CAB432 Cloud Computing

Assignment: Mashup / Docker Project

Project Name

Hated in the Nation (unpopularity leaderboard)

Student

Jaimyn Mayer (n9749331)

Date

17 September 2018

Table of Contents:

Introduction

The purpose of this mashup is to provide detailed sentiment analysis of a list of #tags or @tags. The application will allow users to target a list hashtags/@tags. It will then show users a breakdown of each state's sentiment with some detailed statistics and information.

The main user interface is a single page web app (SPA) and allows people to see who the most liked/disliked politicians, reality tv star actors etc are in near real time. It has a polling feature which means the application automatically pulls new data every few seconds.

The main feature of the application is the "Unpopularity Leaderboard". This bar sits on the left and is always visible. It updates in near real time with a ranking of the least unpopular person or #hashtag.

The 3 different views on the right hand side allow the user to do multiple things. The first one allows you to search through all of the tweets in the database and sort them by username, text, likes, retweets location or sentiment. The second gives you a visual overview of the different states and lists a localised unpopularity leaderboard for each state. Finally, there's a "more stats" view which gives in depth statistics of each states, person/tag and overall. A full usage guide has been included in Appendix A.

Twitter API

https://developer.twitter.com/en/docs.html

This allows us to search for specific hashtags (#yeet) or at tags (@realDonaldTrump). This is also used to find metadata about the users who post each tweet for location statistics purposes.

Microsoft Text Analytics API

https://azure.microsoft.com/en-gb/services/cognitive-services/text-analytics/

This API allows us to analyse all of the tweets and generate a sentiment for each one. This forms the basis of the entire application as all of the statistics etc relies on this service.

Google Find Places API

https://developers.google.com/places/web-service/search

This API allows us to find the location that each tweet was made from. The twitter geolocation searching was unreliable so the user's specified location was used instead. This is a free text input so the location was normalised so the state could be found.

Leaflet

http://leafletjs.com/

This is a mapping API/library and allows us to visualise the sentiment from around the country. It relies on the mapbox API to download map tiles/location data.

Use Cases

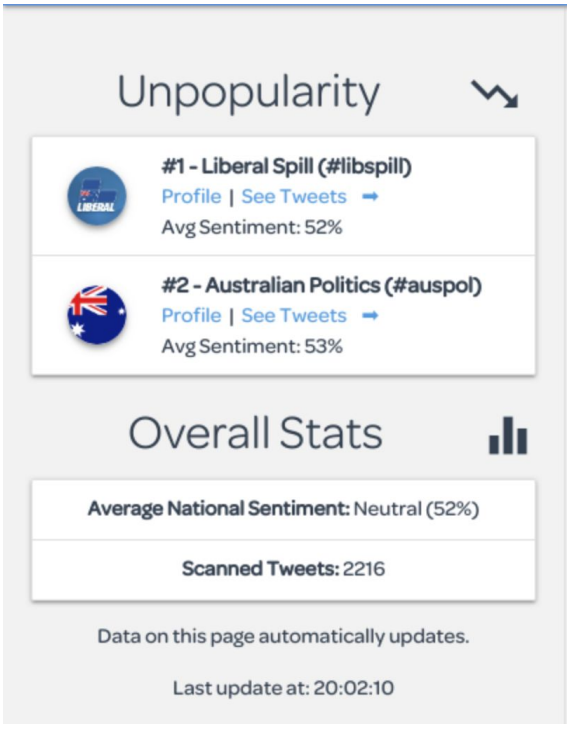
Use Case A

As someone who is interested in politics, I want to be able to see what the entire country is feeling about certain topics (such as #auspol and #libspill)

The left bar is always visible, and contains a leadboard of unpopularity and some brief overall stats about the data. This allows the user to visually see a list of topics and what order they're currently ranked in, based off the sentiment analysis of *all* tweets with the relevant tag in it.

There are two convenient links included in each tag's entry on the leaderboard. "Profile" is a direct link to the tag's twitter page. The "See Tweets ->" link filters the table containing every tweet which is on the right side of the page.

The user is able to quickly determine what the country thinks about certain topics, and how they're ranked in relation to each other.



