General Sentiment of various fragments of the Australian society for some Australian politicians

Cluster and Cloud Computing (COMP90024) – Assignment 2

Web-app: http://172.26.134.46/ (accessible via a VPN client)
Youtube: Part 1 (Web App) https://youtu.be/vqI8GsmlbLs
Part 2 (Ansible) https://youtu.be/Tw8xTogUXGY
Gitlab: https://gitlab.com/AbhishekYadav142110/comp90024-2

Team 65

Abhishek Ugrasen Yadav (Student Id: 1176058)

Siddhesh Sawant (Student Id: 1236818)

Vanshree Bapat (Student Id: 1208561)

Manish Kumar Shaw (Student Id: 1230045)

Pranav Ashutosh Barve (Student Id: 1196722)

Abstract

Understanding the emotions and thoughts of the population of a country towards its politicians can be useful in multiple aspects. Predicting the results of an upcoming election, estimating a politician's likeability during the term, and overall response to the policies implemented by the government are just a few use cases. Our team (Group No. 65) has attempted to build a model that can calculate the general sentiment of the people of Australia towards their head politicians. Our model will generate a sentiment score based on the tweets collected from Twitter, as well as show its distribution against various Local Government Areas (LGA) of a particular state or territory. It also allows you to view this distribution of sentiment score across multiple parameters such as age, economic distribution and population diversity using maps as well as graphs. Australian population specific data of LGAs, age, economy and diversity were extracted from Australian Urban Research Infrastructure Network (AURIN). The Melbourne Research Cloud (MRC) has been used to host our system. This report further describes in detail, the architecture of our system, its functionality and the technologies used to build it.

Introduction

This project attempts to capture a general sentiment shared by the Aussies towards their (current) Premiers/Chief Ministers and the (current) Prime Minister. The user can select one of the nine politicians available in the navigator on the left in the webapp and he/she will be able to observe the state-wise distribution of an averaged sentiment score for the selected politician. We can further drill down by clicking on any state on the map. This gives an averaged distribution as per the Local Government Areas (LGAs) as defined by ABS 2016.

We have further tried to correlate this sentiment score against 3 factors:

1. **Economy Size:** By considering this factor, we have tried to find a correlation between the financial status of a LGA and the sentiment score for a politician in that area. Do rich people like a particular politician more? Is the population from the low-tier happy with this Premier/Chief Minister? We have tried to analyze similar scenarios using this data.

   Data source: RAI - Market Size Indicators (LGA) 2011, AURIN

2. **Age Group of the Population:** Do young people like a certain politician more? Maybe this particular politician has a personality that young people tend to like. Do adults like this politician, and if they do, is it because he/she comes up with policies that tend to suit them more? These kinds of questions can be answered by considering this factor.

