Analysing Twitter Data for iOS and Apple Mentions

https://github.com/JAnderssonCCT/MSC_DA_CA2_T2

Abstract:

This scientific report aims to analyse Twitter data related to iOS, Apple, AAPL, iPhone, and iPad mentions. By leveraging the Twitter API and Spark Streaming, we collect real-time tweets, process them, and visualize the frequency of mentions using a dynamic bar chart. The goals of this study are to gain insights into the popularity and trends surrounding iOS and Apple products based on Twitter user conversations.

Introduction

The rapid growth of social media platforms has provided researchers with vast amounts of valuable data for various analyses. Twitter, being one of the most popular platforms, offers a rich source of information for studying public sentiment and tracking trends. In this study, we focus on analysing Twitter data related to iOS and Apple mentions. By understanding the level of discussion and sentiment surrounding these topics, we can gain insights into the popularity and trends associated with iOS and Apple products.

Objectives

The objectives of this study are to collect real-time tweets related to iOS, Apple, AAPL, iPhone, and iPad using the Twitter API. We aim to process and analyse the collected tweets to extract relevant information such as tweet text, author ID, and creation timestamp. The goal is to visualize the frequency of mentions for each topic using a dynamic bar chart. By continuously updating the chart in real-time, we can monitor the popularity and trends surrounding iOS and Apple products based on Twitter user conversations.

Methodology

The methodology employed in this study involves several steps to achieve the objectives:

3.1 Data Collection: To collect real-time tweets, we establish a connection to the Twitter API using OAuth1 authentication. We specify the topics of interest, including iOS, Apple, AAPL, iPhone, and iPad, and set up a streaming mechanism to receive relevant tweets. This allows us to gather a continuous stream of data from Twitter users discussing these topics.

3.2 Data Processing and Analysis: The collected tweets are processed and analysed using Spark Streaming, a distributed computing framework. We leverage its capabilities to efficiently handle the real-time tweet stream. The data processing tasks involve parsing JSON responses, filtering relevant tweets, and extracting key information such as tweet text, author ID, and creation timestamp. Based on this information, we calculate the frequency of mentions for each topic.

3.3 Visualization: To visualize the frequency of mentions, we set up a Flask web application. Using the Chart.js library, we create a dynamic bar chart that represents the popularity of iOS, Apple, AAPL, iPhone, and iPad based on the collected tweets. The chart is embedded in the web application and continuously updated with the latest data received from the Spark Streaming process. This allows for real-time monitoring of the trends and popularity associated with these topics.

Results and Analysis

In conclusion, this study aimed to analyse Twitter data related to iOS and Apple mentions. By leveraging the Twitter API, Spark Streaming, and visualization techniques, we were able to collect real-time tweets, process them, and visualize the frequency of mentions using a dynamic bar chart. The insights gained from this analysis provide valuable information about the popularity and trends surrounding iOS and Apple products based on Twitter user conversations. This study demonstrates the potential of using social media data for understanding public sentiment and monitoring discussions related to specific topics.

In conclusion, this study demonstrates the power of analysing Twitter data for gaining insights into the popularity and trends surrounding iOS and Apple products. By leveraging the Twitter API, Spark Streaming, and visualization techniques, we successfully collected real-time tweets, processed them, and visualized the frequency of mentions using a dynamic bar chart. The continuous monitoring of Twitter data allows for real-time analysis and provides valuable information about public sentiment and user discussions related to iOS and Apple. This study highlights the potential of social media data for understanding consumer perceptions and improving product strategies.

Code explained.

Importing Libraries

In this section, I will discuss the code snippet that involves importing the necessary libraries for our project. These libraries provide the functionality required to implement data collection, processing, visualization, and communication aspects of our analysis. The following libraries are imported:

The **json** library is used for parsing JSON responses from the Twitter API, while **sys** is utilized for handling system-level operations. The **requests** library enables making HTTP requests to the Twitter API, and **requests_oauthlib** provides OAuth1 authentication for accessing the API. The **tweepy** library is a Python wrapper for the Twitter API, which simplifies the process of retrieving tweets. The **logging** library allows for logging error messages and debugging information. The **pickle** library is used to serialize and deserialize objects, specifically the TCP connection object in this case. Lastly, the **socket** library enables establishing network connections for communication.