This leaderboard is generated from statistics stored in the applications database. The specific APIs being used to generate the data that's visible above are Twitter, Microsoft Text Analytics and Google Maps Find Places. The tweets are pulled from twitter using their API and sent to the Microsoft API for analysis. Then the google API is used to normalise the location and the final result is stored in the database. The front end uses an API provided by the backend to generate the statistics which are finally rendered into the leaderboard above.

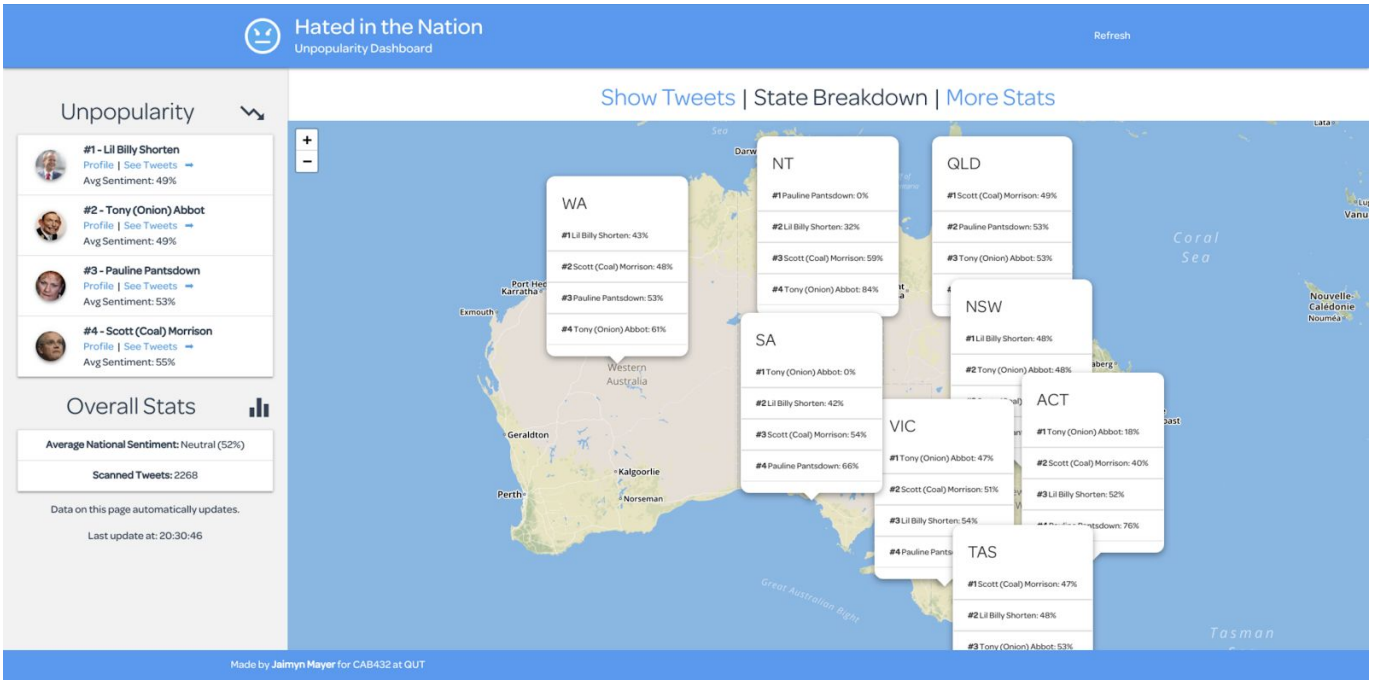
Use Case B

As someone who likes politics I want to see what each state thinks about ministers in the upcoming election.

The “State Breakdown” view allows users to see a graphical representation of each tag’s sentiment per state. The map is interactive so the user can zoom in to only the relevant states or zoom out (pictured below) to see everything all at once. There are more specific statistics available for each state in the “More Stats” view. This is detailed on the next page.

The popup box for each state shows a cut down leaderboard with just their name and average sentiment. Each list then ranks the tags in order of least popular to most popular. These rankings only use tweets with a *confirmed* location in that state. This means the data used by this page is a subset of the overall sentiment statistics.

Due to the state breakdown feature, the user is able to quickly determine that their favourite politician is currently popular in Western Australia and New South Wales, but isn’t as popular in the other states.



