Analyzing Stock Market Values

LE Thi Hoa - 12310380
TRAN Hai Linh - 12310487

November 2023

Link code: https://github.com/tranhailinh97/SparkStreamingStocks/tree/master

I. Introduction

This project aims to analyze streaming data related to the stock market. We have data for stocks with 619,039 rows. The project uses Spark Streaming to analyze the data and Kafka to read the data.

II. Pre-Processing

We use Kafka to read the file and ingest the data into Kafka with the schema (name, price, timestamp). The timestamp uses the current one. This below function to process timestamp:

```
[10]: from datetime import datetime
        #2023-10-13T08:16:13Z
      def construct_stock(row):
          time_stamp = time.time()
          date_time = datetime.fromtimestamp(time_stamp)
          str_date_time = date_time.strftime("%Y-%m-%dT%H:%M:%SZ") #"%d-%m-%Y, %H:%M:%S"
          stock = {"name": row[6],
                   "price": float(row[2]),
                   "timestamp":str_date_time
                   }
          return stock
```

Initialize a SparkSession to work with Spark Streaming and create a schema for the data.

```
schema = StructType(
        [
          StructField("name", StringType(), False),
          StructField("price", DoubleType(), False),
          StructField("timestamp", TimestampType(), False),
        ]
    )
```

Then, use Spark to read data from Kafka with the topic "stock."

```
[3]: kafka_server = "kafka1:9092"
from pyspark.sql.functions import from_json

lines = (spark.readStream                        # Get the DataStreamReader
  .format("kafka")                               # Specify the source format as "kafka"
  .option("kafka.bootstrap.servers", kafka_server) # Configure the Kafka server name and port
  .option("subscribe", "stock")                     # Subscribe to the "en" Kafka topic
  .option("startingOffsets", "earliest")         # The start point when a query is started
  .option("maxOffsetsPerTrigger", 100)           # Rate limit on max offsets per trigger interval
  .load()
  .select(from_json(col("value").cast("string"), schema).alias("parsed_value"))
# Load the DataFrame
)
df = lines.select("parsed_value.*")
```

III. Data Processing

1. The N most valuable stocks in each windows

For the task of obtaining the N most valuable stocks in each window, we operate with a time window of 5 minutes and set a watermark of 30 seconds. We perform a groupBy operation based on each 5-minute window and the stock's name. Subsequently, we compute the maximum price for each stock within each window. For each stock within each window, we consider selecting the maximum price as a suitable choice because, within a window, a stock may have multiple price updates, and the highest price is the most valuable one.

Next, we sort the data for each window in ascending order of time between windows, and within each window, we sort the data in descending order of price and select the top N most valuable stocks.

We save the processed data into memory with the name "MostValuableStock." We can access "MostValuableStock" to display the results and perform basic visualizations on the outcome.

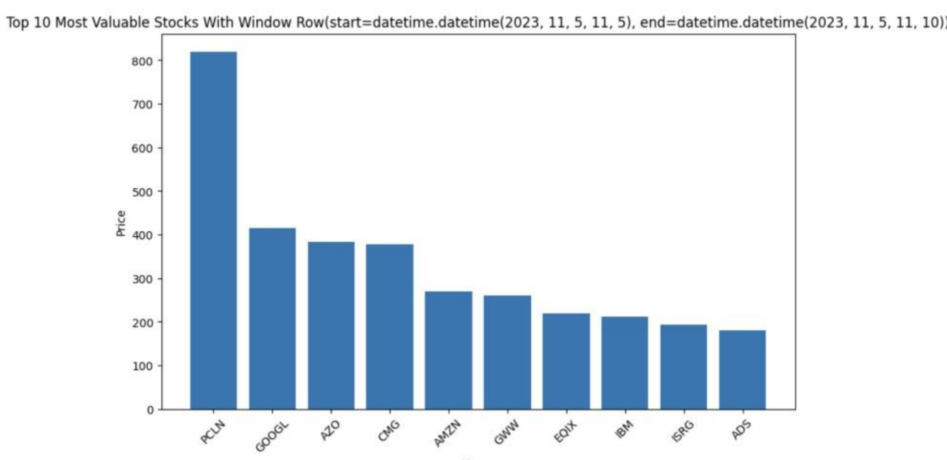
Select the N most valuable stocks in a window

```
[4]: from pyspark.sql.functions import window, col

N = 10

windowedDF = df \
        .withWatermark("timestamp", "30 seconds") \
        .groupBy(window("timestamp", "5 minutes"), "name") \
        .agg({"price": "max"})

top_stocks = windowedDF.orderBy(col("window").asc(),  col("max(price)").desc().limit(10))
top_stocks.createOrReplaceTempView("top_stocks")


# Save results
query = (top_stocks.writeStream
         .outputMode("complete")
         .format("memory")
         .queryName("MostValuableStock")
         .option("truncate", False)
         .start())
```

```
In [5]: from matplotlib import pyplot as plt

data = spark.sql("SELECT * FROM MostValuableStock")
data.show(truncate=False)

name_list = [row["name"] for row in data.collect()]
price_list = [row["max(price)"] for row in data.collect()]
window_time = data.select("window").first()["window"]

# Vẽ biểu đồ
plt.figure(figsize=(10, 6))
plt.bar(name_list, price_list)
plt.xlabel("Name")
plt.ylabel("Price")
plt.title(f"Top 10 Most Valuable Stocks With Window {window_time}")
plt.xticks(rotation=45)

plt.show()
```

