# Abstract

A website that displays interactive graphs for machine learning results and connects to a database or SQL server involves several layers of development, including the backend, database, front end, and machine learning components.

A web-based dashboard that performs real-time sentiment analysis on social media posts related to a specific topic is created in this project. There are options for the user to input his/her query and the number of tweets to be sampled from recent tweets on that topic. The app is deployed onto the a live web server using python anywhere.

To accept Twitter API accessed information to determine the sentiment on the reception of the new videogame ‘Lies of P’ developed by Round 8 Studios, split into three categories; negative, neutral, and positive.

# Intro

[**PythonAnywhere Setup Instructions**](https://github.com/conradho/dashingdemo/blob/master/README.md#pythonanywhere-setup-instructions)

* create a virtualenv and pip install -r requirements3.6.txt into it
* create a new custom webapp on PythonAnywhere
* configure your webapp to use your virtualenv on the PythonAnywhere webapp dashboard
* setup your wsgi.py file as in the example in the repo, and make sure it can find and import the dashing\_demo\_app
* reload your webapp and go to your webapp url!

# Journal

So, I studied all the concepts first.

I wanted to use Heroku to host the web app, but it wouldn’t accept my card details for my country. I went with the free option instead, using glitch.com.

# Implementation of Twitter API

First, I identified which endpoint you would like to use. I used RecentSearch.

Second, I chose a tool to make the request. I chose Postman.

Postman is a visual tool that you can use to make requests to REST endpoints. We’ve created some great materials around Postman to help you get started with and explore the different endpoints available via the Twitter API.

You can’t make requests to streaming endpoints using Postman. Visit the filtered stream or 1% ampled stream quick start guide to learn how to work with those endpoints.

Next, review the response. If you used an endpoint that utilizes a GET HTTP method, you will receive metadata related to the resource (Tweet, user, List, Space, etc) that you made the request to in JSON format. Review the different fields that returned and see if you can map the information that you requested to the content on Twitter.

Adjust the request using parameters. You can also experiment with a variety of different filtering tools with endpoints such as search Tweets, Tweet counts, and filtered stream to help narrow down the data you receive to just those Tweets that you are interested in.

# Implementation of Rendering

The key steps are:

1. Create a Dash instance to render the app
2. Start the server
3. Import the rendered app
4. Create a test client for the rendered app
5. Make assertions and test as usual

UNIT TESTS

To create unit tests for the provided Python code that uses Dash and other modules, you can use Python's built-in unittest framework along with Dash's own testing utilities.

Using unit tests, I tested the functions and whether they were working. This proved tedious, so I downloaded pylint and read up on the documentation for errors in my code to speed up the process. Unit tests, tested the following in the beginning:

* The layout elements exist
* The API call returns expected data

Some things that could be improved:

* Test callbacks and interactions
* Use mocks for data instead of actual API calls
* Break into smaller focused test cases
* Add more assertions on the data

For the finished product, I made another unit\_test2 file. The unittest.mock library is used to mock objects and test the code in isolation.

In the TestTwitterAPI class, the @patch decorator is used to mock the requests.post method. You can use a similar approach to mock other API calls.

This is a basic example and might not cover all edge cases or functionalities.

Run the test suite by executing python -m unittest test\_main.py in your command line.

UPDATING THE DASHBOARD

The update\_dashboard function is called automatically by Dash when one of the specified input events triggers it. You don't need to call it explicitly in your code. Instead, you set up the callback decorator to specify when and how the function should be invoked.

@app.callback(

[Output('sentiment-histogram', 'figure'), Output('sentiment-table', 'data'), Output('sentiment-table', 'page\_size')],

Input('column-selector', 'value'), # Input event

Input('pagination-slider', 'value'), # Input event

Input('export-button', 'n\_clicks'), # Input event

Input('update-database-button', 'n\_clicks'), # Input event

State('sentiment-table', 'page\_current'), # State (data stored in the component)

)

def update\_dashboard(selected\_column, slider\_value, export\_clicks, update\_db\_clicks, current\_page):

I made an input box where the user can enter the query string and a button that, when clicked, will update the Dash table and the Dash graph with the results from the new query.

