	0.7285	1.4939	8 min 1 s
	100%	2.3259	0.7267	1.5020	9 min 8 s
Random Forest Regressor	60%	2.1088	0.7755	1.3712	56 min 4 s
	80%	2.1096	0.7750	1.3747	2 h 3 min 55 s
Linear Regression	60%	2.4403	0.6993	1.6502	4 min 2 s
	80%	2.4398	0.6993	1.6498	6 min 3 s
	100%	2.4395	0.6994	1.6496	7 min 9 s
GBT Regressor	60%	2.0692	0.7838	1.3475	22 min 49 s

20.5 Results with 4 Workers

Table 3: Model Performance with 4 Workers

Model	**Data (%)**	**RMSE**	**R^2**	**MAE**	**Wall Time**
Decision Tree	60%	2.3167	0.7289	1.5020	5 min 31 s
	80%	2.3246	0.7269	1.4923	7 min 34 s
	100%	2.3214	0.7278	1.4941	7 min 6 s
Random Forest Regressor	60%	2.1145	0.7742	1.3757	53 min 19 s
	80%	2.1061	0.7759	1.3700	1 h 7 min 2 s
	100%	2.0912	0.7791	1.3652	1 h 6 min 36 s
Linear Regression	60%	2.4391	0.6995	1.6491	2 min 39 s
	80%	2.4386	0.6995	1.6491	3 min 44 s
	100%	2.4395	0.6994	1.6496	3 min 12 s
GBT Regressor	60%	2.0792	0.7816	1.3480	16 min 50 s
	80%	2.0690	0.7837	1.3472	23 min 8 s
	100%	2.0775	0.7820	1.3508	24 min 49 s

20.6 Results with 5 Workers

Table 4: Model Performance with 5 Workers

Model	**Data (%)**	**RMSE**	**R^2**	**MAE**	**Wall Time**
Decision Tree	60%	2.3371	0.7244	1.5108	6 min 11 s
	80%	2.3697	0.7162	1.5122	7 min 45 s
	100%	2.3171	0.7288	1.4996	9 min 46 s
Random Forest Regressor	60%	2.1156	0.7742	1.3778	54 min 7 s
	80%	2.1072	0.7756	1.3733	1 h 10 min 13 s
	100%	2.0931	0.7787	1.3630	1 h 20 min 4 s
Linear Regression	60%	2.4400	0.6996	1.6498	3 min 31 s
	80%	2.4389	0.6994	1.6495	4 min 20 s
	100%	2.4395	0.6994	1.6496	5 min 28 s
GBT Regressor	60%	2.0772	0.7819	1.3487	22 min 29 s
	80%	2.0769	0.7820	1.3497	24 min 55 s
	100%	2.0675	0.7841	1.3472	35 min

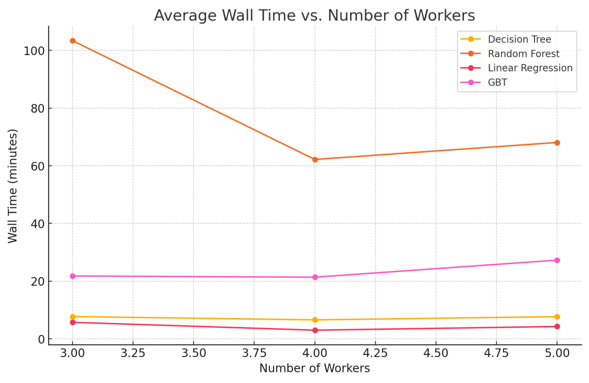


Figure 9: time over workers.

