Part 2

Requirements

Our data was retrieved from two sources. We got the tweets using Twitter’s API and tweepy. The stock price data came from Polygon.io API. Many tweets our twitter API rules retrieved did not have a stock ticker in the text of the tweet. Instead, it was replying or otherwise participating in a conversation about a stock. We needed to include the text of the tweets in the conversation with the text of the tweet we matched to identify which stock a tweet was talking about.

We also happened to match tweets that did have a stock hashtag in them, such as “#EBAY”, but many tweets with that hashtag are not talking about eBay stock, but are talking about item listings.

Design

There were a few steps between retrieving the data, and getting the information we wanted from it. Those steps were:

1. Identify which stocks a tweet is talking about.
2. Extract hashtags
3. Remove irrelevant columns
4. Filter

After filtering the data, we have just the rows and columns we were interested in. Much of the data displayed on the webpage is related to counting. We’re showing how often people are tweeting about stocks by the hour, by the day, and which hashtags people are using in discussions about those stocks. Our tables must have a single stock ticker as a value per row, so we can make queries on those tickers. To make counting those easier, we needed one ticker and one hashtag per row.

Implementation

The technologies we used in Part 2 were Pyspark and MySQL. We used Pyspark to read the multiple json files, and to generate most of the tables you see represented on our webpage. Those tables were then stored in a MySQL database.

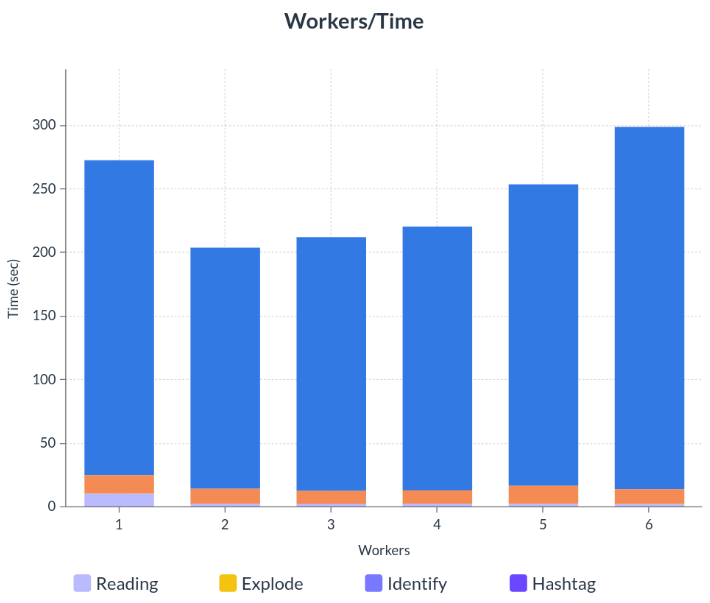
For steps 1 and 2 in the design section above, we used regex to match patterns of the cashtags and hashtags of the 102 stock tickers we collected to identify the tweets. Then we matched hashtags that did not match hashtags for the stocks to extract the hashtags. After identifying the tweets and hashtags in those tweets, we then used Pyspark SQL to select only the columns we needed. Those columns were timestamp, tweet ID, tweet text, stock tickers, and hashtags.

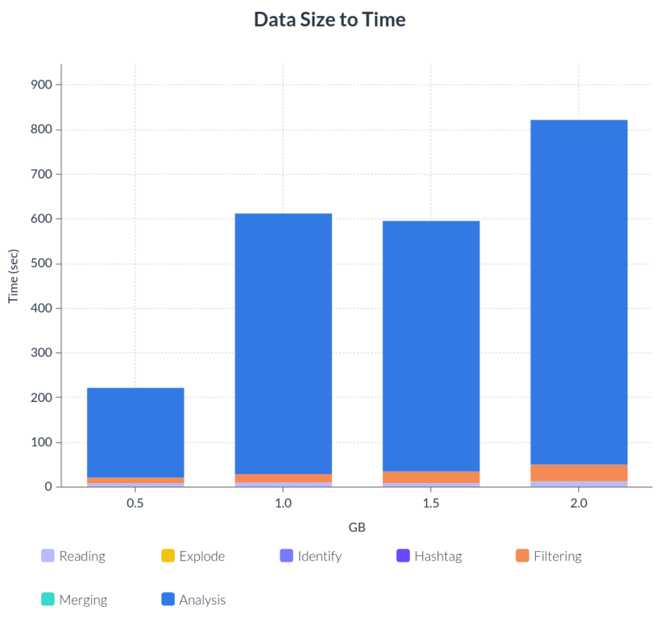
After removing most columns from the tweets dataframes, we then filtered the tweets dataframe containing stock discussion we matched with a hashtag, not a cashtag. One example of filtering is eBay. Most tweets containing #EBAY were not about eBay stock, but automated posts about listings placed on eBay. To remove these tweets, we dropped columns where the tickers column contains “EBAY” and the text starts with “Check out.”

Once the tweets hashtag dataframe was filtered, we combined it with the tweets cashtag dataframe. We then used the explode_outer function on the tickers and tags columns to help us count the occurrences of these tickers and tags. For most of the tables we display on the webpage, we create those tables using SQL count and group by on different columns. For the 3D graph seen in part 3, we use a rank over partition to get the most common hashtag used with a stock for every hour.

Evaluation

Towards the beginning of this part, we tried timing the reading of the tweets only. The time it took to read the data was not that long, so any trends we saw in time was likely due to caching. When timing more of our Pyspark operations, the time/worker relationship was more consistent. The following Workers/Time graph was run on 0.5GB of data, and the Data Size to Time graph was run with 2 workers.