Similar to the last use case, the data used by this view is generated from the statistics stored in the applications database. The front end uses an API provided by the backend to generate the state by state statistics which are rendered into the leaderboard above. The specific APIs being used to generate the data that’s visible above are Twitter, Microsoft Text Analytics and Google Maps Find Places.

In addition to this, leaflet is being used to graphically display the data. Leaflet relies on the mapbox API to provide the imagery data, and the front end dynamically creates popup boxes with near real time data.

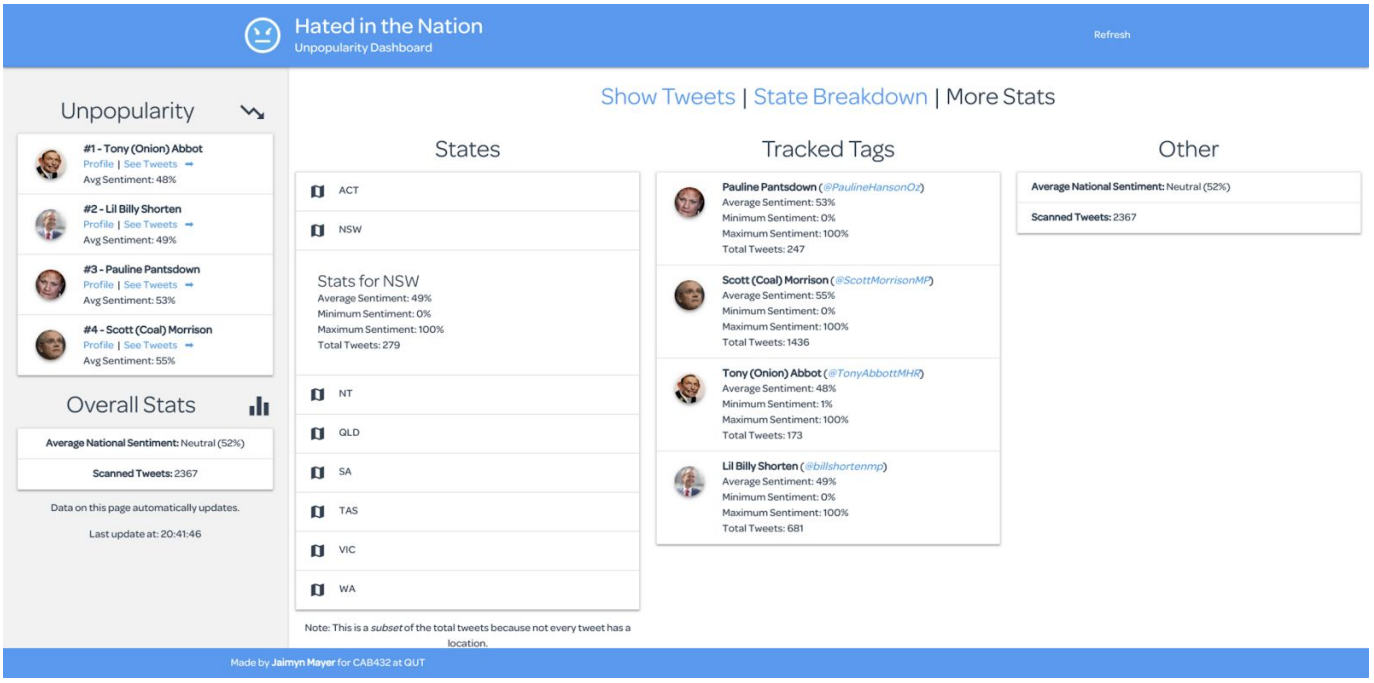
Use Case C

As a user who loves data, I want to be able to see specific statistics about different people/tags and locations.

The “More Stats” view allows the user to see detailed statistical data about each state, each tag that is being tracked and other data. Because there is a lot of data available, the user interface collapses each state into a list. The user is able to click on a state to view the data. The data that is viewable for each state includes average, minimum and maximum sentiment, as well as the total amount of tweets used to generate those statistics.

The tracked tags column is a list similar to the “unpopularity leaderboard” on the left, but contains more data. In this column, the user can see the average, minimum and maximum sentiment, as well as the total amount of tweets used to generate them. The final column shows the rest of the currently available data for completeness.

The user is able to see there isn’t much data available for the ACT as there is for NSW so the statistics may not be a fair representation of each state. They can also see that there is a wide variety in the sentiment of each tweet by each person as there’s a large difference between the minimum and maximum sentiments.