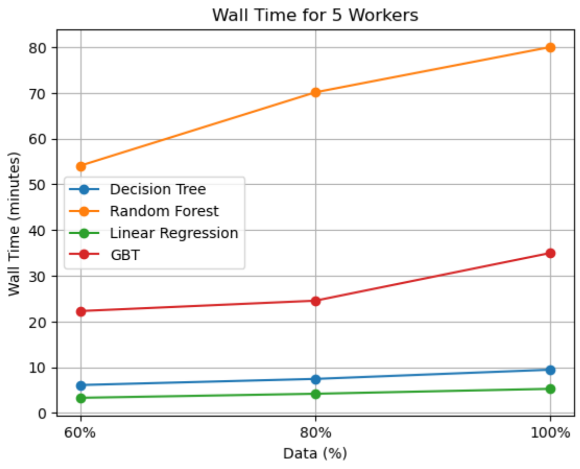


Figure 11: time for 5 workers.

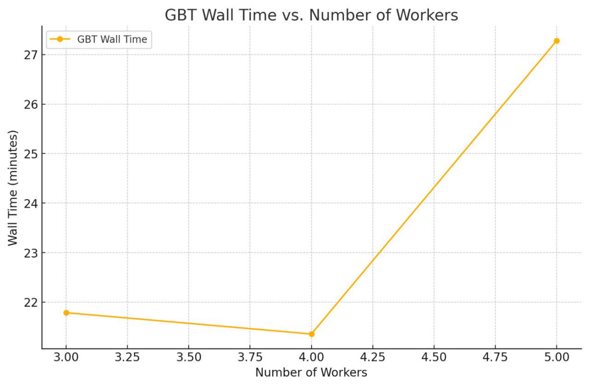


Figure 10: Best model time vs number of workers.

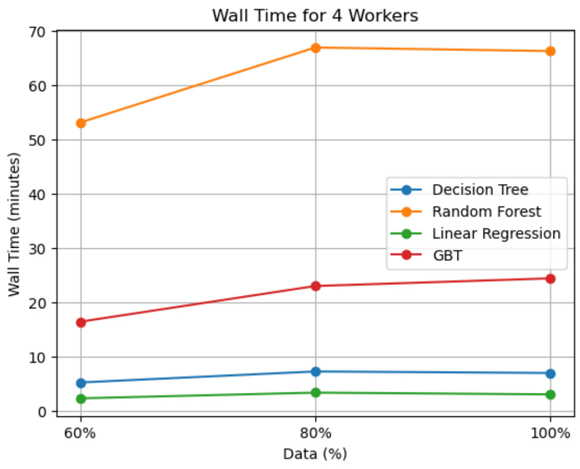


Figure 12: time for 4 workers.

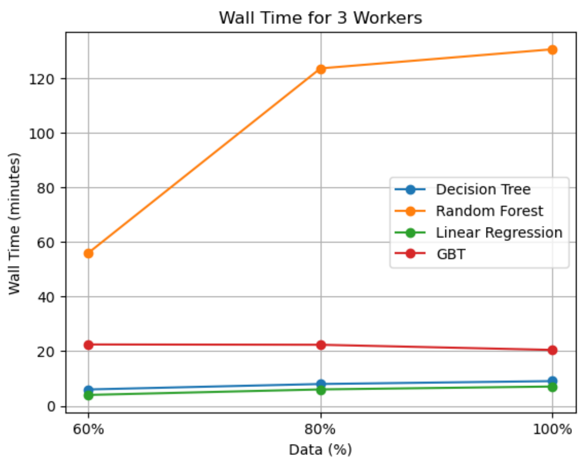


Figure 13: time for 3 workers.

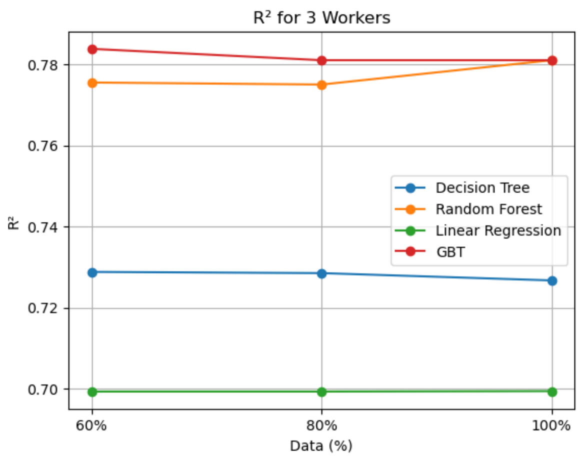


Figure 16: R2 - 3 worker.