Screenshots/Snippets

```
   popular_tags_hourly_count = spark.sql(
       "select * from (select *, row_number() over (PARTITION BY t_hour ORDER BY cnt desc) rank from tags_hourly_count) tmp where rank<=5 order by t_hour desc, rank")
```

```
   time_tickers_explode_df = time_tickers_tags_df.withColumn(
       "tickers", explode_outer("tickers"))

   time_tickers_tags_explode_df = time_tickers_explode_df.withColumn(
       "tags", explode_outer("tags"))
```

Contribution

Garrett: Helped identify the #MU spam and helped create criterion to filter this stock ticker. Analyzed frequencies of hashtags, which excludes the ticker itself. Joined financial data of stocks from polygon with these frequencies.
HanYu: No contribution
Wyatt: Used regex to identify the stocks a tweet is discussing. Made that function into a Pyspark UDF to use on every row of the tweets dataframes. Filtered tweets_hashtag_df. Got timing data with different numbers of workers and data sizes.
Zeyu:Use pyspark to analyze the stock’s data(e.g.stock's hot words, hourly, daily,retweets) collected by our part1 and store them in the database. Filter out some hashtags and cashtags which are not stock

TA Strike Impact

In part 2 of the lab, we were wondering why varying the number of spark-workers in our for-loop produced the same result as fixing the number. We thought that this may be due to some caching, but when we turned off caching not much changed.

We also had an issue with writing to our DB, where one column of our data was completely missing, which took a couple of days to debug.

Part 3

Requirements

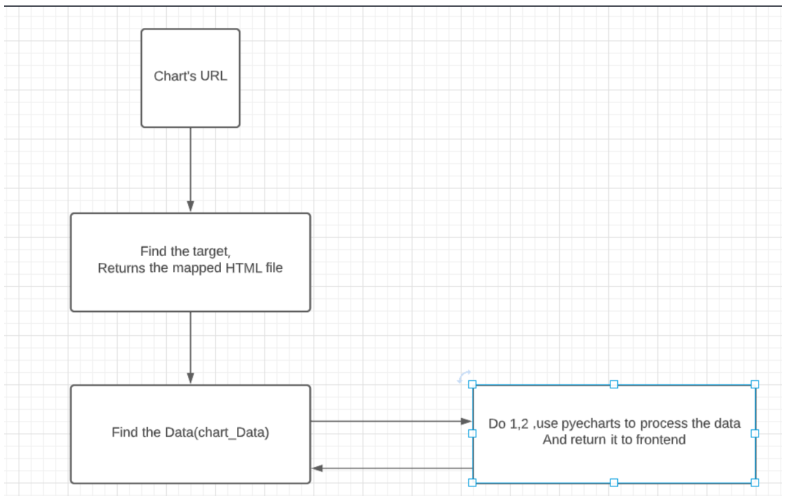
We need to create an interactive web interface to explore our analyzed data. This application should be high-speed by precomputing all values and storing them in a Database.

Design

The user can input some information that they are interested in and get back the relevant information via forms. The user also can directly manipulate the perspective or certain attributes of our graphs.

Implementation

We used the pyecharts and plotly libraries to graphically display our data which we acquired from our DB through the mysql connector. Additionally, we used CSS(Bootstrap) Bootstrap is a free, open source front-end development framework for the creation of websites and web apps. And html to format our charts and graphs in a visually appealing way.


Chart's URL
Find the target,
Returns the mapped HTML file
Find the Data(chart_Data)
Do 1,2 ,use pyecharts to process the data
And return it to frontend

Our framework was Django and we created one view for each graph or chart. For the plotly chart, we rendered it directly in home-page template by using plotly's to_html(). For charts1 through 4 we used href to redirect to the url that contains the charts view and back to the home-page.

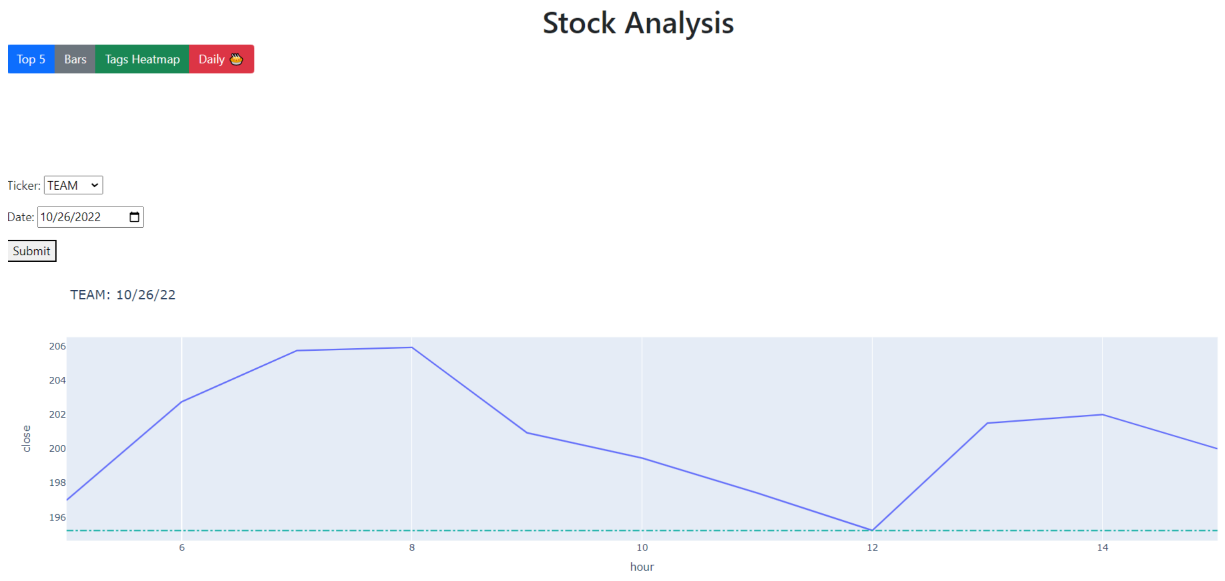
Evaluation

We found our website to be pretty interactive aside from chart1, which just displayed frequencies of tweets for each stock. A potential improvement would be to give the user more freedom by allowing them to give tags in addition to tickers and time-frames. This would allow for a backwards search of stocks.