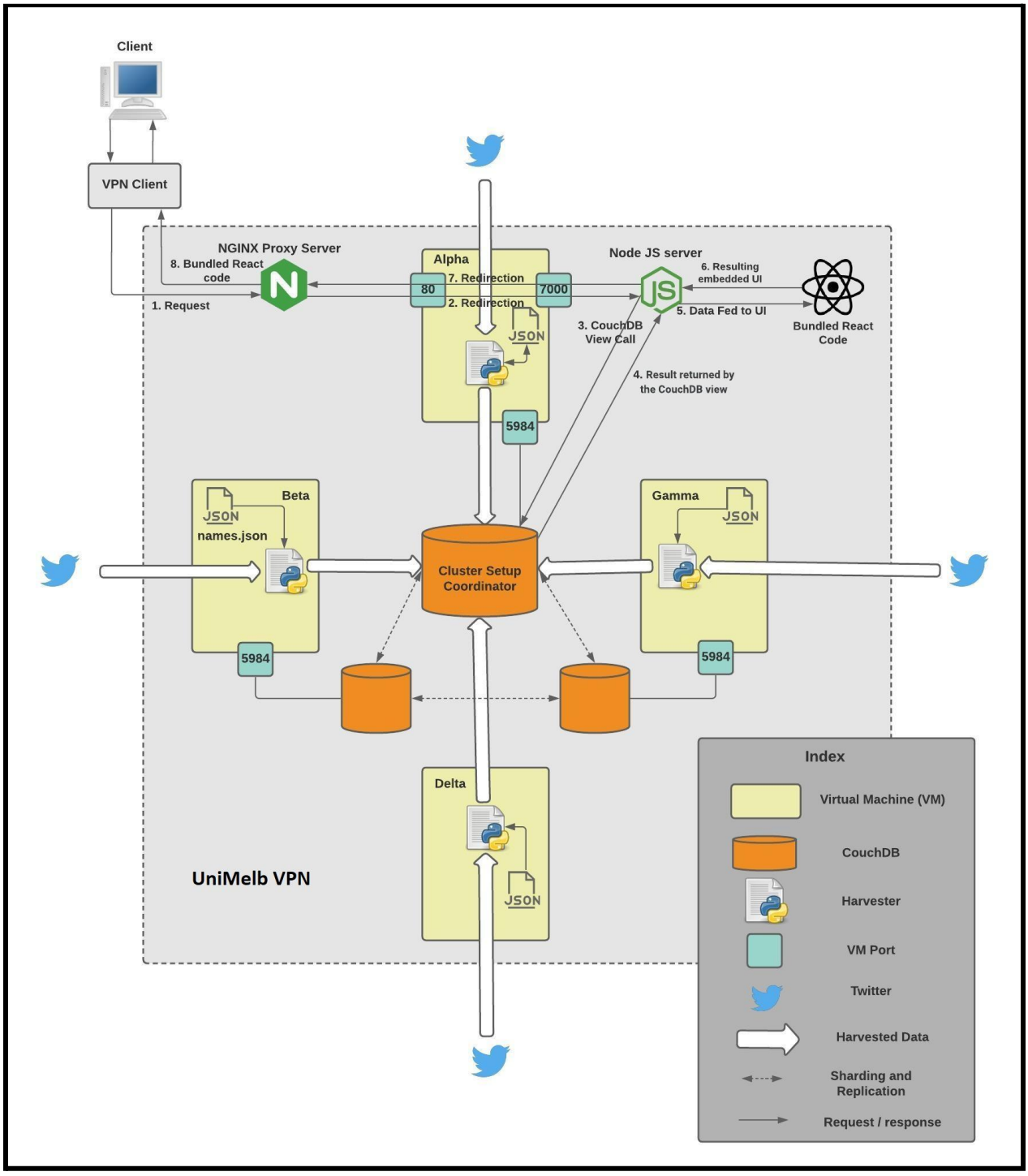
   Data source: LGA-G04[a,b] Age by Sex-Census 2016, AURIN

3. **Population Diversity:** Why do the Chinese or the Indians like this particular politician? Is it because he openly supports policies beneficial for migrants and international students? Do the Italians like a particular politician more than the British in this area? These and similar questions can be answered using this factor.

   Data source: LGA-P09 Country of Birth by Sex-Census 2016

Although limited in scope, this project can be extended further to give a “politician search” capability to the user wherein the user can search for any famous politician, and our system will capture the sentiment on various levels of areas. This can be a valuable model to predict the result of an upcoming election. Alternatively, this system can be extended to provide extra capabilities like time analysis of sentiment scores of various fragments of society for any politician. Do these people have started liking him more or less during this period? If they have started liking less, is it because of the statements or the policies that this politician has passed or made which affect these populations?

System Architecture



We have used 4 virtual machines (VMs): Alpha, Beta, Gamma, and Delta having Ubuntu 18.04 LTS as an operating system on top with pre-installed Docker. The former 3 have the following resources: 9GB RAM, 2 virtual CPUs (VCPUs), and 30GB disk while Delta has 4.5GB RAM, 1 VCPU, and 30GB disk. Since Delta has the least computing power, we have only used it for harvesting tweets.

Along with the harvester running, all the VMs except Delta have CouchDB installed on them too and are a part of a cluster with default sharding and replication configurations where CouchDB at Alpha is the cluster setup coordinator. Every harvester extracts tweets independently of the others and extracts only the tweets associated with the assigned politicians to the harvester. The assigned politicians are provided in the "names.json" file which is different for every harvester. For example, names.json for Alpha has Scott Morrison, Daniel Andrews, and Peter Gutwein listed in it, so a harvester running on Alpha will only extract tweets for these three. Similarly, Beta has a different list of politicians mentioned in its names.json so it will only extract tweets for them, and so on…

The extracted tweets are preprocessed before storing in the Couchdb. Each of the 9 politicians have a dedicated database for efficiency reasons. Preprocessing a tweet includes calculating the sentiment score for each tweet, extracting only the relevant keys of all the keys returned by Twitter and checking if a tweet is already stored using the tweet id, and then storing the tweet if it was not already stored. Our CouchDB cluster comprises 3 nodes, running inside a Docker container on Alpha, Beta, and Gamma with the default sharding and replication configurations.

