NYC Traffic Volume Analysis

Ziyu Zhu

Problem Definition

New York City (NYC) is known for its bustling streets and constant flow of vehicles, making traffic management a critical aspect of urban planning. Analyzing traffic volume in NYC provides valuable insights into the city's transportation patterns and could help to improve traffic flow, reduce congestion, predict traffic, and enhance overall mobility.

The traffic volume data comes from New York City Open data, a free public data published by NYC agencies and other partners. The data set used in this project is Automated Traffic Volume Counts. New York City Department of Transportation (NYC DOT) uses Automated Traffic Recorders (ATR) to collect traffic sample volume counts at bridge crossings and roadways. These counts do not cover the entire year, and the number of days counted per location may vary from year to year. So, it should be noted that the data is not continued counted during the year, so matching the exact same date between years may not work due to different data collection times.

The data used in this paper is published on 20240328, while the latest is published on 20240412. The format of the 2 versions is the same; the 0328 version has 27414482 rows of data, while 0412 only has 1673726 rows of data. With the dataset is about 16 times larger, the paper still use the older 20240328 version. There are 14 columns in the dataset:

RequestID	An unique ID that is generated for each counts request.
Boro	Lists which of the five administrative divisions of New York City the location is within, written as a word
Yr	The two digit year portion of the date when the count was conducted.
M	The two digit month portion of the date when the count was conducted.
D	The two digit day portion of the date when the count was conducted.
HH	The two digit hour portion of the time when the count was conducted.
MM*	The two digit start minute portion of the time when the count was conducted.
Vol	The total sum of count collected within a 15 minute increments.
SegmentID	The ID that identifies each segment of a street in the LION street network version 14.
WktGeom	A text markup language for representing vector geometry objects on a map and spatial reference systems of spatial objects.
street	The 'On Street' where the count took place.
fromSt	The 'From Street' where the count took place.
toSt	The 'To Street' where the count took place.
Direction	The text-based direction of traffic where the count took place.

*: Minute is different in 15, i.e. 00, 15, 30, 45

Table 1 Columns Description

Due to there being thousands of stations in the whole dataset, the analysis in this paper is focused on certain stations taking the measurements. With the same WktGeom, SegmentID and Direction, the top 5 stations data are analyzed. The top 5 station are located on:

1. AMSTERDAM AV between W 95 ST and W 96 ST, North Bound.
2. AMSTERDAM AV between W 60 ST and W 61 ST, North Bound.
3. LONG IS EXWY between DEAD END and LITTLE NECK PKWY, East Bound.
4. 1 AV between E 60 ST/KOCH BR PEDESTRIAN AND BIKE PATH and E 61 ST, East Bound
5. AMSTERDAM AV between W 85 ST and W 86 St, North Bound

Taking all previous years as the sample, the latest year will be simulated as the live-streamed data and then analyzed if the data is normal or anomalous. The data compared in each station will have the same Day of Week, Hour and Minute.

Challenges Faced

One of the challenges faced is finding the station's geographical location and visualizing it. Due to my limited knowledge of geography and cartography, finding the definition of SegmentID in the LION street network and WktGeom coordinates took a very long time. Thanks to Center for International Earth Science Information Network at Columbia Climate School, I used ArcGIS to locate the NodeID in the LION street network. After carefully digging into WktGeom, it turns out that NYC Open Data uses *EPSG:2263* Projected coordinate system for United States (USA) - New York. Then, using a python package *pyproj*, it can turn *EPSG:2263* into EPSG:4326(WGS 84) Geodetic coordinate system for World. It is the most standard machine-readable WKT representation, and can be visualized in map for packages like *folium*. The code to draw the map will not be discussed in this paper.