We found that our charts were mostly high-speed, but when we used larger datasets, some of the graphs would take a couple seconds longer to load. A way to increase performance would be to statically render our data instead of having the server render it. The downside of this for our application is that we would not be able to update it with more recent data. Also, for the plotly graph, we would have to create many graphs, but the user is probably not going to view every graph.

Screenshots

First page of our website:



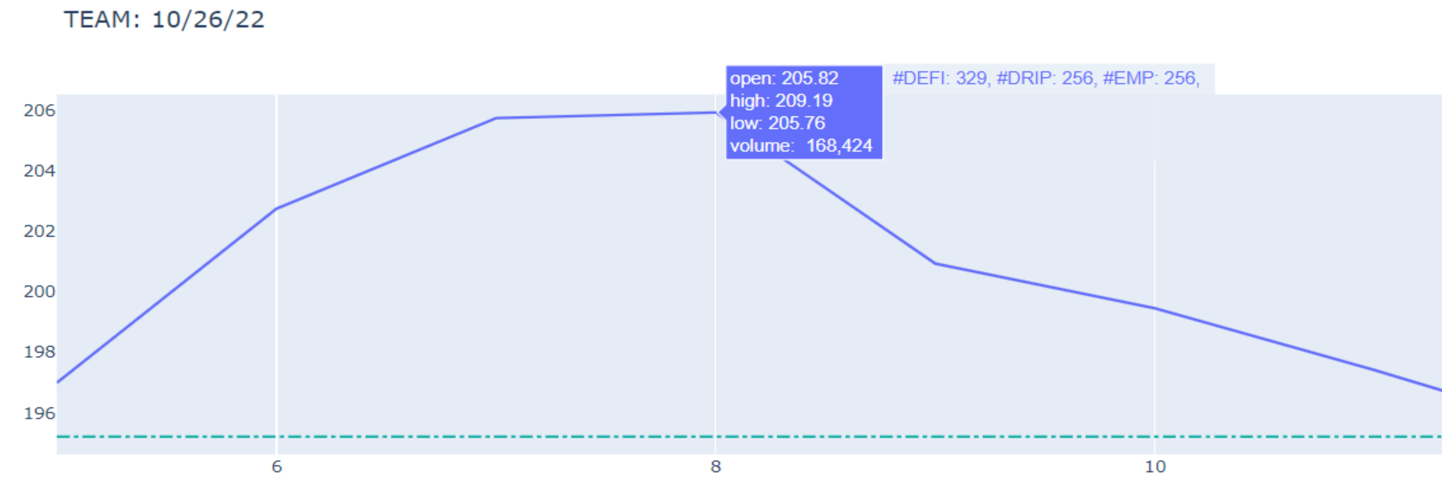
Ticker: TEAM

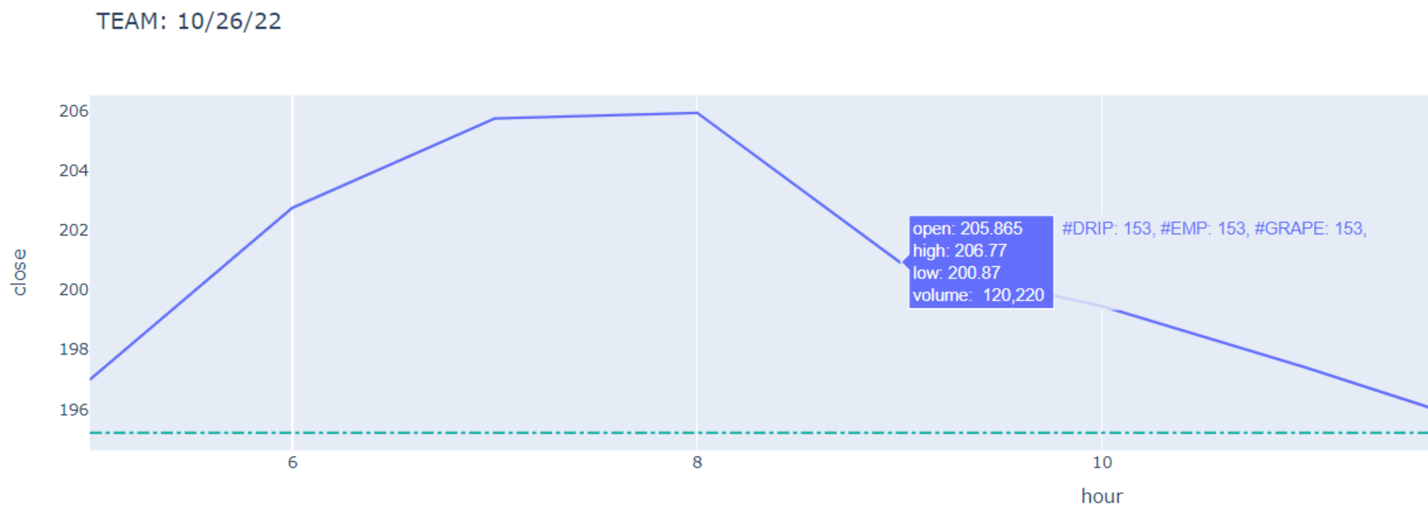
Date: 10/26/2022

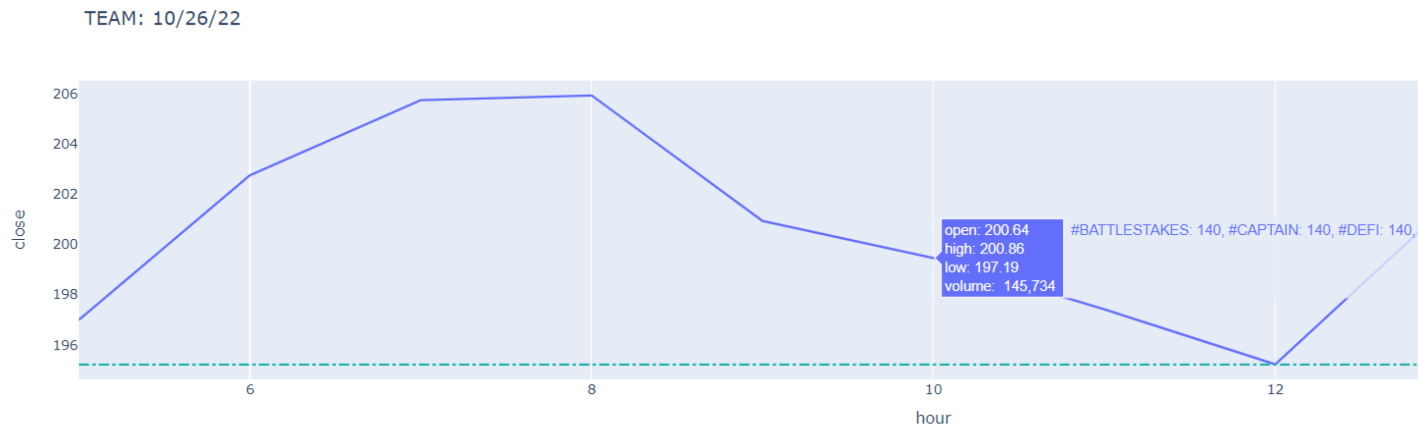
TEAM: 10/26/22

Examples of graph (hover over point to interact):

The user can input a date and a stock ticker to get the hourly financial information and the most used hashtags.







Example of graph (hover over point to interact):

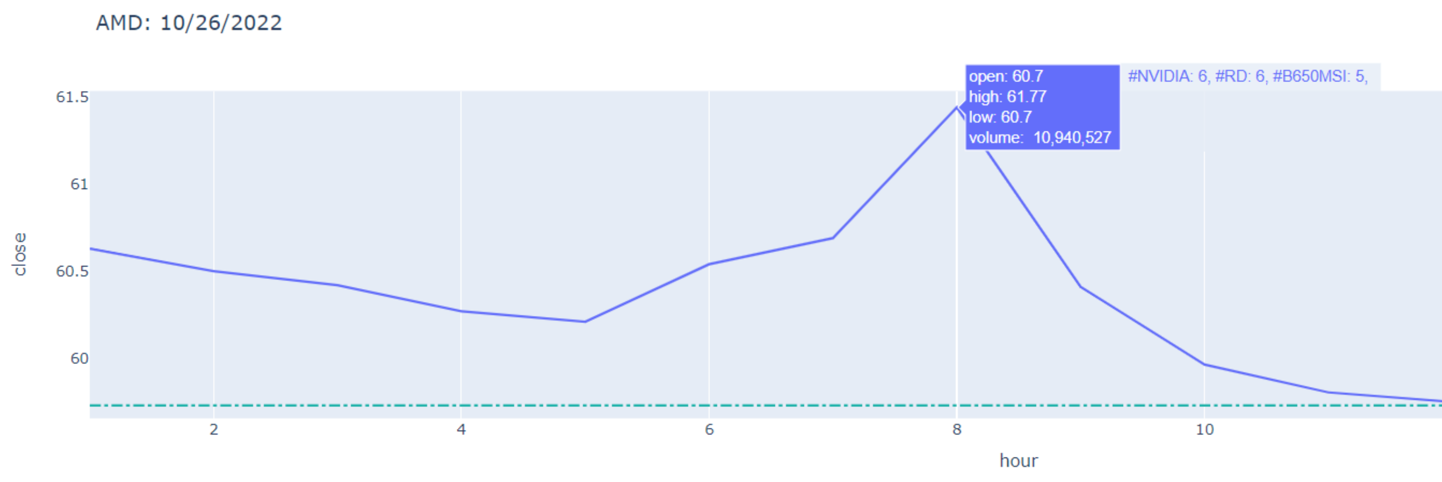


Chart1:
This chart shows the total frequencies of tweets for the top 5 stocks throughout the whole data collection process.