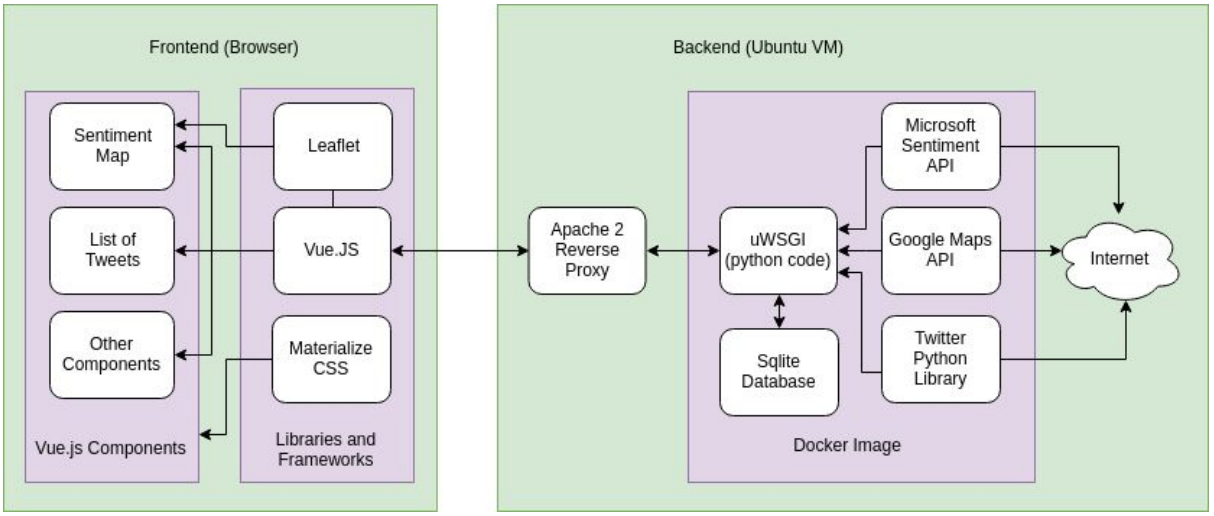
Due the similarities with use case A, the APIs used will just be a brief mention. The specific APIs being used to generate the data that’s visible above are Twitter, Microsoft Text Analytics and Google Maps Find Places. As is the case previously, the front end receives data which is calculated by the applications backend.

Technical Description

The goal from the very beginning was to create a nice looking user interface that performed well. To do this, minimal data processing is being done on the client side frontend. All of the external API requests and statistical calculations are being done on the back end and provided to the front end to display. Due to the “live polling” feature, there are quite a few constant network requests but the size and frequency of these have been minimised. What this means is the client makes a network request to the backend every few seconds to check for updated data.

The entire application was created in Python (backend) and JavaScript (frontend). The main framework/libraries that have been used include flask, vue.js, webpack and materialize css. Vue.js was used as it has some nice features that make it easier to dynamically update data. This also allowed the design to be a single page app (SPA) which generally tends to create nicer user experience as the user doesn’t have to move between completely separate pages.

The diagram below shows the major components and how the data flows through them.



Frontend

Overview

The front end is made up of numerous components that are built with a framework called vue.js. These reusable components dynamically pull data from the backend and update in near real time. The individual components make GET requests to the backend API endpoints using a library called axios.

Materialize CSS

This is a CSS framework that makes it easier to develop nice looking, material design inspired websites and web apps. It has default styling for many elements like titles, lists, etc. It also has some nice default styling and ui behaviours.

Vue.js

Vue.js is an adaptive JavaScript framework and is incrementally adoptable. This was used for the front end user interface. It has a large feature set, but only the basics were used. Vue.js has a components system, which allows the developer to create small, reusable sections of code. Many components were created for the front end. These include All Statistics, Home, Overall Stats, Sentiment Map, Tags and Tweets. Together, these components form the entire front end user interface. As the data each component references is updated, vue.js will automatically update the html elements.

Leaflet

Leaflet is a mapping framework that allows you to display maps and overlay them with interactive data. Leaflet is used to show popup boxes with live sentiment information about each state.