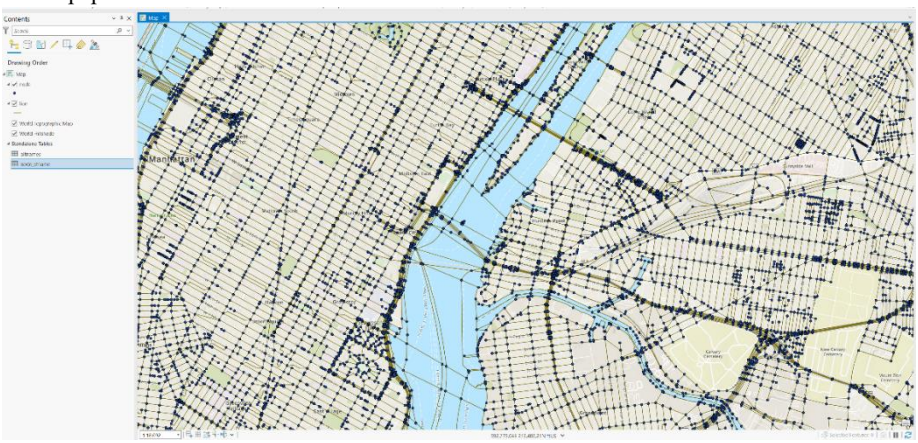


Figure 1 NYC RoadMap

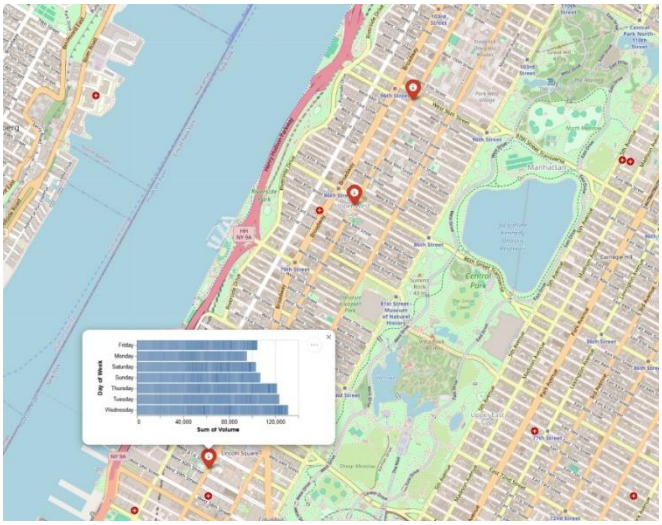


Figure 2 Data Visualization on Map

Technical Details

In the following will discuss some technical details of the code.

The following still uses *pyspark.sql.functions* package. make_date helps generate a connected date e.g. 2024-05-09 date_format helps generate the Day Of Week.

```
def add_day_of_week(df):
    df = df.withColumn("Date", make_date(col("Yr"), col("M"), col("D")))
    df = df.withColumn("DayOfWeek", date_format(col("Date"), "EEEE"))
    return df
```

The following uses groupby to group the rows with same value in columns. In this project, WktGeom, SegmentID, Direction should be same to make sure the geographical location, Road Segment ID in NYC DOT and the direction of the traffic is all the same. Then use orderBy to choose the top 5 stations.

```
sorted_df = spark_df.groupBy("WktGeom", "SegmentID", "Direction").count()
top_5_values = sorted_df.orderBy(desc("count")).limit(5)
```

The following uses agg.(max()) to find the maximum number in the column. Then use the filter to filter out the lastest year to treat as the live data stream and others the static data for the model.

```
Station1_lastest_year =
Station1_df.agg(max("Yr").alias("LatestYear")).collect()[0]["LatestYear"]
Station1_df.filter(col("Yr") < Station1_lastest_year).write.csv("Station1",
header=True, mode='overwrite')
Station1_df.filter(col("Yr") == Station1_lastest_year).write.csv("Station1_input",
header=True, mode='overwrite')
```

