OPTIMIZING NYC BIKE SHARE: HOURLY DEMAND FORECASTING WITH BIG DATA

Chenwei Wan
Télécom Paris, Institut Polytechnique de Paris
chenwei.wan@telecom-paris.fr

ABSTRACT

This project presents a big data system designed to predict the hourly demand and supply of Citi Bike stations in NYC, aiming to optimize the efficiency of the public bike-sharing system. Our system aggregates multiple public data sources, including the complete Citi Bike trip data for 2023, hourly NYC weather data for 2023, and GeoJSON data for visualization. By integrating Apache Spark, we efficiently clean and enrich large volumes of data, facilitating the training of a demand/supply prediction model based on historical time series and weather information. Our XGBoost prediction model achieves superior performance, with a Mean Absolute Error (MAE) of 0.85 and an R^2 score of 0.98. Additionally, we develop an interactive dashboard using D3.js to visualize the demand and supply dynamics of the entire bike-sharing system.

Code: https://github.com/cw-wan/BigDataProject-CityBikeNY

1 INTRODUCTION & BUSINESS OBJECTIVE

Bike-sharing systems are widely adopted worldwide, particularly in major cities like New York City and Paris. They offer a convenient transportation option for citizens while promoting a green and healthy lifestyle. Users can rent or return a bike at any station by swiping their membership, generating a rental or return record that includes their user ID, bike ID, station name, and timestamp.

However, jamming situations can frequently occur, hindering the efficiency of bike-sharing systems. Some stations may face a shortage of bikes, while others may receive more incoming bikes than the demand, potentially exceeding their capacity. To address this issue, system operators continuously reposition bikes among stations using trailers. However, reacting only after stations become jammed or starved is often too late, especially when large-scale repositioning is required. *Therefore, **forecasting** the demand and supply of bike stations is a promising **business objective.*** By predicting stations with an excess supply and those experiencing shortages, bikes can be proactively transported before jamming situations occur.

Building such a forecasting system is far from easy, as the demand and supply of individual bike stations can fluctuate significantly and are influenced by a combination of complex factors, including *time*, *weather conditions*, and *events* (Li & Zheng, 2020).

Considering these challenges, we propose the design of a big data system capable of forecasting the hourly demand-supply dynamics in a bike-sharing system. Our system aggregates multiple data sources, including trip data generated by city bike users, hourly weather condition data, and city geometric data. By cleaning and enriching large-scale data, we are able to train a time series prediction model *offline*, which demonstrates superior performance across several metrics, including a Mean Absolute Error (MAE) of 0.85 and an R^2 score of 0.98. Additionally, we develop an interactive dashboard using D3.js to visualize the demand and supply dynamics of the entire bike-sharing system.

2 Data

We use the complete 2023 trip data released by Citi Bike Company[1] as our primary data source. The trip data records the starting and destination points (including station name, station ID, longitude, and latitude) for each trip, along with their corresponding timestamps.

Our weather data comes from Kaggle[2], which provides detailed hourly weather conditions for New York City in 2023. This dataset, available in CSV format, includes weather category, temperature, wind speed, visibility, rainfall, snowfall, and other weather variables.

Additionally, we collect geometric data for NYC and Jersey City in GeoJSON format from GitHub[3] and the Jersey City government database[4].

3 System Pipeline

The overall pipeline of our proposed big data system (*offline part*) is illustrated in Figure 1. The inputs to this pipeline are raw trip data and weather data, while the outputs include **station and clustering center (zone) data**, as well as **time series data** that records the demand and supply along with weather information for each zone throughout the entire year, using an hourly timestamp as the basic granularity. This **time series data** is then used to train an XGBoost prediction model, which serves as the core of the *online* forecasting system. In this project, we implement only the *offline* part of the system. Our implementation is based on Scala Spark (Core, SQL and MLlib), Scikit-learn, Akka HTTP and D3.js.

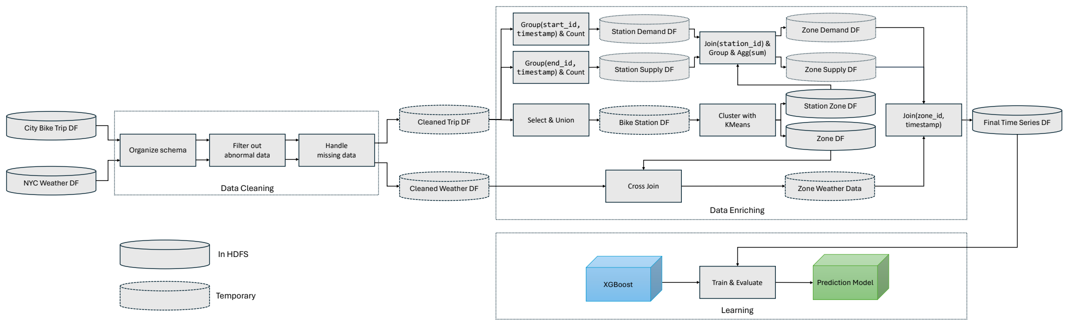


Figure 1: Overview of our big data pipeline (*offline part*).