GLITCH

I tested out the code on glitch.com and tried using javascript, html and css code to run some of the operations I was running in python and much more. Although I did not use this information, I will list some of the thing I learned for each language here relevant for this project.

In a javascript file:

• DOM elements are selected using document.getElementById.

• Event listeners are added to various elements to handle user interactions.

• Four functions (handleColumnSelection, handleSliderChange, handleExportButtonClick, handleUpdateButtonClick) are defined to respond to these interactions.

* You can replace the console.log statements in each function with the actual logic for your project. For example, you can update your application state, make API requests, or trigger other actions.

This CSS file I used included styles for:

• The body and container of your web page.

• Header text (<h1>).

• A dropdown (#column-selector) for column selection.

• A pagination slider (#pagination-slider) for controlling the number of displayed rows.

• Export and update buttons (#export-button and #update-database-button).

• A table (#sentiment-table) for displaying data, with styles for table headers, alternating row backgrounds, and cell padding.

* You can adjust the colors, fonts, padding, and other styles as needed to match your desired design.

HTML:

* This HTML version replicates the structure of your Dash app's layout using HTML elements, including dropdowns, buttons, and tables.
* You'll need to replace some placeholder values and dynamically generate options, headers, and table data as per your Python code.
* Additionally, you may need to include any custom JavaScript code for interactivity, which is typically handled by Dash callbacks in the Python code.

1. **User Authentication:** Implement user authentication to secure your dashboard. Allow users to sign in and access personalized data or settings.
2. **Export Data:** Add the ability to export data from the table to various formats like CSV or Excel for further analysis.
3. **Data Filtering:** Enable users to filter data based on specific criteria, such as date ranges, sentiment thresholds, or specific keywords.
4. **Real-Time Updates:** If your data source provides real-time updates, incorporate WebSocket or other technologies to display live sentiment analysis results.
5. **Data Visualization:** Include additional types of visualizations, such as line charts, scatter plots, or heatmaps, to provide different insights into the sentiment data.
6. **Sentiment Trends:** Implement features to analyze sentiment trends over time and display them in a time series plot or chart.
7. **Word Clouds:** Create word clouds to visually represent the most frequent words or phrases in the sentiment data.
8. **Sentiment Heatmap:** Develop a heatmap to visualize sentiment scores across different categories or topics.
9. **Customizable Dashboard:** Allow users to customize the layout and appearance of the dashboard, such as choosing colors or themes.
10. **Data Source Integration:** Integrate with additional data sources, such as social media APIs or external databases, to expand the data available for analysis.
11. **Notifications:** Implement notifications or alerts for specific sentiment events or trends, such as sentiment turning highly negative or positive.
12. **Data Preprocessing:** Enhance data preprocessing by cleaning and normalizing text data, removing stopwords, and performing advanced NLP techniques.
13. **Sentiment Benchmarking:** Compare sentiment scores with industry benchmarks or competitor sentiment for additional context.
14. **User Guides and Help:** Provide documentation or tooltips within the dashboard to help users navigate and understand the features.
15. **Performance Optimization:** Optimize the performance of your application for large datasets by implementing data caching, pagination, or server-side rendering.
16. **Responsive Design:** Ensure that your dashboard is responsive and works well on various screen sizes, including mobile devices.
17. **Error Handling:** Implement robust error handling and error messages to guide users in case of issues or incorrect inputs.
18. **Feedback Mechanism:** Include a feedback form or mechanism for users to provide input or report issues.
19. **Dashboard Sharing:** Allow users to share specific views or analyses with others, possibly through URL sharing or embedding.
20. **Multilingual Support:** If applicable, add support for multiple languages in your dashboard.

Functional vs. OOB Methods  
This can be seen in the classes created where encapsulation occurs. For example, in the Tweet Class the VADER sentiment analysis is encapsulated in the statements blob.sentiment.polarity and blob.sentiment.subjectivity. Whereas external functions in the form of analyze\_sentiment\_pattern(text), can be used for sentiment analysis and do not need the static VADER method within the class.

# Analysis

# Discussion

# Conclusion