Alpha also hosts the webapp using an NGINX proxy server running on port 80. The webapp runs on port 7000 and all the HTTP requests which are by default made on port 80 are redirected to the Node.js server running on port 7000 which serves a single HTML page along with a bundled React code to be rendered on the HTML page. This rendering is done on the client side by their respective browsers.

The webapp is accessible through Alpha's IP at http://172.26.134.46/ and is hosted inside the University of Melbourne VPN. Thus, the webapp is only accessible by connecting to the VPN through any VPN client (we've used Cisco AnyConnect).

Though not mentioned in the architecture explicitly, deployment of both the web app and the harvester is automated using Ansible.

Web App

The webapp is implemented using Node.js (backend) + React.js (frontend). The backend code is maintained under /app/server directory while the frontend code is maintained under /app/src and /app/public directories.

Backend

The backend specifically uses the **Express.js** framework under the Node.js hood. As mentioned earlier, we have considered all the states and territories of Australia and all the Local Government Areas (LGAs) within each state. The webapp uses the LGA information in various places, and hence the first thing the server does on startup is to check whether all the LGAs are loaded in the “all_lga” database. If the “all_lga” database does not exist it creates it before loading the LGA data. We have used the official CouchDB client for Node.js - nano, to connect to our CouchDB databases.

```
nano.db.list().then((dblist) => {
  if (!dblist.includes(`all_lga`)) {
    nano.db.create(`all_lga`).then(() => {
      const db_handle = nano.db.use(`all_lga`);
      ["VIC", "NSW", "ACT", "WA", "SA", "NT", "QLD", "TAS"].forEach((st) => {
        let lga_data = require(`../data/lga/${st}.json`);
        lga_data.features.forEach((ele) => {
          db_handle.insert({
            state_territory: st,
            lga_name16: ele.properties.lga_name16,
            lga_code16: ele.properties.lga_code16,
          });
        });
      });
    });
  }
});
```

For every LGA, we store in the “all_lga” its name and code according to ABS 2016 and the state/territory it is situated in.

The webapp also defines a POST API which takes the twitter username of a specific politician as a parameter and returns an average sentiment score towards that politician on state/territory level and also on LGA level for every state/territory.

```
API name: /api/summary
Request Body: { screen_name: <twitter_username> }
Response:
{
```

```
        lga_wise: {
                Ballarat: 0.023434,
                Yarra: -0.3465546,
                ...
        },
        state_wise: {
                VIC: 0.12644,
                NSW: 0.0543,
                ...
        }

}
```

Frontend

The frontend is implemented in **React.js**. We have used React.js for its salient features like client-side rendering and having the capability of developing single-page applications (SPA).

With old server-side rendering solutions, you built a web page—with PHP for example—the server compiled everything, included the data, and delivered a fully populated HTML page to the client. It was fast and effective. But… every time you navigated to another route, the server had to do the work all over again: Get the PHP file, compile it, and deliver the HTML, with all the CSS and JS delaying the page load to a few hundred ms or even whole seconds.

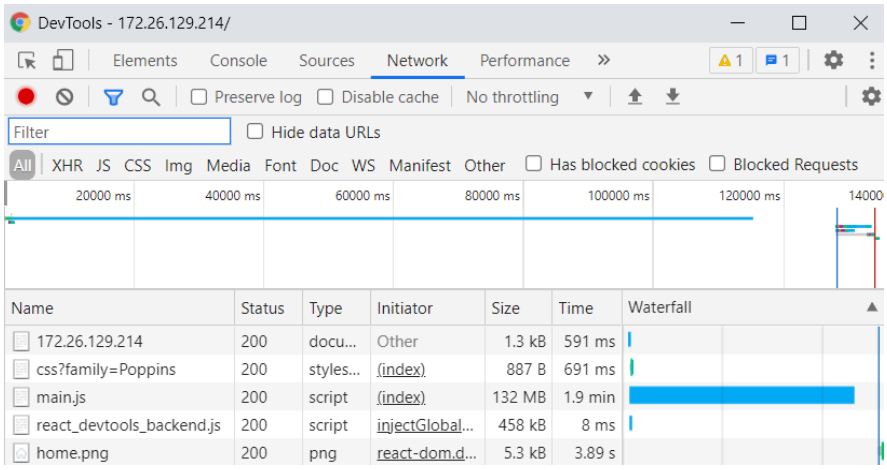
With a **client-side rendering** solution, you redirect the request to a single HTML file and the server will deliver it without any content (or with a loading screen) until you fetch all the JavaScript and let the browser compile everything before rendering the content. Under a good and reliable internet connection, it's pretty fast and works well. Thanks to Google and its ability to "read" JavaScript, it works pretty well, and it's even SEO friendly.