=> Result: We select the 10 most valuable stocks

```
+------------------------------------------+-----+----------+
|window                                    |name |max(price)|
+------------------------------------------+-----+----------+
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|PCLN |819.98    |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|GOOGL|414.2338  |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|AZO  |384.01    |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|CMG  |378.4     |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|AMZN |269.98    |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|GWW  |261.34    |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|EQIX |219.94    |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|IBM  |211.98    |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|ISRG |192.6465  |
|[2023-11-05 11:05:00, 2023-11-05 11:10:00]|ADS  |180.39    |
+------------------------------------------+-----+----------+
```



Top 10 most valuable stock

2. Select the stocks that lost value between windows

For the task of "Select the stocks that lost value between two windows," our main idea is to compare the average price of each stock within a window with its average price in the previous window.

To achieve this, we have created a function called process_batch(df, epoch_id) to process each batch (using foreachBatch).

```python
[5]: from pyspark.sql.functions import lag, window

# The function processes each batch of data
def process_batch(df, epoch_id):

    window_spec = Window.partitionBy("name").orderBy("window")

    # Create a column "previous_window" and "previous_price" using the lag function
    df = df.withColumn("previous_price", lag("avg(price)").over(window_spec))
    df = df.withColumn("previous_window", lag("window").over(window_spec))
    df = df.filter(df["previous_price"] > df["avg(price)"])

    # Show data
    df.show(truncate=False)


# Apply a time window to the data with a watermark of 30 seconds
# Group the data by a 5-minute window and the stock name
# Calculate the average price within each window for each stock

windowedDF_2 = df \
        .withWatermark("timestamp", "30 seconds") \
        .groupBy(window("timestamp", "5 minutes"), "name") \
        .agg({"price": "avg"})

# Order the results by average price in descending order
lost_value_stocks = windowedDF_2.orderBy("avg(price)", ascending=False)


# Apply function process_batch, save and show result
query_2 = (lost_value_stocks.writeStream
           .outputMode("complete")
           .format("memory")
           .queryName("TheStocksThatLostValue1")
           .option("truncate", False)
           .foreachBatch(process_batch)
           .start())

query_2.awaitTermination()
```

We apply a time window to the data with a watermark of 30 seconds. Sau đó, group the data by a 5-minute window and the stock name và calculate the average price within each window for each stock.

For process_batch - a function that processes each batch of data in a Spark Structured Streaming job. The goal of this code is to identify stocks that have experienced a loss in value between consecutive time windows and display the relevant information about these stocks for further analysis. The lag function is used to access the previous window's data and compare it with the current window's data to detect value losses.

For each batch of data, the process_batch function should be applied. This function processes the batch and identifies stocks that have lost value.

=> Result:

window	name	avg(price)	previous_price	previous_window
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	AVY	42.56	43.17	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	CLX	84.975	86.4	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	COP	60.8	61.82	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	LB	50.34	51.48	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	XYL	26.63	27.835	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	CAG	33.64	35.64	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	HCP	46.78	50.394999999999996	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	APA	81.94	85.82	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	HD	75.43	77.0	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	SLB	76.88	77.04	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	GS	152.77	156.32333333333332	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	AEE	35.31	36.25	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	KO	41.07	43.05	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	SBAC	74.28	80.1325	[2023-11-05 11:05:00, 2023-11-05 11:10:00]
[2023-11-05 11:10:00, 2023-11-05 11:15:00]	BEN	46.235	47.025	[2023-11-05 11:05:00, 2023-11-05 11:10:00]

Table 1: *The stocks that lost value between windows*

Table 1 hiện thị the stocks that lost value between window. In this table, you can observe that stocks with decreasing values between windows, for instance, AVY had a price of 43.17 in the previous window, but in the current window, it has reduced to 42.56

3. Find the stocks that gained the most between windows

Similar to the steps performed in task 2, in this task, we are looking for stocks that have increased in value between windows.