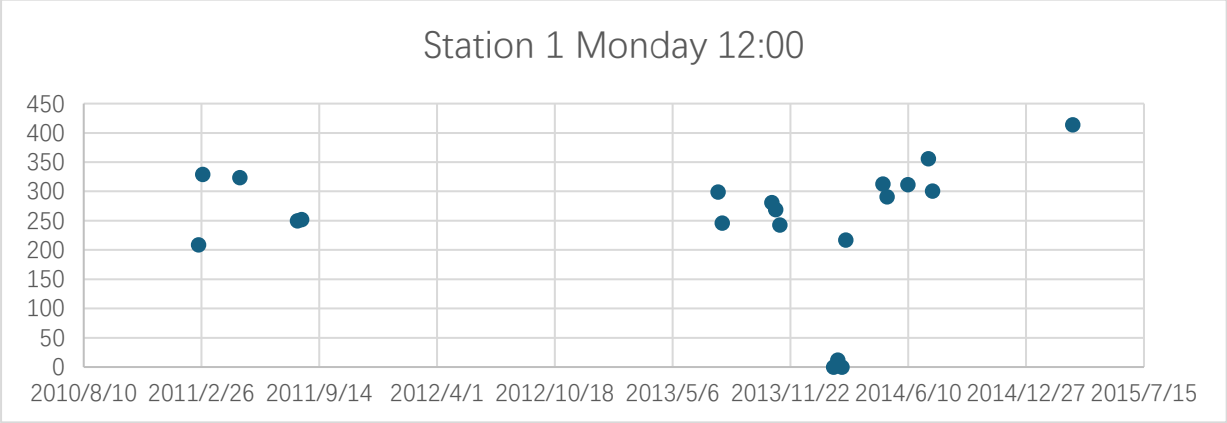
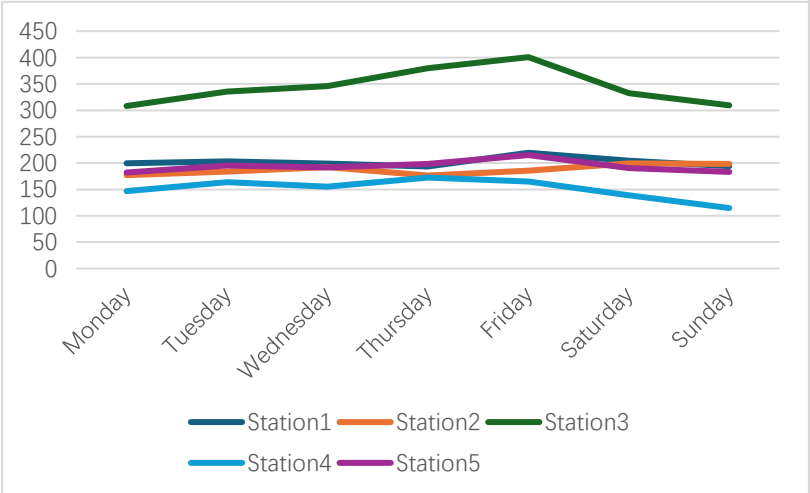
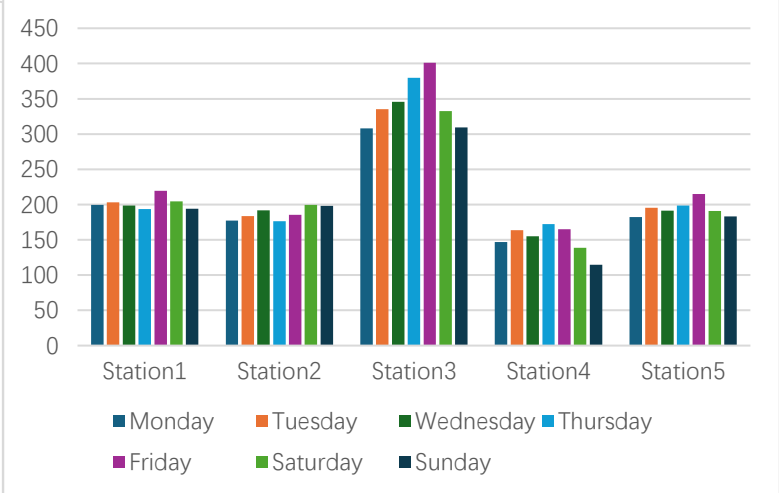


Figure 3 Static data of Station 1 Monday 12:00





To determine the abnormality, I used the empirical rule. As the traffic volume can be considered as normal distribution, using the empirical rule is very easy to calculate. In statistical testing, a statistical significance has a confidence level of two-sigma effect (95%), which is calculated by $\mu \pm 2\sigma$, while μ is mean, and σ is standard deviations.

```
vol_std =
matching_rows.agg(stddev("Vol").alias("VolumeStdDev")).collect()[0]["VolumeStdDev"]
vol_mean =
matching_rows.agg(mean("Vol").alias("VolumeMean")).collect()[0]["VolumeMean"]if
volume > vol_mean + 2 * vol_std or volume < vol_mean - 2 * vol_std:
    abnormal_counter += 1
    print("Anomaly Detected! Volume: ", volume, "Mean: ", vol_mean, "StdDev: ",\
    vol_std, "Anomaly Counter: ", abnormal_counter)
```