By importing these libraries, we ensure that we have the necessary tools to interact with the Twitter API, handle data, and establish a connection for data transmission through a TCP socket.

Deserializing TCP Connection and Sending Tweets

The next code section focuses on deserializing the TCP connection object and sending tweets to the Spark Streaming process. It involves the following code:

In this section, we first deserialize the TCP connection object using the **pickle** library. This allows us to establish a connection to the Spark Streaming process, which will receive the tweets for further analysis.

Next, we define the **send_tweets_to_spark** function, responsible for iterating over the response lines from the Twitter API and sending them to the TCP connection. The tweets are parsed as JSON objects, and the text of each tweet is extracted. We print the tweet text for debugging purposes and send it to the TCP connection.

To collect tweets from the Twitter API, we call the **get_tweets()** function, which will be explained in the subsequent section. The obtained tweets are then sent to the Spark Streaming process using the **send_tweets_to_spark** function.

This code section ensures that the tweets received from the Twitter API are transmitted to the Spark Streaming process through the TCP connection for further analysis.

Getting Tweets from the Twitter API

In this section, we focus on retrieving tweets from the Twitter API. The code snippet responsible for this functionality is as follows:

In this code section, we define the **get_tweets()** function responsible for making a request to the Twitter API and retrieving tweets based on specified query parameters. Here's a breakdown of the code:

1. We set the URL to **'https://api.twitter.com/1.1/search/tweets.json'**, which is the endpoint for searching tweets in the Twitter API.
2. Query parameters are defined in the **query_params** dictionary. These parameters allow us to filter tweets based on specific criteria. In this case, we search for tweets containing keywords related to iOS, Apple, AAPL, iPhone, and iPad. We specify the language as English (**'lang': 'en'**), request recent tweets (**'result_type': 'recent'**), and set the count to 1500, indicating the maximum number of tweets to retrieve.
3. We create an OAuth1 authentication object using the **requests_oauthlib.OAuth1** class. This object requires the consumer key, consumer secret, access token, and access token secret, which are obtained from the Twitter Developer account credentials.
4. Finally, we make a GET request to the Twitter API using **requests.get()**, passing the URL and query parameters. We include the OAuth1 authentication object to ensure proper authorization.

The **get_tweets()** function returns the response obtained from the Twitter API, which contains the tweets matching the specified query parameters. This response will be processed and analyzed in the subsequent sections of our project.

Processing Tweets and Extracting Information

Once we have obtained the response from the Twitter API, the next step is to process the tweets and extract relevant information from them. The code snippet responsible for this functionality is as follows:

In this code section, we define the **process_tweets()** function, which takes the response obtained from the Twitter API as input. Here's a breakdown of the code:

1. We first check the status code of the response to ensure that the request to the Twitter API was successful (**response.status_code == 200**).
2. If the response is successful, we extract the JSON data from the response using **response.json()**. This data contains the tweets returned by the Twitter API.
3. We retrieve the **statuses** field from the JSON response, which represents the list of tweets.
4. We iterate over each tweet in the **tweets** list and extract relevant information such as the tweet ID, creation timestamp, author ID, and tweet text.
5. We print the extracted information for each tweet, including the tweet ID, creation timestamp, author ID, and tweet text. This allows us to verify that the data has been successfully processed.
6. Finally, if the response status code is not 200, we print an error message indicating the encountered error.

The **process_tweets()** function provides us with the ability to analyse and work with the extracted tweet information in subsequent sections of our project.

Sending Tweets to the TCP Connection

Once we have processed and extracted the tweet information, the next step is to send the tweets to the TCP connection. This is achieved using the **send_tweets_to_spark()** function, as shown in the following code snippet:

In this code section, we define the **send_tweets_to_spark()** function, which takes two parameters: **http_resp** representing the response obtained from the Twitter API, and **tcp_connection** representing the TCP connection object.