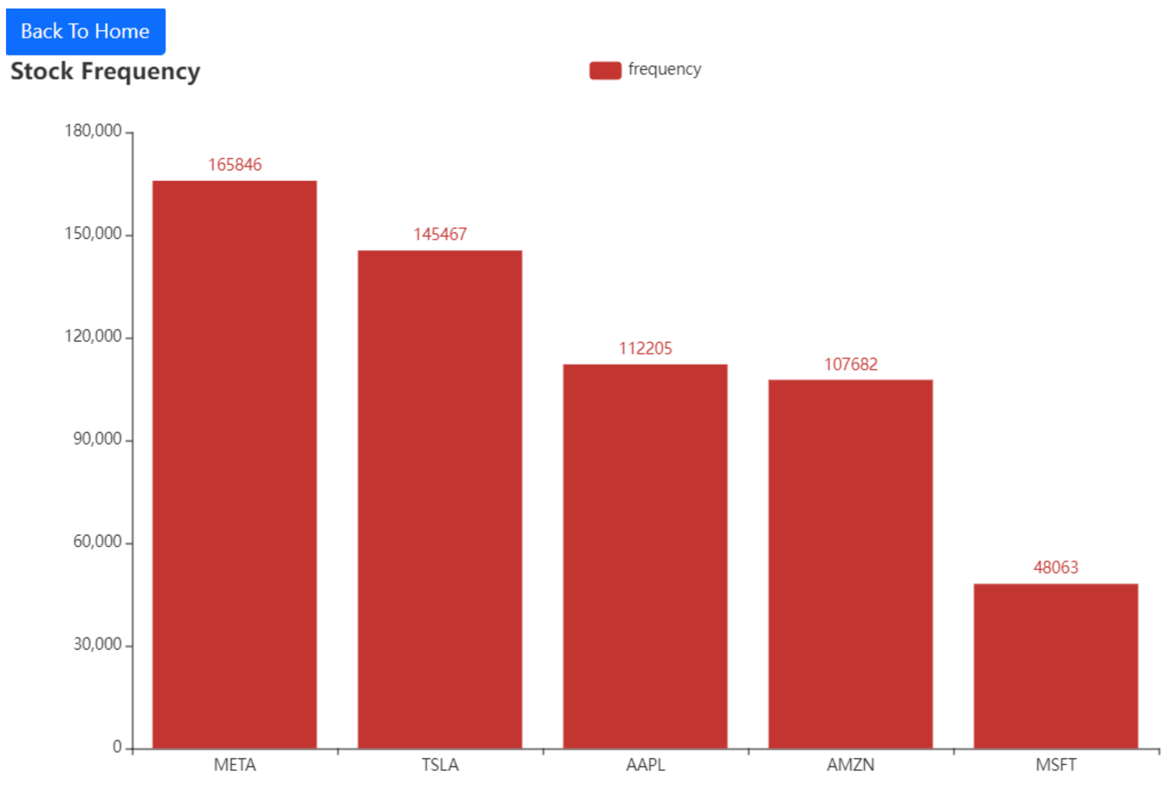
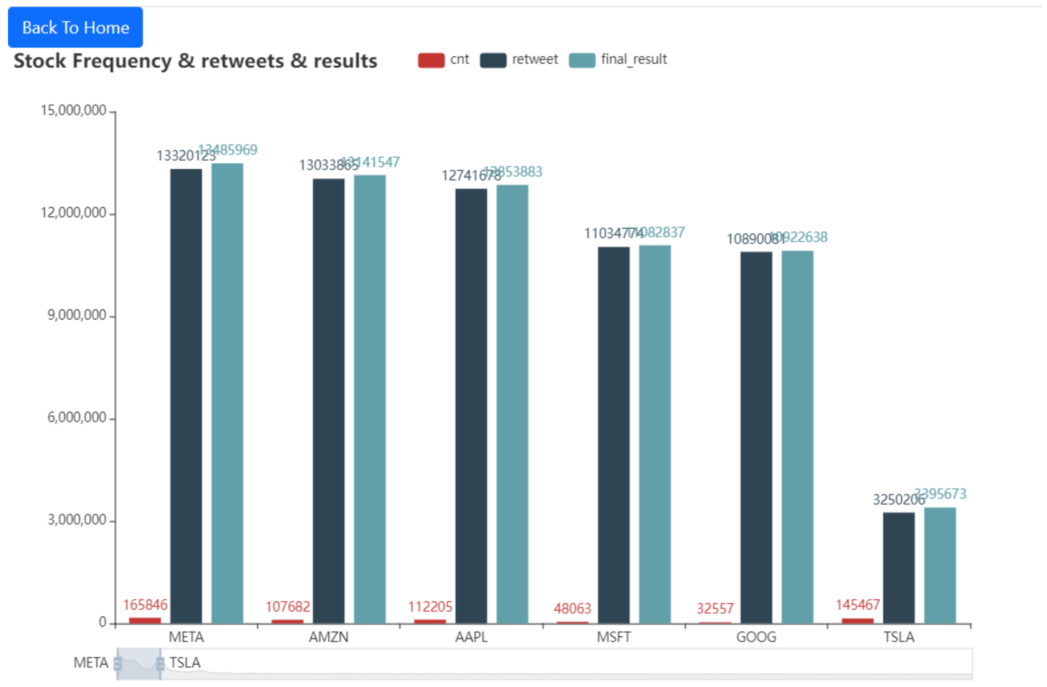
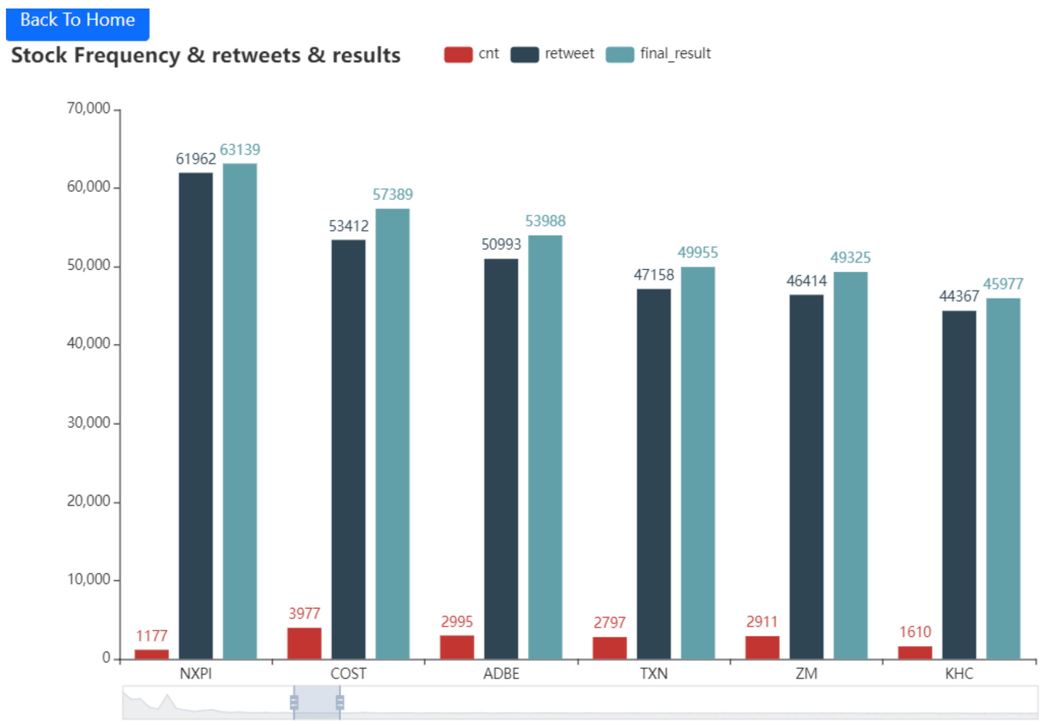


Chart2 (can scroll through the different stocks):

This chart visibility separates the number of actual tweets versus retweets for each stock.