Backend

Overview

The backend basically acts like a data collection, storage and number crunching service. While front end clients are connected, it will scrape tweets, use the microsoft API to run sentiment analysis, normalise the location with the Google API then cache the results in a local sqlite database. The backend creates a private API used by the application. Each endpoint searches the database and runs calculations like min/max/average sentiment etc.

Flask

Flask is a python microframework which makes it easy to create python based web applications. It's designed for smaller/less complex projects and APIs which makes it perfectly suited to this project. This handles accessing all of the external APIs, database interfacing and number crunching. In combination with apache and uWSGI it also serves the application files.

Google Maps Places API

The google maps places API is used to help normalise and geolocate the tweet locations. The twitter geolocation search filter was unreliable so the application uses the location specified in the twitter person's profile. This is a free text input so this API is used to search for and normalise the location.

Microsoft Text Analytics API

The Microsoft text analytics API will run advanced machine learning models over strings of text and return the best approximation of it's sentiment. This sentiment value is a number between 0 signifying negative and 1 signifying positive.

Twitter Python Library

This library abstracts away some of the HTTP requests to the twitter API. This library lets us search twitter for specific #tags or @tags. The geolocation search filter was unreliable so location filtering is done as explained above.

Issues Encountered

There were no majors issues encountered while creating the project. However, there were a few minor problems and changes along the way.

Initially, a full task scheduling system was going to be used. (celery or AP scheduler) However, this presented difficulties with running in a docker environment so a more basic approach was taken. The backend will only scrape and process data while front end clients are connected. This was done to get around the scheduler issues and to minimise the external API requests. (Microsoft has a relatively small limit on the amount of requests allowed)

The IBM Watson tone analysis API was going to be used to provide detailed tone/emotional analysis on each tweet. However, there's a limit of 1000 API requests which was far too low to be useful. In just a day of development, over 1000 tweets were collected and processed which exceeds the limit. This planned feature was dropped.

Docker

Due to the requirements of this project, Docker was used. Docker is system of containers that allow a developer to easily create deployable applications. It also creates isolation between different apps and allows better use of resources. The docker application contains all of the backend code and talks to the outside world through an apache reverse proxy which adds https support. The full dockerfile is attached as an appendix.

The base image used was *ubuntu:16.04*. This was due to familiarity with the operating system and ease of use.

```
# Our base image is Ubuntu 18.04.
FROM ubuntu:16.04
MAINTAINER Jaimyn Mayer (hello@jaimyn.com.au)
```

After updating the repository cache, a few operating system and python packages are then installed. These include curl, python3, python3-pip and others. These were installed with the "RUN" command. Next the source code is copied to the container and the frontend is compiled/built with the "npm run build" command.

```
# Copy our codez.
ADD /app /app

# Set our default working directory to build our front end (webpack etc.).
WORKDIR /app/frontend
RUN npm run build

# Set our default working directory to install python and run flask.
WORKDIR /app/backend
```

Finally, port 8000 is exposed from the container and uWSGI is started.

```
# Expose our port and start our app.
EXPOSE 8000
CMD uwsgi --http 0.0.0.0:8000 --module app:app --processes 1 --threads 8
```

Testing and Limitations

A wide variety of error handling has been implemented to ensure a pleasant user experience. The use of external APIs and services always increase the likelihood of failure having a large impact on usability. Because the application caches all previous tweets, it will continue work with existing data even in the event of a complete failure on all external services. Of course without the Twitter and Microsoft API no new tweets can be processed and the Google API is required for location based filtering/statistics.

An example of a runtime configuration error being handled. The application checks to make sure all of the environment variables exist and warns the developer by throwing a runtime exception.

```
# All of these environment variables are needed to run.
needed_vars = (
    "TWITTER_CONSUMER_KEY",
    "TWITTER_CONSUMER_SECRET",
    "TWITTER_ACCESS_TOKEN_KEY",
    "TWITTER_ACCESS_TOKEN_SECRET",
    "GOOGLE_FIND_PLACES",
    "MICROSOFT_API",
)

# Check if all of the environment variables exist and raise an exception if any of them don't.
for var in needed_vars:
    if var not in os.environ:
        self.debug("missing environment variable: " + var, 1)
        raise RuntimeError("Missing environment variable: " + var)
```