Also React supports building routers on the frontend. So, with every new route the browser does not have to make a request to the server but the React code can handle it independently. All the frontend logic along with routing is bundled into one single Javascript file and sent by the server as a response when the page first loads. The client code can thus run independently of the server until further data is needed for which an API request is sent to the server. The server then extracts data from CouchDB using the view defined and then does the necessary processing and formatting before sending a response to the client. This feature of React where it can handle all the route handling and dynamically change the UI without server's interference allows it to make **Single Page Applications (SPA)**.

Since all the client-side code is delivered in one-go the initial loading might take some time but all the subsequent requests are very fast since we have made a single-page app. While React provides even a faster solution to this, that did not suit our requirements.

All the React code as whole is segregated into different parts and is generally not optimised to be loaded in a browser as it is. We have also used a bundler called **Webpack** which as the name suggests bundles all the React code from different files in a single file and does the necessary optimisations like removing spaces, removing comments, replacing long variable names with shorter one, etc. The output that is a bundled Javascript file is then served to the browser. The client-side bundled code is available in **/app/dist/main.js** and the server-side bundled code is available in **/app/build/bundle.js.**

NOTE: *Our UI heavily depends on maps and maps are rendered with the help of GeoJSON files which are themselves quite bulky. The rendering is done on the client side itself. This makes our bundled main.js file too heavy, since the majority of its size is because of GeoJSON files being bundled together as well. Because of this reason, the initial loading of the web app may take quite some time in case of poor bandwidth. Till the main.js file is being loaded from the server, a loader with an appropriate message is rendered for time being.*



While we could have used a Content Delivery Network (CDN) or our server itself to load the GeoJSON files as and when required, this would have made the user experience poor since every time a user selected a new politician he/she would have to wait for the map to render. Instead, we chose to load everything at once on the first load and then after give the user a smooth journey.

React uses a special syntax called JSX (abbreviation for Javascript Expression) which is different from vanilla Javascript and since our browser only understands vanilla Javascript we have to first transpile and compile the JSX to vanilla Javascript. This is done with a tool called **Babel** which is used inside Webpack using babel-loader.

In React, **Components** let you split the UI into independent, reusable pieces, and think about each piece in isolation. Components are syntactically written in angled brackets < />. We have split the UI into following components:

Custom Components

1. **`<Navigator />`** : The section on the left where the politicians are listed and there's an About page option. This component is constantly present in all the UI screens.