Streaming Processing

To start the streaming processing, I used query in PySpark. The input of the query needs to define the schema. In the following, most important function is .foreachBatch, its argument is a function that each batch will have to run the function with predefined batch_df, batch_id in the argument of the function. After starting the query, the program will wait for the folder to read incoming files.

```
schema = static_df.schema
streaming_df = spark.readStream.schema(schema).option("header",
"true").csv("Station4_input")

query = streaming_df.writeStream \
    .trigger(processingTime="10 seconds") \
    .outputMode("append") \
    .format("console") \
    .foreachBatch(anomaly_detection) \
    .option("maxFilesPerTrigger", 1) \
    .start()

query.awaitTermination()
```

Streaming Algorithm

The streaming algorithm implemented in the project is State Sharing. To optimize for space by avoiding unnecessary calculations, I defined a dictionary. The key of the dictionary is the connected string of Day of Week, Hour, and Minute. As the key number is not very large, the possibility of hash conflict would be very small. The benefit of using a dictionary is the search time is O(1), and the space is also reduced because the function will not calculate the mean and standard deviation for each row but first search in the dictionary if there is a past calculation.

```
dict_name = str(day_of_week)+str(hour)+str(minute)
        if dict_name not in vol_dict:
```

```
            matching_rows = static_df.filter((col("DayOfWeek") == day_of_week) &
(col("HH") == hour) & (col("MM") == minute))
            vol_std =
matching_rows.agg(stddev("Vol").alias("VolumeStdDev")).collect()[0]["VolumeStdDev"]
            vol_mean =
matching_rows.agg(mean("Vol").alias("VolumeMean")).collect()[0]["VolumeMean"]
            vol_dict[dict_name] = (vol_mean, vol_std)
```

Result Analysis

After calculating the anomaly, the anomaly percentage is shown below. Station 3 and 4 has relatively much higher anomaly percentage than Station 1, 2 and 5. The location of the station may be the main factor of the anomaly. Station 1, 2 and 5 are located on Amsterdam Ave, it is an normal avenue for daily use. There is no specific use for the avenue on certain days. While Station 3 is Long Island Expressway(I-485) and Station 4 is the entrance of Ed Koch Queensboro Bridge. So there could happen frequently lots of traffic or some maintenance. So the traffic volume could vary significantly and cause an anomaly value.

	Station1	Station2	Station3	Station4	Station5
Anomaly	0	2	1008	1268	1
Normal	768	1342	3420	2811	671
Anomaly Percentage	0%	0%	23%	31%	0%

Table 2 Anomaly Percentage

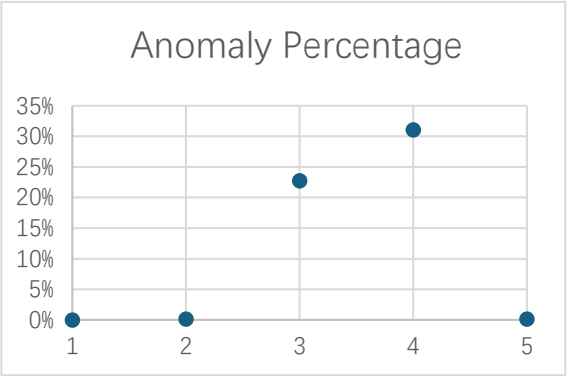


Figure 4 Anomaly Percentage

Future Work

Some machine learning algorithms could be implemented for this project's future. For example, SVM could be used to check the anomaly. As there is only one input to check each time, having the maximum margin to find the hyperplane could help achieve better anomaly accuracy. Another future of this project is gathering more data. I did not find APIs for precipitation history in the past 10 years. If there is more environmental or social data, the data can be analyzed from more points of view.

Parallel computation can be another method to improve the data processing speed. The code can implement several threads with the spark by introducing the threading package in Python. After setting Master, AppName, and the scheduler, the thread can be created by defining the target as the function that needs to operate on each dataframe. Parallelizing computations can be functioned as fission in streaming algorithms. With more data frame processing at the same time, the output will increase from one station to many stations. In addition, if the station data is still input in order, introducing threads can implement load shedding. With a well-designed scheduler, the data can be split into different threads to process and merge together in the end. This could further increase the processing efficiency of the code.