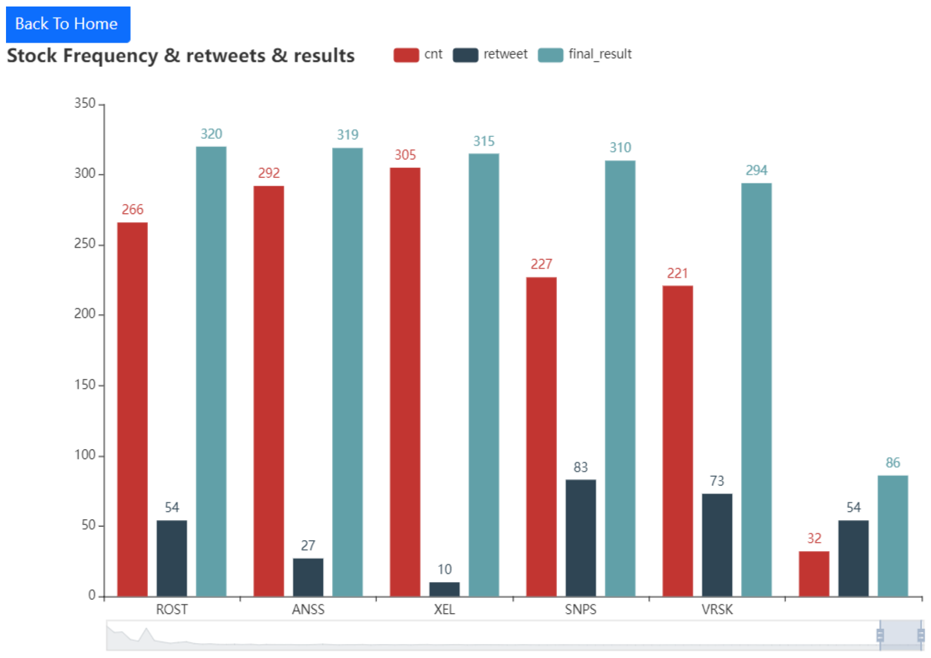
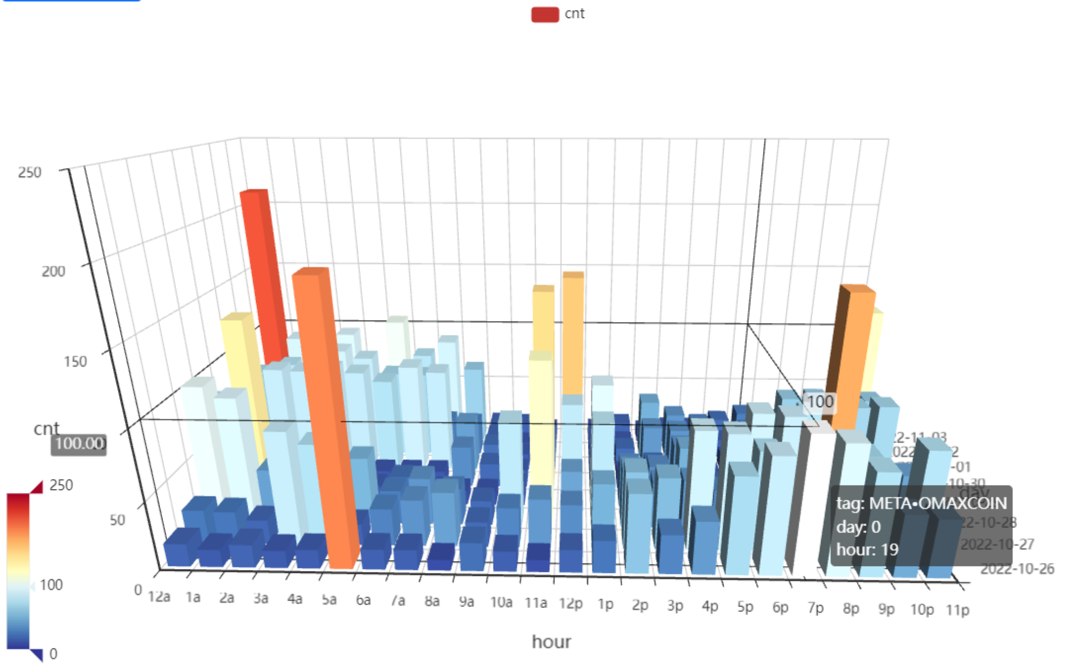
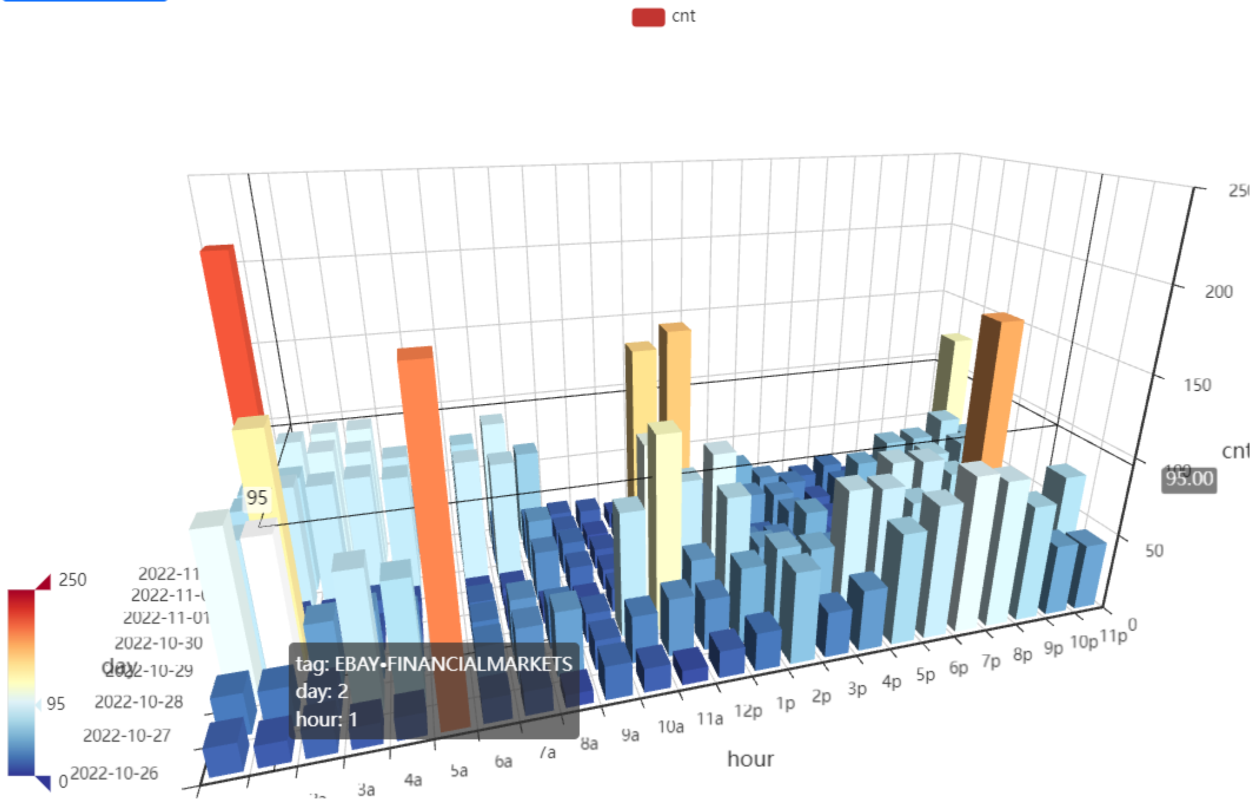