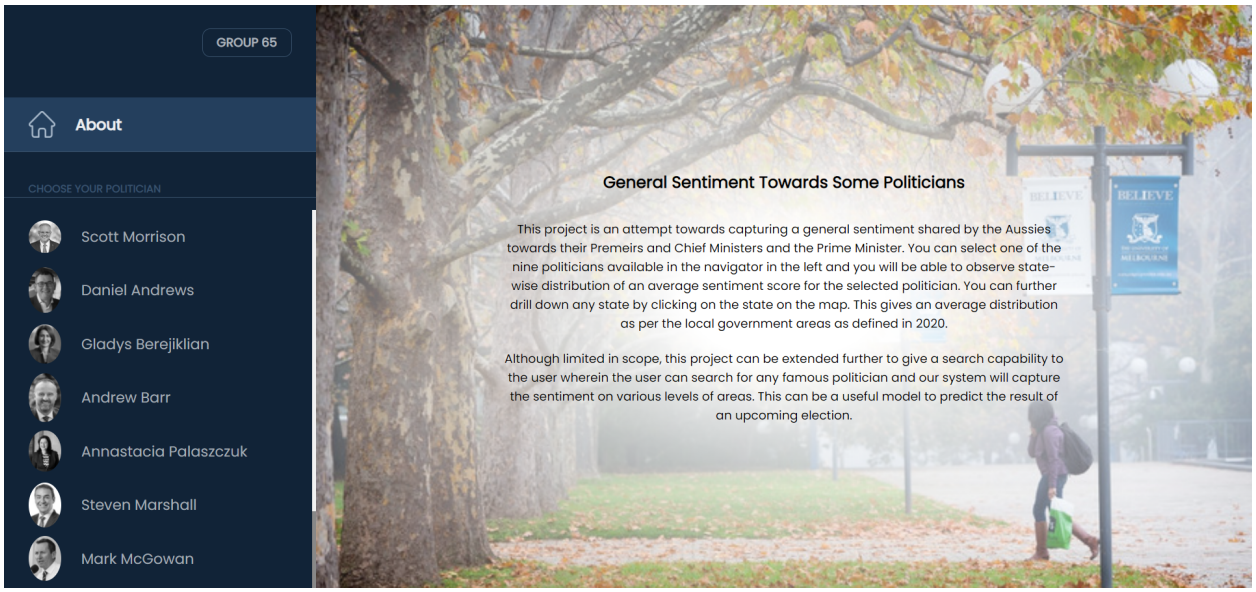
2. **<About />** : The About page.
3. **<AussieMap />**: The main UI when the Australian map with its states and territories is rendered. It internally renders a third-party component, `<AustraliaMap />` which is described below.
4. **<StateTerritoryMap />** : The main UI when you click on a particular state/territory in `<AustraliaMap />`. It internally renders many third-party libraries: `<Map />, <GeoJSON />, <ContinuousColorLegend />` which are described below.
5. **<ComparisonGraph />** : The modal which pops up when you click on the "Compare on graph" button. This renders either a bar graph (when comparing sentiment score against population diversity or age groups) or a scatter plot (when comparing sentiment score against economy size). It uses a range of third-party components to render different type of graphs: `< XYPlot />, <XAxis />, <YAxis />, <HorizontalGridLines />, <VerticalGridLines />, <MarkSeries />, <VerticalBarSeries />, <Hint />`. These are described below.
6. **<CountryModal />** : The modal which pops up when you click on the filter option. Filter is available for "Age wise distribution" and "Population diversity" comparisons.
7. **<TeamModal />**: The modal which pops up when you click on "GROUP 65" in the `<Navigator />`. This modal displays all the team members and their details.
8. **<App />** : The root component. All the above components are rendered by this component.