Another example is the Google Maps Find Places API. The API returns a “status” different than “OK” if it was not successful. This is handled with the first if statement. However, it was found during testing that sometimes the API will return a blank list, even if the status is OK. The try/catch block handles this error and saves the tweet without a normalised location.

```
# If the Google API was successful then save their location and attempt to recognise the state.
if normalised_location["status"] == "OK":
    user.raw_location = tweet.user.location

    # Try to extract the normalised location and return false if it can't find one.
    # This is needed because sometimes the API returns an empty list, even if it says it was OK. :/
    try:
        user.normalised_location = str(normalised_location["candidates"][0]["formatted_address"]).lower()
    except KeyError:
        self.debug("Normalisation failed, saving {} with undefined location.".format(user.username), 3)
        return False
```

Extensions

Various things could be done to extend the functionality of the application. The most obvious thing is the implementation of tone/emotional analysis as an extra data point. The main thing blocking this is the prohibitive cost of using the API as the free plan is severely restricted. Otherwise, it would be a valuable addition as it creates a significant amount of more data that can be analysed and used to search/filter with.

The next thing is a proper scheduling system. Due to limitations with the time and API request limits, this wasn't implemented. This would mean the application can continue to collect and process tweets even when a user isn't connected. This is beneficial as it allows more tweets to be collected and analysed.

References

Various online documentation sites were used to assist the development of the web application. However, no specific resource tutorials etc were used. For completeness, these reference documents are below. Various stack overflow posts were also useful during debugging/troubleshooting steps.

https://leafletjs.com/reference-1.3.4.html
https://developers.google.com/places/web-service/search
https://azure.microsoft.com/en-gb/services/cognitive-services/text-analytics/
https://developer.twitter.com/en/docs.html

Appendix - Dockerfile

```
# Our base image is Ubuntu 18.04.
FROM ubuntu:16.04
MAINTAINER Jaimyn Mayer (hello@jaimyn.com.au)

# Update the repos and install the basics.
RUN apt-get update
RUN apt-get install -y \
   apt-utils \
   curl \
   python3 \
   python3-pip \
   vim
   # vim lyfe - makes it easier to troubleshoot in the container
if need be.

# Add the node repo and install it, then check version.
RUN curl -sL https://deb.nodesource.com/setup_8.x | bash
RUN apt-get update
RUN apt-get install -y nodejs
RUN nodejs -v

# Copy our codez.
ADD /app /app

# Set our default working directory to build our front end
(webpack etc.).
WORKDIR /app/frontend
RUN npm run build

# Set our default working directory to install python and run
flask.
WORKDIR /app/backend

# Upgrade pip then install the requirements.
RUN pip3 install --upgrade pip
RUN pip3 install -r requirements.txt

# Expose our port and start our app.
EXPOSE 8000
CMD uwsgi --http 0.0.0.0:8000 --module app:app --processes 1
--threads 8
```