We filter out stocks with an average price in the current window greater than their average price in the previous window. This means that they have experienced a price increase compared to the previous time window.

```python
#remember you can register another stream

from pyspark.sql.functions import lag, window

# The function processes each batch of data
def process_batch(df, epoch_id):

    window_spec = Window.partitionBy("name").orderBy("window")

    # Create a column "previous_window" and "previous_price" using the lag function
    df = df.withColumn("previous_price", lag("avg(price)").over(window_spec))
    df = df.withColumn("previous_window", lag("window").over(window_spec))
    df = df.filter(df["previous_price"] < df["avg(price)"])

    # Show
    df.show(truncate=False)


# Apply a time window to the data with a watermark of 30 seconds
# Group the data by a 5-minute window and the stock name
# Calculate the average price within each window for each stock
windowedDF_3 = df \
        .withWatermark("timestamp", "30 seconds") \
        .groupBy(window("timestamp", "5 minutes"), "name") \
        .agg({"price": "avg"})

gained_value_stocks = windowedDF_3.orderBy("avg(price)", ascending=False)


# Apply process_batch function, save and show result
query= (gained_value_stocks.writeStream
           .outputMode("complete")
           .format("memory")
           .queryName("GainedValueStocks1")
           .option("truncate", False)
           .foreachBatch(process_batch)
           .start())

query.awaitTermination()
```

=> Result:

window	name	avg(price)	previous_price	previous_window
[2023-11-05 11:15:00, 2023-11-05 11:20:00]	ALXN	180.2	108.02	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:25:00, 2023-11-05 11:30:00]	ALXN	186.99	164.66	[2023-11-05 11:20:00, 2023-11-05 11:25:00]
[2023-11-05 11:30:00, 2023-11-05 11:35:00]	ALXN	201.66995	186.99	[2023-11-05 11:25:00, 2023-11-05 11:30:00]
[2023-11-05 11:20:00, 2023-11-05 11:25:00]	GIS	50.92	48.6	[2023-11-05 11:15:00, 2023-11-05 11:20:00]
[2023-11-05 11:30:00, 2023-11-05 11:35:00]	GIS	57.61	50.92	[2023-11-05 11:20:00, 2023-11-05 11:25:00]
[2023-11-05 11:15:00, 2023-11-05 11:20:00]	K	67.63	65.865	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:30:00, 2023-11-05 11:35:00]	K	66.81	62.1798	[2023-11-05 11:25:00, 2023-11-05 11:30:00]
[2023-11-05 11:15:00, 2023-11-05 11:20:00]	LEN	38.958349999999996	33.89705	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:25:00, 2023-11-05 11:30:00]	LEN	49.676500000000004	35.5196	[2023-11-05 11:20:00, 2023-11-05 11:25:00]
[2023-11-05 11:15:00, 2023-11-05 11:20:00]	SPGI	77.12	60.16	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:20:00, 2023-11-05 11:25:00]	SPGI	86.69	77.12	[2023-11-05 11:15:00, 2023-11-05 11:20:00]
[2023-11-05 11:25:00, 2023-11-05 11:30:00]	SPGI	90.47	86.69	[2023-11-05 11:20:00, 2023-11-05 11:25:00]
[2023-11-05 11:30:00, 2023-11-05 11:35:00]	SPGI	97.46	90.47	[2023-11-05 11:25:00, 2023-11-05 11:30:00]
[2023-11-05 11:15:00, 2023-11-05 11:20:00]	AIV	29.98	28.738	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:25:00, 2023-11-05 11:30:00]	AIV	37.57	29.98	[2023-11-05 11:15:00, 2023-11-05 11:20:00]
[2023-11-05 11:35:00, 2023-11-05 11:40:00]	AIV	40.975	37.57	[2023-11-05 11:25:00, 2023-11-05 11:30:00]
[2023-11-05 11:25:00, 2023-11-05 11:30:00]	AVY	62.78	42.56	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:35:00, 2023-11-05 11:40:00]	AVY	68.55000000000001	59.358	[2023-11-05 11:30:00, 2023-11-05 11:35:00]
[2023-11-05 11:15:00, 2023-11-05 11:20:00]	BF.B	41.9325	35.775	[2023-11-05 11:10:00, 2023-11-05 11:15:00]
[2023-11-05 11:25:00, 2023-11-05 11:30:00]	BF.B	45.39163333333334	41.9325	[2023-11-05 11:15:00, 2023-11-05 11:20:00]

Table 2: the stocks that gained the most between windows

4. Implement a control that checks if a stock does not lose too much value in a period of time

We apply a time window and calculate the price change during that period. Here, we're using a 10-minute window with a 1-minute watermark to track changes in the stock's price over time. We filter the data to select rows with the specified stock name. (example: FITB)

We then calculate the percentage price change between the first and last prices within each time window.