Third-Party Components

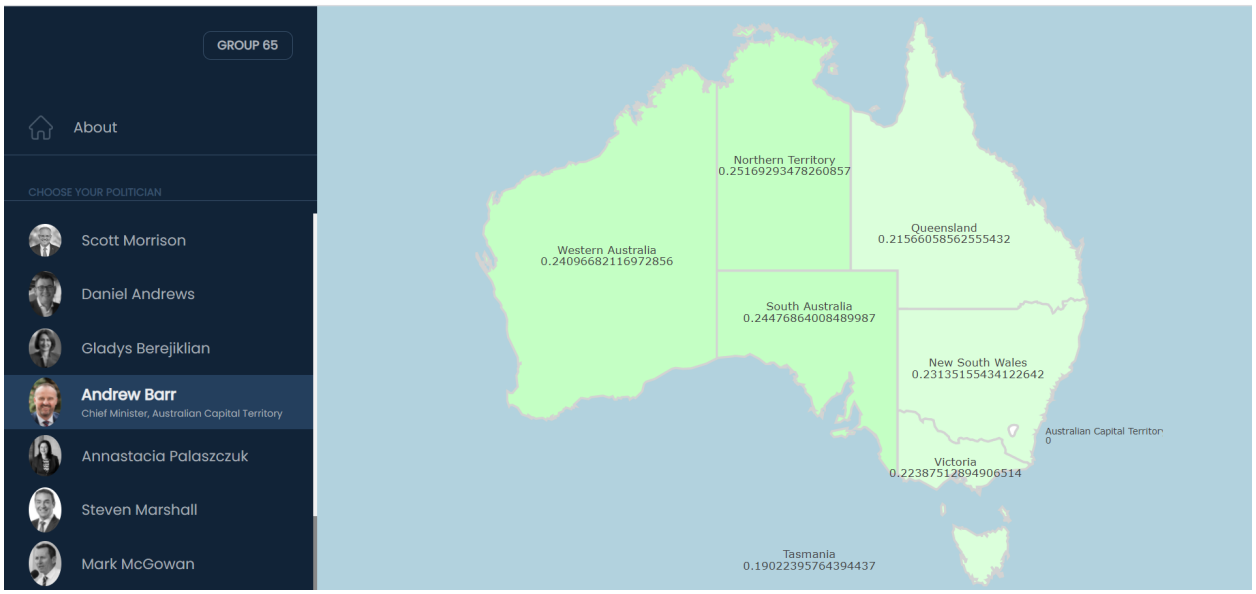
1. **<Loader />** : Third-party library for showing a loader from the time the data for a politician is requested from the Node.js server until the time a response (the data itself or an error) is received.
2. **<AustraliaMap />** : This is a simple customizable SVG map of Australia on HTML. This map shows state delimitations.
3. **<Map />** : A wrapper component for `<GeoJSON />`. It provides common map functions like zooming.
4. **<GeoJSON />** : It takes a geojson file as an input and renders the map as a SVG in HTML. This renders the various heatmaps of the states/territories.
5. **<ContinuousColorLegend />** : This renders a legend for the heatmaps displaying color codes and their meanings.
6. **<XYPlot />** : A wrapper component for all the graph components.
7. **<XAxis />, <YAxis />** : Components for rendering the x-axis and the y-axis respectively. They provide various customisation options according to the graph.
8. **<HorizontalGridLines />, <VerticalGridLines />** : Components for rendering grid lines behind a graph.
9. **<MarkSeries />** : A component for rendering data as a scatter plot.
10. **<VerticalBarSeries />** : A component for rendering data as a bar graph
11. **<Hint />** : A component for rendering tooltip when hovered on graph.

UI Description:

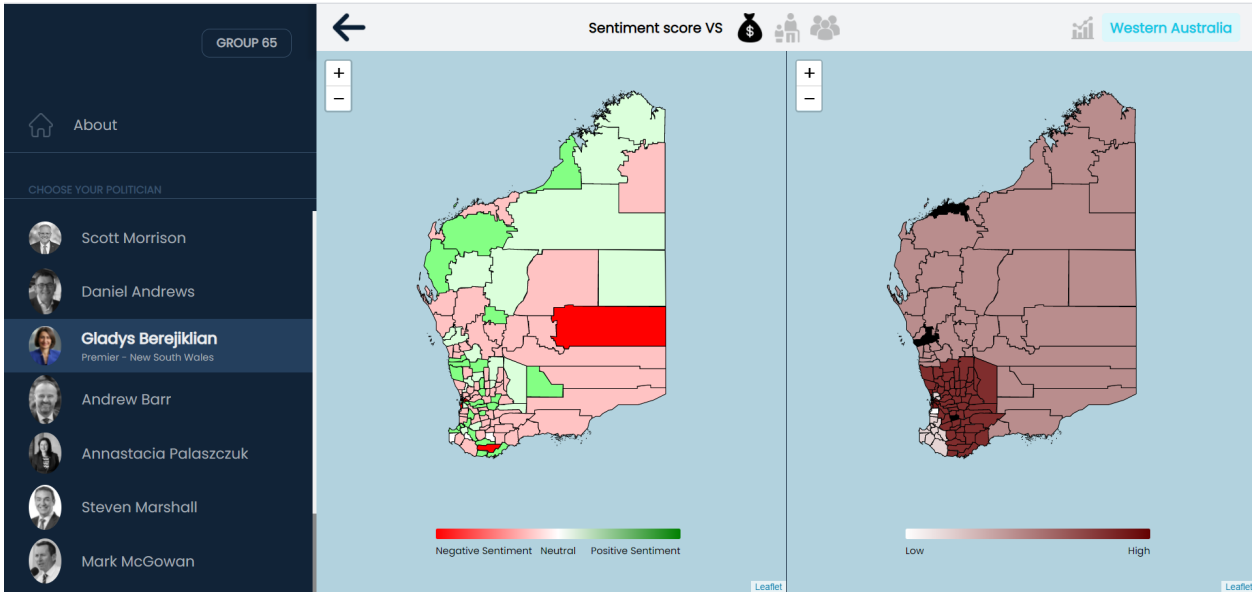
1. The project description is available under the "About" section. The <About /> component is rendered here.