Chart3 (can rotate model and hover over rectangles):
This chart helps the user visualize what tags are used for each stock, hour, date combination.

Back To Home



Back To Home



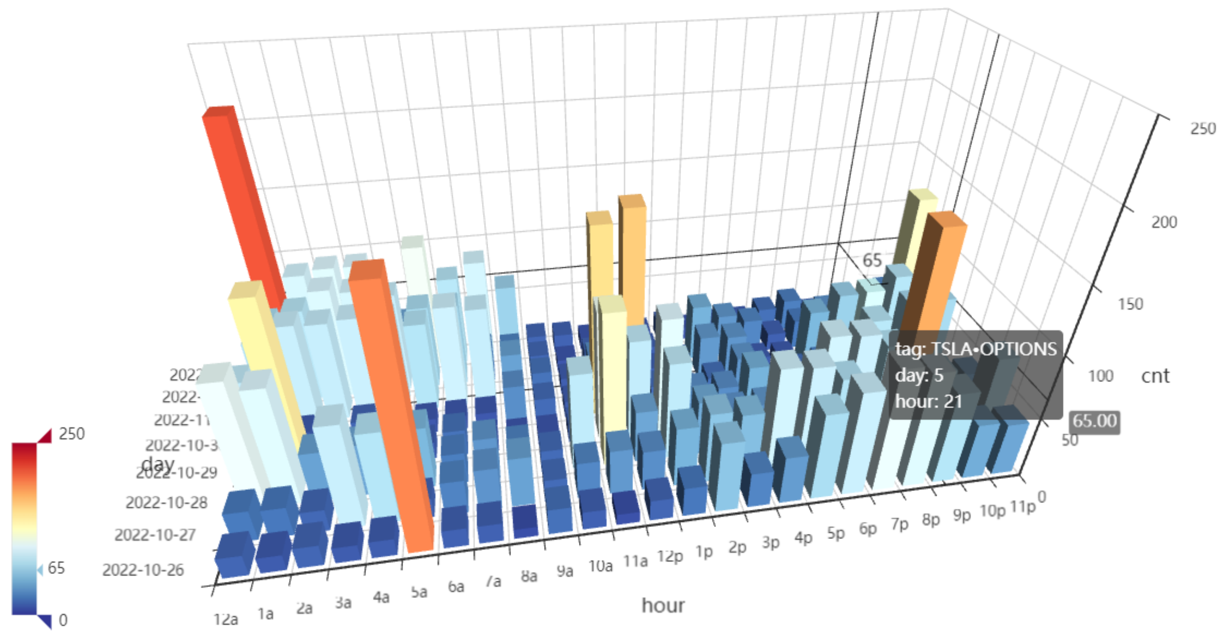
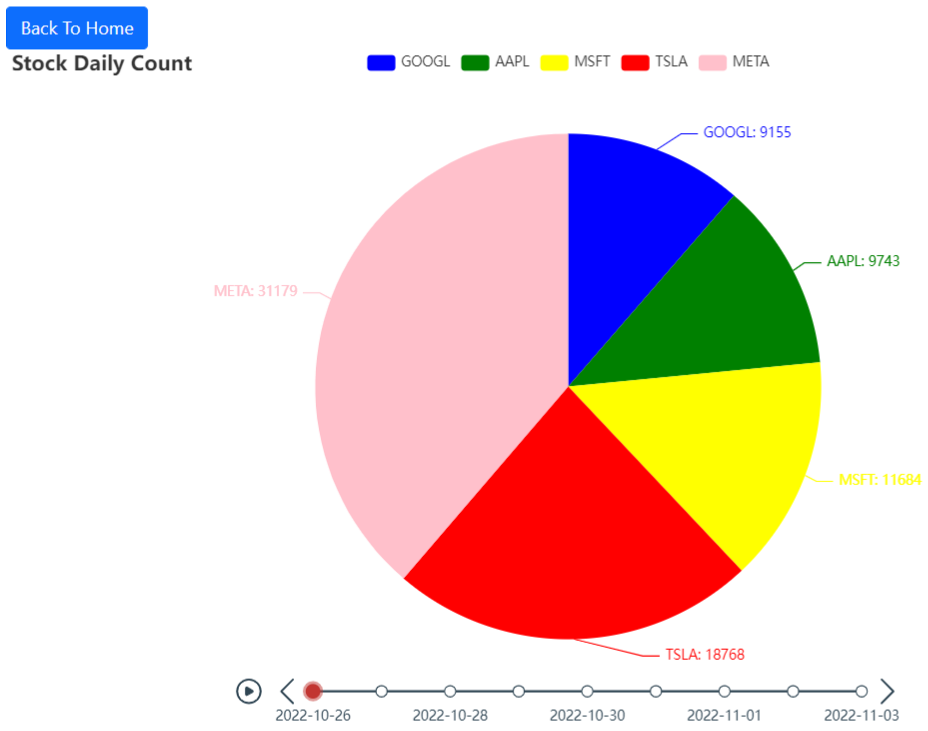
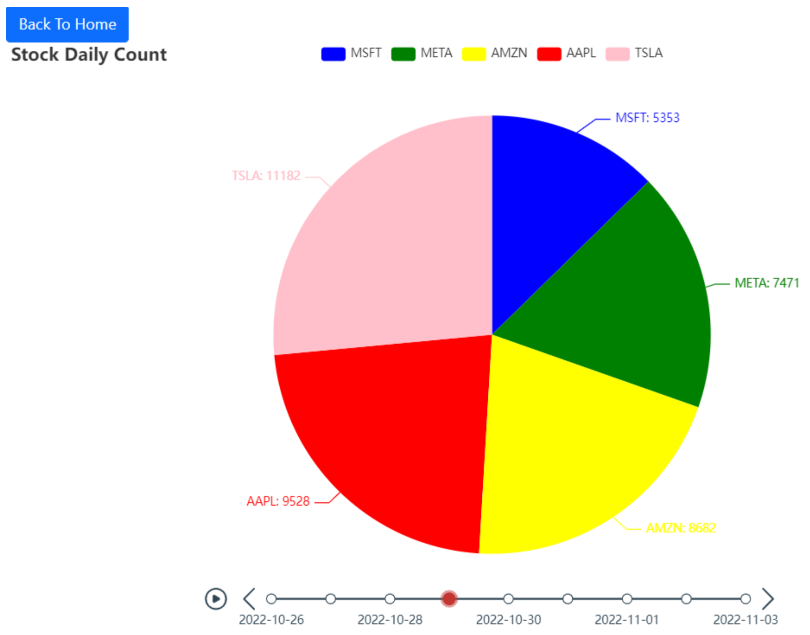
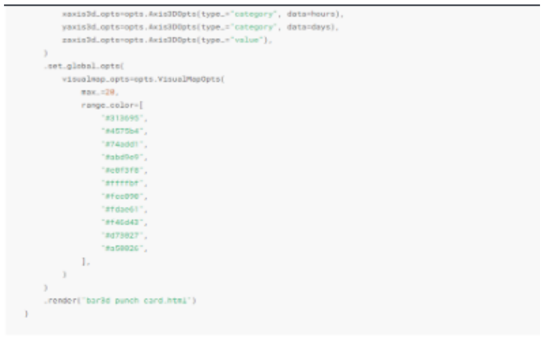