Here's an overview of the code:

1. We iterate over the response lines obtained from the Twitter API using the **iter_lines()** method of the **http_resp** object. Each line represents a tweet in JSON format.
2. For each line, we parse it as JSON using **json.loads(line)**, storing the tweet data in the **full_tweet** variable.
3. We extract the tweet text from the **full_tweet** dictionary using **full_tweet['text']** and store it in the **tweet_text** variable.
4. We print the tweet text using **print("Tweet Text: " + tweet_text)** to verify the contents of the tweet.
5. Finally, we send the tweet text to the TCP connection by encoding it as bytes (**tweet_text.encode()**) and appending a newline character (**b'\n'**). This ensures that each tweet is sent as a separate message over the TCP connection.

The **send_tweets_to_spark()** function enables the real-time streaming of tweets from the Twitter API to the TCP connection, facilitating further analysis or processing of the data on the receiving end.

Integration of Tweet Retrieval and TCP Connection Setup

To integrate the tweet retrieval from the Twitter API and the TCP connection setup, we utilize two separate threads: **thread1_function()** and **thread2_function()**. Let's examine how these threads work together to achieve the desired functionality.

The code above demonstrates the orchestration of the two threads. Let's break it down:

1. In **thread1_function()**, we set up the parameters for the TCP connection, such as the IP address (**TCP_IP**) and port number (**TCP_PORT**). We create a TCP socket and bind it to the specified IP and port using the **socket.socket()** and **bind()** functions, respectively.
2. We start listening for incoming TCP connections using the **listen()** method of the socket. Once a connection is established, we accept it and store the connection object and address in variables **conn** and **addr**.
3. We then proceed to retrieve tweets from the Twitter API by calling the **get_tweets()** function. This function sends a request to the API, retrieves the response, and returns it.
4. Next, we process the response and extract tweet information using the **process_tweets()** function. This function parses the response JSON, extracts relevant data such as tweet IDs, creation timestamps, author IDs, and tweet text, and prints them to the console.
5. After processing the tweets, we serialize the connection object using **pickle.dumps(conn)**. Serialization converts the connection object into a byte stream, which can be passed as an argument between different processes or threads.
6. We return the serialized connection object from **thread1_function()** and store it in the **serialized_conn** variable.
7. In **thread2_function()**, we deserialize the connection object using **pickle.loads(serialized_conn)**. Deserialization converts the byte stream back into a connection object.
8. We then run the script **send_data.py** using the **subprocess.run()** function. The serialized connection is passed as an argument to the script, allowing it to establish a connection and send the tweet data to the TCP connection.
9. Finally, we create a **ThreadPoolExecutor** with two threads using the **concurrent.futures.ThreadPoolExecutor()** function. We submit **thread1_function()** to execute in **thread1** using **executor.submit()**, and retrieve the result using **future1.result()**.
10. We pass the serialized connection obtained from **thread1_function()** as an argument to **thread2_function()** and submit it to execute in **thread2**. We wait for **thread2_function()** to complete by calling **future2.result()**.

By utilizing two separate threads, we achieve concurrent execution of the TCP connection setup and tweet retrieval, ensuring efficient and real-time streaming of tweets to the TCP connection for further processing or analysis.

Flask App Setup and Routes

The next section of the code sets up a Flask application and defines the necessary routes for the web application. Let's explore how these routes handle chart rendering, data refreshing, and data updating.