3.1 Data Cleaning

Data cleaning consists of three stages, as illustrated in Figure 1:

1. *Organizing schema*: Since the raw data is in CSV format, we explicitly define the schema and declare data types for the columns. An important task is to unify timestamps in the trip data and weather data, as they are recorded in different formats. We also drop columns containing unnecessary information or missing values across all records, such as `snow_3h` in the weather data.
2. *Filtering*: We design several rules to filter out abnormal data. For example, by computing the time difference between the start and end timestamps for each trip record, we remove erroneous records with negative trip durations and abnormal records where trips exceed 24 hours.

[1] https://citibikenyc.com/system-data

[2] https://www.kaggle.com/datasets/heqiang01/hourly-weather-data-in-2023-in-usa

[3] Outline of districts: https://github.com/codeforgermany/click_that_hood/blob/main/public/data/new-york-city-boroughs.geojson; City streets: https://github.com/fillerwriter/nyc-streets/blob/master/nyc-streets.geojson

[4] https://data.jerseycitynj.gov/explore/dataset/jersey_city_censustract_geojson/api/

3. *Handling missing data*: For weather metrics that do not change abruptly, such as visibility, we fill missing values by inheriting the value from the previous hour's record. For metrics like rainfall and snowfall, we replace missing values with 0 directly.

The schema for cleaned data can be found in Appendix A.1 and A.2.

3.2 Data Enriching

The objective of data enrichment is to integrate trip data and weather data to generate time series data that captures the demand and supply dynamics of the station clusters in the bike-sharing system throughout the year, along with hourly weather data. The detailed workflow is illustrated in Figure 1.

As discussed earlier, a major challenge in building a demand forecasting system is the large fluctuations at individual bike stations. To address this, we *cluster* the thousands of bike stations into 50 zones using the K-Means algorithm based on their longitudes and latitudes. This aggregation regularizes bike usage within each zone, thereby improving prediction accuracy (Li & Zheng, 2020).

The schema of the final time series data frame is provided in Appendix A.3. We also visualize a sample of the demand and supply dynamics for zone 27 in August 2023. Figure 2 depicts how demand and supply fluctuate, while Figure 3 illustrates the difference between demand and supply, which appears to follow some regular pattern. Additionally, we represent weather conditions using background colors: blue for rainy periods, gold for sunny periods, and gray for cloudy periods.

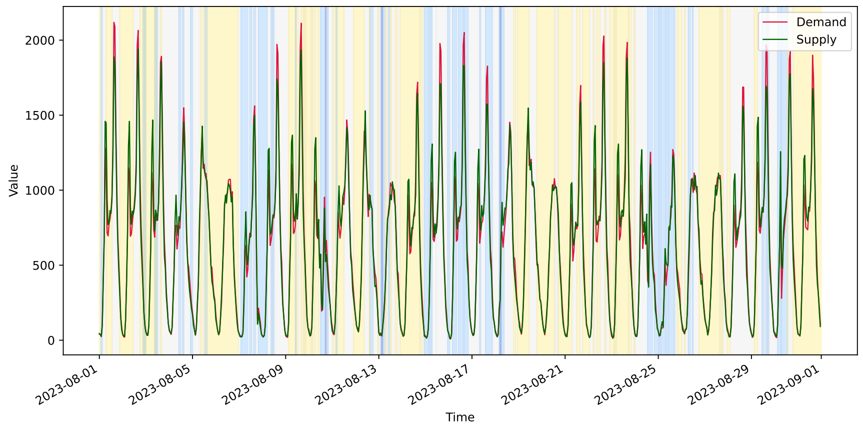


Figure 2: Demand & Supply time series sample for zone 27 in August 2023.

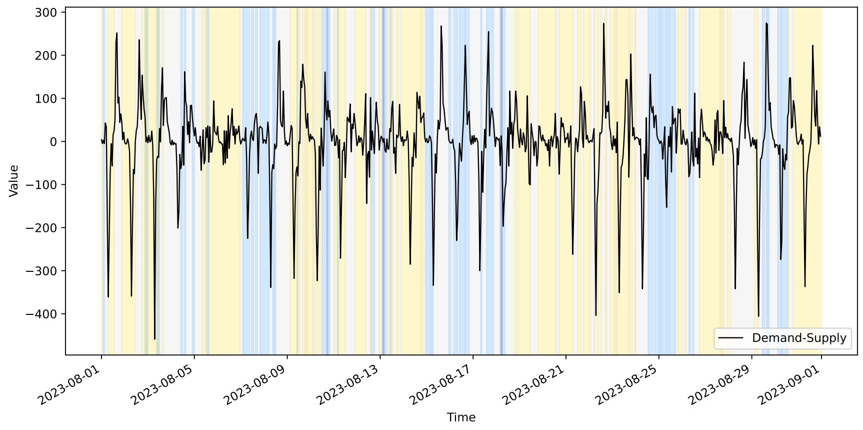


Figure 3: Demand-Supply difference time series sample for zone 27 in August 2023.