Chart4 (can select dates or select auto-play):
This chart shows the daily frequencies of tweets for the top 5 stocks as pie-charts.

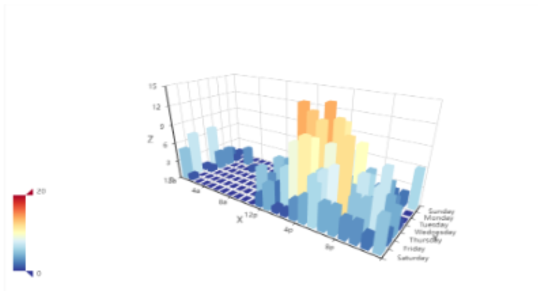

Back To Home
Stock Daily Count
GOOGL
AAPL
MSFT
TSLA
META
GOOGL: 9155
AAPL: 9743
MSFT: 11684
TSLA: 18768
META: 31179
2022-10-26
2022-10-28
2022-10-30
2022-11-01
2022-11-03


Back To Home
Stock Daily Count
MSFT
META
AMZN
AAPL
TSLA
MSFT: 5353
META: 7471
AMZN: 8682
AAPL: 9528
TSLA: 11182
2022-10-26
2022-10-28
2022-10-30
2022-11-01
2022-11-03

Reference:
Django template:https://docs.djangoproject.com/en/4.1/contents/
Bootstrap:https://getbootstrap.com/docs/3.4/css/
JS:https://getbootstrap.com/
Example using echart template(Pyechart)
https://echarts.apache.org/examples/en/index.html#chart-type-pie



```
            xaxis3d_opts=opts.Axis3DOpts(type_="category", data=hours),
            yaxis3d_opts=opts.Axis3DOpts(type_="category", data=days),
            zaxis3d_opts=opts.Axis3DOpts(type_="value"),
        )
        .set_global_opts(
            visualmap_opts=opts.VisualMapOpts(
                max_=20,
                range_color=[
                    "#313695",
                    "#4575b4",
                    "#74add1",
                    "#abd9e9",
                    "#e0f3f8",
                    "#ffffbf",
                    "#fee090",
                    "#fdae61",
                    "#f46d43",
                    "#d73027",
                    "#a50026",
                ],
            )
        )
        .render("bar3d_punch_card.html")
    )
```



Contribution

Garrett: Implemented the ploty graph and rendered it within index.html, and created forms to allow for user-interaction with this graph. Fixed minor formatting issues with charts 1-4.
Wyatt: No contribution other than changing table names when I made new ones.
Zeyu:Web interface design, frontend and backend coding. Figuring out how to use the pyechart template to process the data from our database.
Han:Edited and debug the side webpages followed by interface examples. Used the data from the linked database. Optimize CSS bootstrap styling.