In this section, we define the Flask routes responsible for rendering the chart page, refreshing the graph data, and updating the data. Let's delve into each route:

1. The route **'/'** is associated with the **get_chart_page()** function, which serves as the route handler for the main chart page. When a user visits the root URL, this function is executed. It clears the **labels** and **values** lists, and then renders the **chart.html** template using **render_template()**. The template is passed the empty **values** and **labels** lists.
2. The route **'/refreshData'** is mapped to the **refresh_graph_data()** function, which handles the request to refresh the graph data. When a client sends a GET request to this route, it returns the current **labels** and **values** as a JSON response using **jsonify()**.
3. The route **'/updateData'** is associated with the **update_data()** function, which handles the request to update the data. This route expects a POST request with the form fields **'label'** and **'data'**. It updates the **labels** and **values** lists based on the received data. If the data is successfully updated, it returns a "success" response with status code 201. Otherwise, it returns an "error" response with status code 400.
4. The final block **if __name__ == "__main__":** ensures that the Flask app is run only if the script is executed directly (not imported as a module). It starts the Flask development server, specifying the host as **'localhost'** and the port as **5001**.

The Flask app and its routes provide the necessary functionality for serving the chart page, refreshing the data, and updating the data based on user interactions.

Embedding the Flask App in Jupyter Notebook

To seamlessly integrate the Flask app into the Jupyter Notebook environment, we utilize the **IFrame** class from the **IPython.display** module. The following code snippet demonstrates how the Flask app is embedded in the Jupyter Notebook:

In this section, we specify the URL of the Flask app by setting the **app_url** variable to **'http://localhost:5001/'**, indicating that the app is running locally on port 5001. Then, we use the **IFrame** class to embed the Flask app in the Jupyter Notebook. The **src** parameter is set to the **app_url**, and we specify the width and height of the embedded frame.

By embedding the Flask app in the Jupyter Notebook, we can interact with the chart and visualize the dynamically updated data directly within the notebook environment.

Discussion of Rationale and Justification

In the development of this project, several choices were made in terms of data processing and storage, programming language, and machine learning models. Each choice was carefully considered based on its suitability for the project's requirements and objectives. Let's discuss the rationale and justification behind these choices:

Data Processing and Storage: The project involves retrieving and processing real-time data from the Twitter API. To accomplish this, the Python programming language was chosen due to its rich ecosystem of libraries and frameworks for data processing and analysis. The Tweepy library was used for interacting with the Twitter API, allowing us to retrieve tweets based on specific criteria such as keywords and language. The received tweets were processed and extracted using the JSON format, making it convenient for further analysis and visualization.

For storing the data, a TCP connection was established between the data retrieval process and the visualization process. This choice was motivated by the need for real-time data streaming from the Twitter API to the Flask app. By utilizing a TCP connection, the tweets could be efficiently transmitted and received by the visualization component.

Programming Language Choice: Python was chosen as the programming language for this project due to its versatility, ease of use, and extensive range of libraries and frameworks for data processing, analysis, and visualization. Python's readability and concise syntax make it ideal for rapid prototyping and development. Additionally, Python has excellent support for working with APIs and handling data in various formats, making it well-suited for interacting with the Twitter API and processing the received data.

Machine Learning Models and Algorithms: In this project, the focus is on real-time data visualization and monitoring rather than applying complex machine learning models or algorithms. Therefore, the use of machine learning models or algorithms was not a primary consideration. However, if desired, the collected data could be further analysed and processed using machine learning techniques to derive insights or make predictions.

The choice of machine learning models and algorithms would depend on the specific goals and objectives of the analysis. For example, if sentiment analysis of the tweets was desired, natural language processing (NLP) techniques and models such as recurrent neural networks (RNNs) or transformer-based models like BERT could be employed. Alternatively, if the objective was to identify trending topics or detect anomalies, unsupervised learning algorithms like clustering or anomaly detection could be explored.

Forecasted Sentiment explained:

The table above presents the forecasted sentiment values for different time horizons: 1 week, 1 month, and 3 months. These sentiment values are numerical representations of the predicted sentiment associated with the collected Twitter data. The sentiment values range from -1 to 1, where negative values indicate negative sentiment, positive values indicate positive sentiment, and values close to 0 indicate neutral sentiment.

It's important to note that the forecasted sentiment values are presented for the time periods ranging from data point 120 to 209. The table showcases the sentiment forecasts for each corresponding data point within the given time range.