3.3 Model Training

With the time series data obtained from the data enrichment process, we train an XGBoost model as the prediction model for the forecasting system. The features used in the forecasting model are organized into several groups, each capturing different aspects of the data. Below is an explanation:

- **Zone Features:**
  - `zone_id`: Identifier for the geographic zone.
  - `zone_lat` and `zone_lng`: Latitude and longitude coordinates of the zone, respectively.
- **Weather Features:**
  - `temp`: The temperature measurement.
  - `visibility`: Visibility level, indicating how far one can see.
  - `wind_speed`: The speed of the wind.
  - `weather_main_index`: A numerical encoding of the primary weather condition (originally a categorical variable such as ”Clear”, ”Clouds”, etc.) obtained via label encoding.
- **Temporal Features:**
  - `hour`: The hour of the day extracted from the timestamp.
  - `day_of_week`: The day of the week (Monday = 0, Sunday = 6).
  - `month`: The month of the year.
  - `is_weekend`: A binary indicator denoting whether the day is a weekend (1 for Saturday/Sunday, 0 otherwise).
- **Time Series Features:**
  - `lag_1`, `lag_24`, and `lag_168`: Lag features representing the target value (i.e., demand minus supply) from 1 hour, 24 hours, and 168 hours (one week) earlier, respectively.
  - `rolling_3h_mean`: A rolling mean computed over a 3-hour window, summarizing short-term trends.
  - `rolling_24h_max`: The maximum value observed in a rolling 24-hour window, capturing peak behaviors.
  - `delta_1h`: The difference between the current target value and the value from one hour ago, which highlights the immediate change.

Detailed parameters of XGBoost can be found in Appendix B. To assess model performance, the following metrics are used:

Root Mean Squared Error (RMSE) RMSE measures the average magnitude of errors in predictions, giving higher weight to large errors:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{1}$$

where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of samples.

Mean Absolute Error (MAE) MAE calculates the average absolute difference between actual and predicted values:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2}$$

It provides a linear scale for errors, treating all deviations equally.

R-squared (R^2) Score The R^2 score evaluates how well the model explains variance in the target variable:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{3}$$

where $\bar{y}$ is the mean of actual values. A value close to 1 indicates a strong model fit.

Median Absolute Error (MedAE) MedAE measures the median of absolute differences between predictions and actual values, being more robust to outliers:

$$\text{MedAE} = \text{median}(|y_i - \hat{y}_i|) \tag{4}$$

Table 1 compares the model's performance with and without weather features. We use data from January to September as the training set and data from October to December for evaluation.

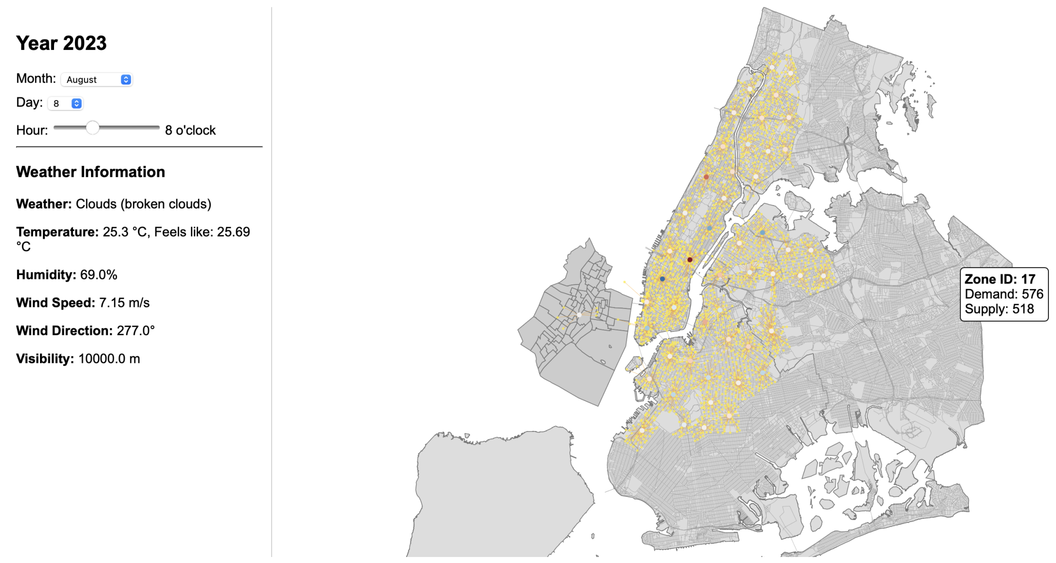
	Full Features	w/o Weather
RMSE↓	**4.8112**	4.8921
MAE↓	**0.8511**	0.9063
R^2↑	**0.9815**	0.9809
MedAE↓	**0.1644**	0.1831