We filter and select rows where the last price is lower than the first price and where the percentage price change exceeds the defined loss threshold.

This code helps monitor and identify cases where a specific stock's price decreases beyond a predefined threshold within the specified time window.

```python
from pyspark.sql.functions import window, col
from pyspark.sql import functions as F

# Set the price loss threshold and the name of the stock you are interested in
loss_threshold = 0.05 #5%
# Replace with the name of the stock ticker you want to track
stock_name = "FITB"

# Apply a time window and calculate the price change during that period
windowedDF = df \
    .withWatermark("timestamp", "1 minutes") \
    .filter(col("name") == stock_name) \
    .groupBy(window("timestamp", "10 minutes")) \
    .agg(
        F.expr("min_by(price, timestamp)").alias('first_price'),
        F.expr("max_by(price, timestamp)").alias('last_price'),
        F.max('timestamp').alias('lastTimeStamp'),
        F.min('timestamp').alias('firstTimeStamp')
        )

# Check if the price change exceeds the threshold
result_df = windowedDF.withColumn("percent_value_lost", ((col("first_price") - col("last_price"))/col("first_price")))
result_final = result_df.filter((col("last_price") < col("first_price")) &
                                (col("percent_value_lost") > loss_threshold))

# Save and show
query = (result_final.writeStream
        .outputMode("complete")
        .format("console")
        .start())
```

=> Result:

```
Batch: 83
-------------------------------------------
+--------------------+-----------+----------+-------------------+-------------------+-------------------+
|              window|first_price|last_price|      lastTimeStamp|     firstTimeStamp| percent_value_lost|
+--------------------+-----------+----------+-------------------+-------------------+-------------------+
|[2023-11-05 12:10...|      20.67|     18.31|2023-11-05 12:16:44|2023-11-05 12:15:14|0.11417513304305771|
+--------------------+-----------+----------+-------------------+-------------------+-------------------+
```

Table 3: The FITB stock lost beyond the threshold.

5. Compute how your asset changes with the fluctuation of the market

We have created a DataFrame called stock_portfolio, which represents the stocks you own along with the amount of each stock. We then join this DataFrame with the df DataFrame (which contains stock price data) using the common column "name."

After joining, we calculate the asset value for each stock by multiplying the "amount_of_stocks_owned" by the "price" for each stock. This is done to compute the asset value for each stock you own.

Finally, we calculate the total asset value by summing up the asset values of all the stocks in your portfolio using the selectExpr function.

The result of this computation is written to a streaming query named "AssetValue1" with an output mode set to "update," which means the result will be updated as new data arrives.

```
# The stocks that I own, along with the amount of each stock
stock_portfolio = spark.createDataFrame([
    ('ISRG', 10.0),
    ('BBT', 15.0),
    ('FITB', 15.0),
    ('ZION', 5.0),
    ('CL', 15.0),
    ('KR', 90.0),
    ('WEC', 80.0),
    ('VRTX', 40.0),
    ('PNW', 15.0),
    ('DTE', 16.0)], ["name", "amount_of_stocks_owned"])

stock_portfolio.show()

# Join two DataFrames based on the stock name.
portfolio_with_prices = stock_portfolio.join(df, "name", "inner")

# Compute asset
portfolio_with_prices = portfolio_with_prices.withColumn("asset_value", col("amount_of_stocks_owned") * col("price"))

# Compute sum asset
total_asset_value = portfolio_with_prices.selectExpr("sum(asset_value) as total_asset_value")


query_5 = (total_asset_value.writeStream
        .outputMode("update")
        .format("memory")
        .queryName("AssetValue1").start())
```

=> Result:

```python
from matplotlib import pyplot as plt
import mplcursors

test = spark.sql("SELECT * FROM AssetValue1")
test = test.toPandas()
# Vẽ biểu đồ đường
plt.plot(test.index, test["total_asset_value"], marker='o', linestyle='-', color='b')
plt.ylabel("Total Asset Value")
```

```
Text(0, 0.5, 'Total Asset Value')
```