For example, at data point 120, the forecasted sentiment value for all three-time horizons (1 week, 1 month, and 3 months) is -0.09. Similarly, at data point 121, the forecasted sentiment value remains consistent at 0.05 for all three-time horizons.

However, beyond data point 205, the forecasted sentiment values are represented as "nan," indicating that no sentiment forecast is available for those data points. This could be due to the limitations of the forecasting model or the unavailability of sufficient data for accurate predictions.

It's worth mentioning that the sentiment forecast plays a crucial role in understanding the overall sentiment trends and patterns over time. By examining the forecasted sentiment values, one can gain insights into the expected sentiment trajectory and identify potential shifts in public opinion or sentiment.

To generate these sentiment forecasts, various forecasting techniques and models could have been employed, depending on the nature of the sentiment data and the desired forecasting accuracy. Time series analysis methods, such as autoregressive integrated moving average (ARIMA) or exponential smoothing models, could be utilized to capture the temporal dependencies and patterns in the sentiment data.

Overall, the forecasted sentiment values provide valuable information for tracking sentiment trends and making informed decisions based on the anticipated sentiment changes in the future.

Optimizing with L-BFGS-B

The output begins by providing the machine precision, denoted as 2.220D-16. This figure signifies the level of precision of the floating-point arithmetic employed during the computation. It serves as a reference point for the accuracy of subsequent calculations.

Next, the output presents N and M, where N represents the number of variables in the optimization problem, and M signifies the maximum number of variable metric corrections utilized in the limited memory BFGS update. In our case, N is 3, implying we are dealing with a three-dimensional optimization problem, while M is set to 10.

Iterative Information:

The subsequent sections of the output provide information about the progress of the algorithm at various iterations:

1. At X0: This section assesses the initial point of the optimization process, referred to as X0. Here, it states that none of the variables in the optimization problem are exactly at the bounds, indicating that no constraints are imposed on the variables.
2. At iterate 0: The output reveals the results obtained at the initial iteration, denoted as iterate 0. It includes the objective function value (f), which is -2.22531D-01, and the norm of the projected gradient (|proj g|), measuring 1.86492D-02. These values reflect the initial state of the optimization process.
3. At iterate 5: This section displays the results achieved at the 5th iteration. It provides insights into the progress of the optimization algorithm, presenting the objective function value and the norm of the projected gradient at this stage.
4. At iterate 10: Similarly, this part showcases the outcomes at the 10th iteration, shedding light on the developments in the optimization process. The objective function value and the norm of the projected gradient are reported here.

Statistics and Convergence:

The output also presents several statistics regarding the optimization process:

- Tit: The total number of iterations executed throughout the optimization process.
- Tnf: The total number of function evaluations performed during the optimization.
- Tnint: The total number of segments explored during Cauchy searches, which is relevant to the algorithm's optimization strategy.
- Skip: The number of BFGS updates skipped during the optimization, indicating the efficiency of the algorithm.
- Nact: The number of active bounds at the final generalized Cauchy point, providing insights into the constraint handling aspect of the optimization problem.
- Projg: The norm of the final projected gradient, which characterizes the magnitude of the gradient at the optimized solution.
- F: The final function value achieved at the end of the optimization process.

The output concludes with a convergence message. In this case, it signifies that the optimization process has converged based on the specified criterion. The convergence criterion compares the relative reduction of the objective function (REL_REDUCTION_OF_F) to a factor (FACTR) multiplied by the machine precision (EPSMCH). This indicates that the algorithm has successfully minimized the objective function within the desired precision.

By unravelling the output of the L-BFGS-B optimization algorithm, we gain valuable insights into its execution and outcomes. Understanding the machine precision, iterative information, statistics, and convergence criteria empowers us to comprehend the progress and results of the optimization process. With this knowledge, we can effectively utilize the L-BFGS-B algorithm for solving unconstrained optimization problems, harnessing its power to unlock optimal solutions.