Table 1: Model Performance Comparison

The results indicate that including weather features improves model performance across all metrics. The R^2 score remains high in both cases, showing that the model effectively captures the underlying patterns in demand-supply fluctuations.

3.4 Interactive Visualization

We use Scala with Akka HTTP to build a simple backend server that receives requests from the frontend and queries the data frames stored in HDFS. For the frontend, we use D3.js to create an interactive visualization, as shown in Figure 4. By adjusting the time panel, users can observe how demand and supply change dynamically: if a node turns red, it indicates that the zone is experiencing a shortage of bikes, whereas if it turns blue, it signifies that the zone has an excess supply and can transfer bikes to areas in need.



Year 2023

Month: August

Day: 8

Hour: 8 o'clock

Weather Information

Weather: Clouds (broken clouds)

Temperature: 25.3 °C, Feels like: 25.69 °C

Humidity: 69.0%

Wind Speed: 7.15 m/s

Wind Direction: 277.0°

Visibility: 10000.0 m

Zone ID: 17
Demand: 576
Supply: 518

Figure 4: User interface.

4 Future Work

As this project primarily focuses on the *offline* part of our forecasting system, integrating a data streaming framework is a crucial direction for future work. We can use Spark Streaming to ingest real-time trip and weather data, and process them into time series format using the pipeline proposed above. Predictions can then be made using our offline-trained XGBoost model. Additionally, we can periodically retrain the XGBoost model on newly accumulated streaming data within the Spark framework to adapt to recent trends. Due to various constraints, including limited time, we leave this aspect for future work.

References

Yexin Li and Yu Zheng. Citywide bike usage prediction in a bike-sharing system. *IEEE Transactions on Knowledge and Data Engineering*, 32(6):1079–1091, 2020. doi: 10.1109/TKDE.2019.2898831.

A Schema

A.1 Cleaned City Bike Trip Data Frame

- **`ride_id`**: *string*
- **`rideable_type`**: *string*
- **`started_at`**: *timestamp*
- **`ended_at`**: *timestamp*
- **`start_station_name`**: *string*
- **`start_station_id`**: *string*
- **`end_station_name`**: *string*
- **`end_station_id`**: *string*
- **`start_lat`**: *double*
- **`start_lng`**: *double*
- **`end_lat`**: *double*
- **`end_lng`**: *double*
- **`member_casual`**: *string*
- **`duration`**: *double*
- **`started_at_full`**: *timestamp*
- **`ended_at_full`**: *timestamp*

A.2 Cleaned Weather Data Frame

- **`temp`**: *double*
- **`visibility`**: *double*
- **`feels_like`**: *double*
- **`humidity`**: *double*
- **`wind_speed`**: *double*
- **`wind_deg`**: *double*
- **`wind_gust`**: *double*
- **`rain_1h`**: *double*
- **`snow_1h`**: *double*
- **`clouds_all`**: *double*

- **`weather_main`**: *string*
- **`weather_description`**: *string*
- **`time`**: *timestamp*

A.3 Zone Supply-Demand Time Series Data Frame

- **`zone_id`**: *int*
- **`zone_lat`**: *double*
- **`zone_lng`**: *double*
- **`time`**: *timestamp*
- **`temp`**: *double*
- **`visibility`**: *double*
- **`feels_like`**: *double*
- **`humidity`**: *int*
- **`wind_speed`**: *double*
- **`wind_deg`**: *double*
- **`wind_gust`**: *double*
- **`rain_1h`**: *double*
- **`snow_1h`**: *double*
- **`clouds_all`**: *int*
- **`weather_main`**: *string*
- **`weather_description`**: *string*
- **`demand`**: *int*
- **`supply`**: *int*

B XGBoost Parameters

- **Objective**: Regression using squared error loss (`reg:squarederror`).
- **Number of Trees**: 400 estimators.
- **Max Depth**: 9, controlling tree complexity.
- **Learning Rate**: 0.05, balancing convergence speed and performance.
- **Subsample**: 0.8, reducing overfitting by training on a fraction of data.
- **Colsample by Tree**: 0.8, selecting a fraction of features per tree.
- **Gamma**: 0.1, controlling tree pruning.