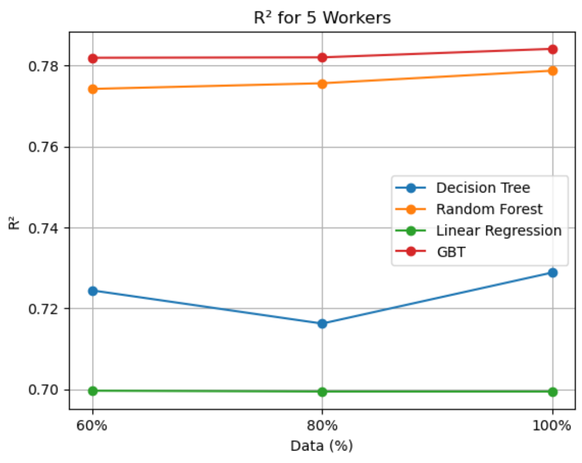


Figure 14: R2 5 workers.

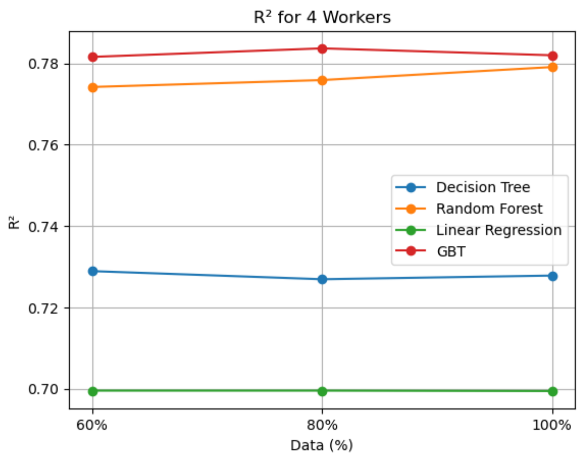


Figure 15: R2 - 4 worker.

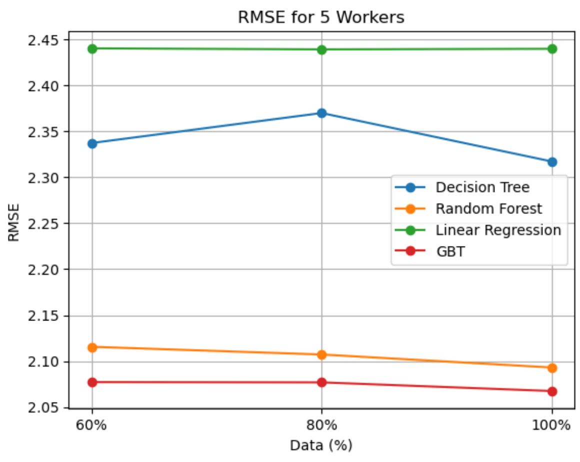


Figure 17: RMSE - 5 worker.

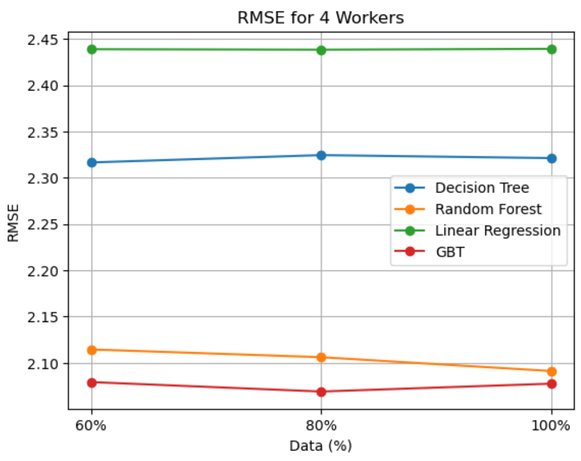


Figure 18: RMSE - 4 worker.