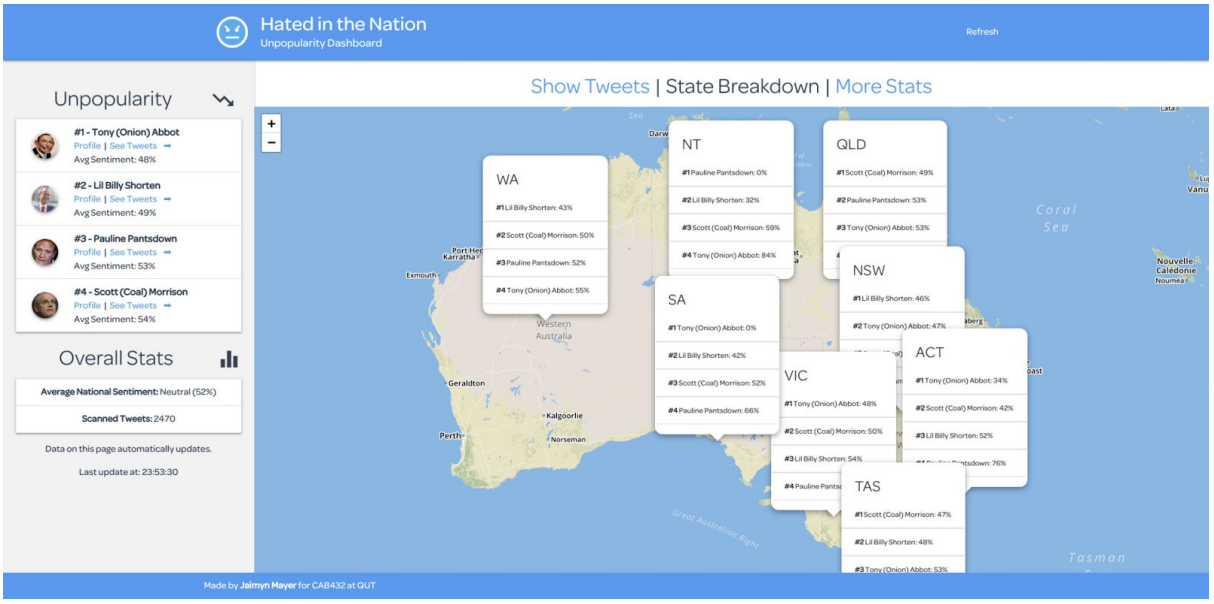
Appendix Usage Guide

The application is very simple and easy to use. An online version has been setup and is available at https://hatedinthenation.com . Please take into account this is hosted as a best effort attempt and there is no guarantee as to uptime/reliability.

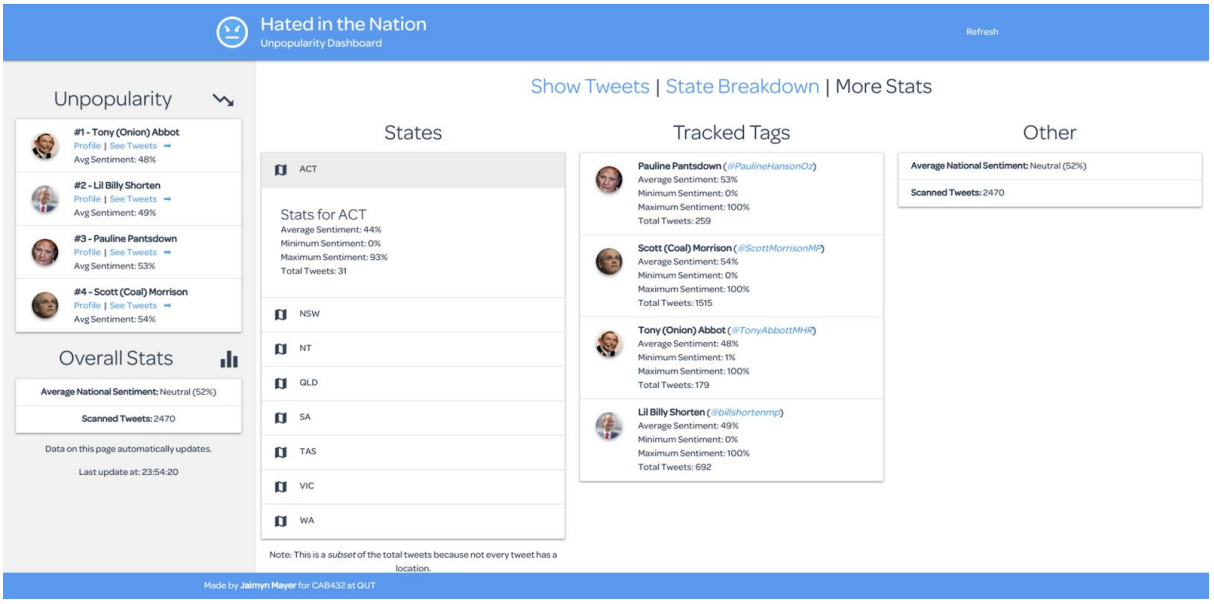
The main interface is very simple and easy to use. The leaderboard on the right has a list of the least popular tags that are being followed. Simply click on either of the blue links under each one to be taken to their twitter page or filter the tweets on the table to the right.

The table to the right is simply a list of every tweets stored in the database. The user can filter it by column, by clicking the arrows above each one. They can also use the search box in the top right corner or the navigation buttons in the bottom right to jump between pages.

The user can click on the “State Breakdown” link to see an overview of each state.



Clicking on the “More Stats” link will take them to the detailed stats page. Here the user can click on a tracked tag link to go to the twitter page or click on a state’s name for more info.



The user may see a warning after some time indicating the application will stop scraping tweets and will likely stop receiving data updates. However, if other users are using the application they may see data updates as they will continue scraping tweets.