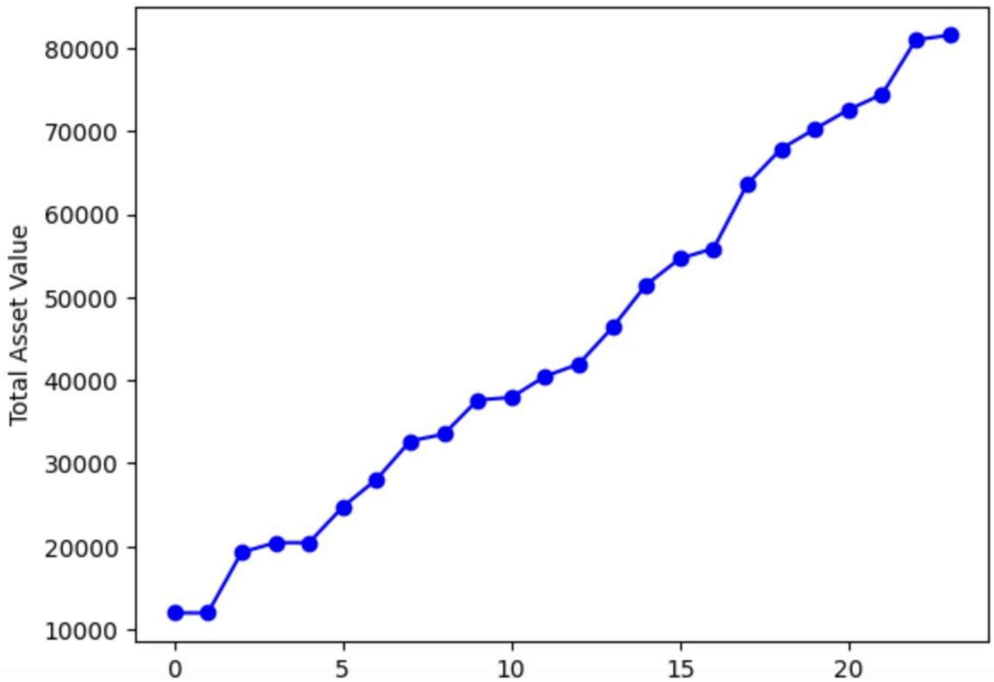
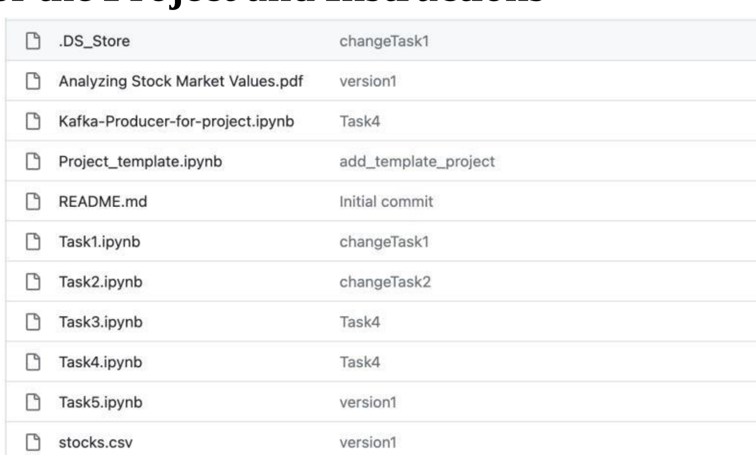


Figure 1: The change in total asset value.

IV. Structure of the Project and Instructions



.DS_Store	changeTask1
Analyzing Stock Market Values.pdf	version1
Kafka-Producer-for-project.ipynb	Task4
Project_template.ipynb	add_template_project
README.md	Initial commit
Task1.ipynb	changeTask1
Task2.ipynb	changeTask2
Task3.ipynb	Task4
Task4.ipynb	Task4
Task5.ipynb	version1
stocks.csv	version1

Figure 2: Structure of the Project

The project consists of 7 code files, including:

- Kafka-Producer-for-project.ipynb
- Project_template.ipynb
- Task1.ipynb
- Task2.ipynb
- Task3.ipynb
- Task4.ipynb
- Task5.ipynb

In the Template_Project file, we have implemented the code for all 5 tasks.

We have also divided each task into a separate code file for easier tracking and debugging.

To execute this project, you should start by running the Kafka_project file to read data from the stock.csv file. Then, you can run each cell in the Template_Project file, or run each task separately by executing the corresponding individual files.

Make sure to install Kafka and Spark to run the code.

V. Conclusion

In this project, we have conducted an analysis of 5 tasks related to stock market analysis. In the future, we want to enhance the quality of analysis and focus on improving visualization.