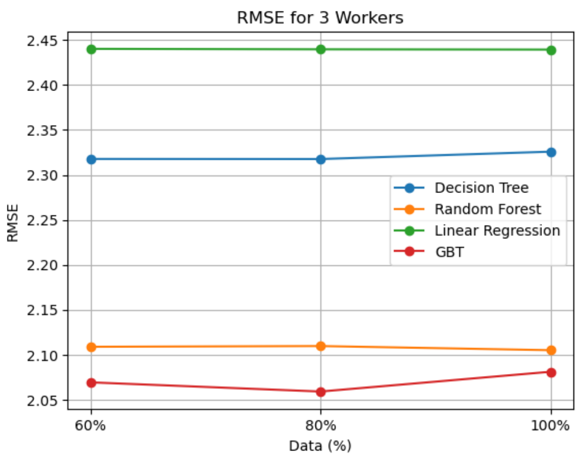


Figure 19: RMSE - 3 worker.

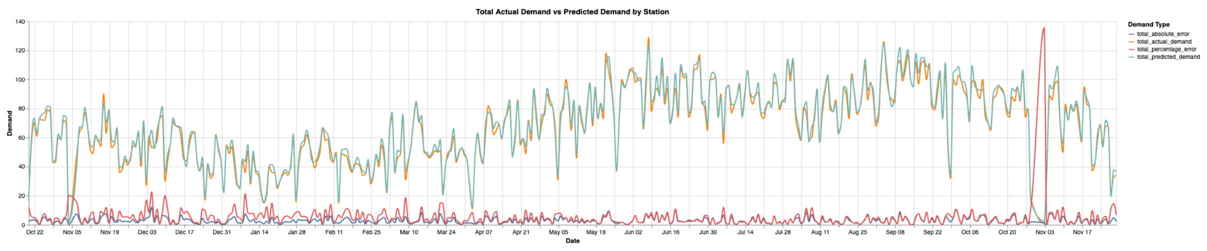


Figure 20: predicted vs actual RF

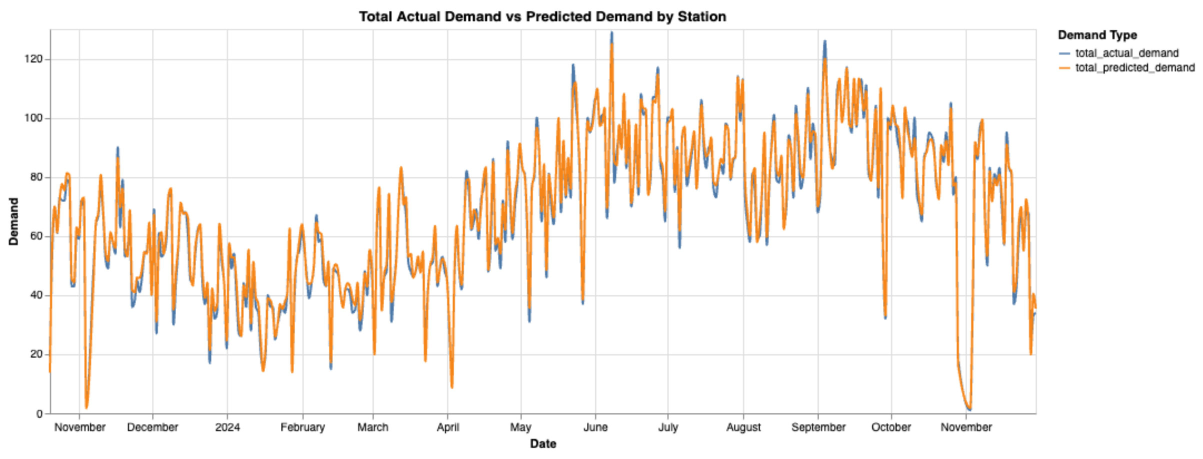


Figure 21: predicted vs actual LR.

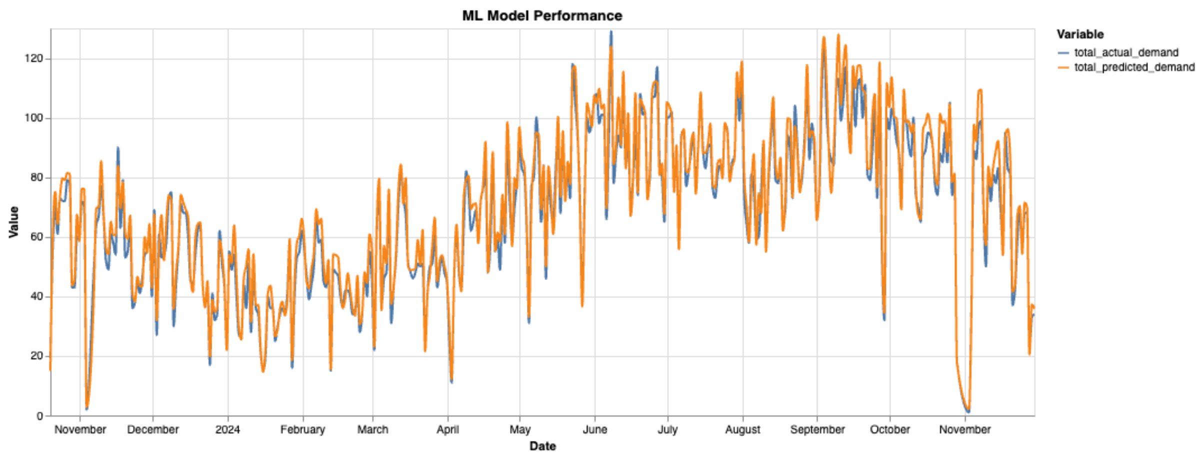


Figure 22: predicted vs actual.

21 Conclusion

To evaluate the performance of different models based on the number of workers and data percentages, we analyzed their RMSE (lower is better), R^2 (higher is better), and Wall Time (lower is better).

Analysis by Metrics

- **RMSE (Root Mean Square Error):**
  - **3 Workers:** Best Model: **GBT** (lowest RMSE at 60% data: 2.0692).
  - **4 Workers:** Best Model: **GBT** (lowest RMSE at 80% data: 2.0690).
  - **5 Workers:** Best Model: **GBT** (lowest RMSE at 100% data: 2.0675).

  Overall, **GBT consistently has the lowest RMSE across all worker groups.**

- **R^2 (Coefficient of Determination):**
  - **3 Workers:** Best Model: **GBT** (highest R^2 at 60% data: 0.7838).
  - **4 Workers:** Best Model: **GBT** (highest R^2 at 80% data: 0.7837).
  - **5 Workers:** Best Model: **GBT** (highest R^2 at 100% data: 0.7841).

  Overall, **GBT performs best in terms of R^2 across all worker groups and data percentages.**

- **Wall Time:**
  - **3 Workers:** Fastest Model: **Linear Regression** (lowest wall time at 60% data: 241.2 seconds).
  - **4 Workers:** Fastest Model: **Linear Regression** (lowest wall time at 60% data: 143.4 seconds).
  - **5 Workers:** Fastest Model: **Linear Regression** (lowest wall time at 60% data: 198.6 seconds).

  Overall, **Linear Regression is the fastest model, with consistently low wall time across all workers and data percentages.**

Summary of Best Models

RMSE	GBT	All worker groups, all data	Lowest error in predictions.
R^2	GBT	All worker groups, all data	Best predictive performance.
Wall Time	Linear Regression	All worker groups, all data	Fastest computation time.

Overall Recommendation

- **GBT:** Best for accuracy (lowest RMSE, highest R^2), making it ideal when prediction quality is critical.
- **Linear Regression:** Best for speed (lowest Wall Time), suitable for scenarios requiring quick results over high accuracy.