2. There is a range of politicians for whom you can find sentiment scores. Select any one to know more. The `<AussieMap />` component is rendered initially whenever you select any politician.

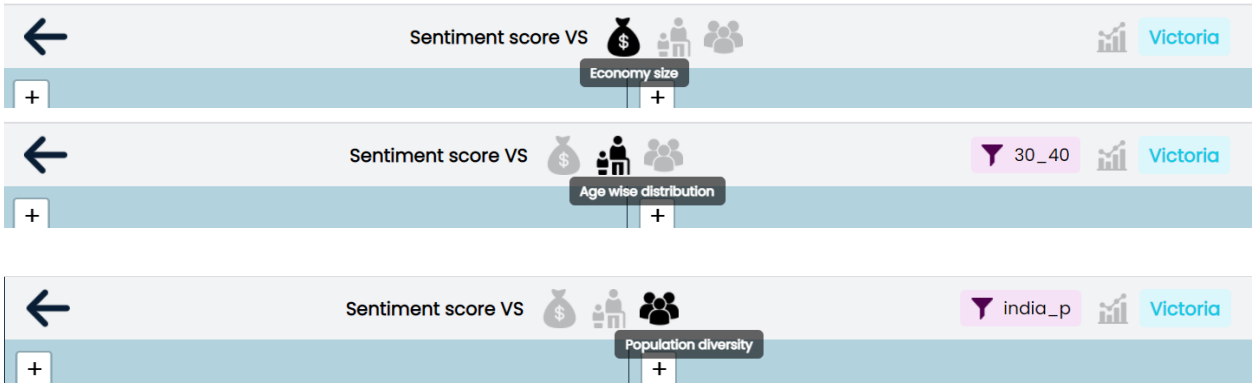


3. You can click on any state/territory to drill down to the LGAs of that state/territory. The `<StateTerritoryMap />` component is rendered here.



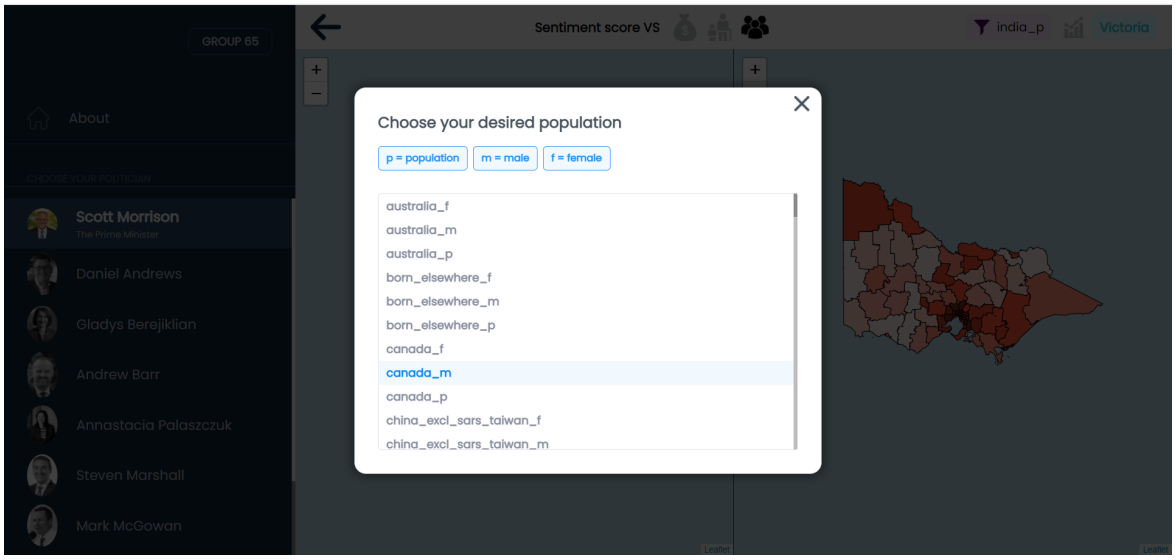
NOTE: *The color code for a given LGA is decided on a relative scale and not with respect to the absolute scale (for example, absolute scale for sentiment score is -1 to 1). This is done for better visualisation and also it is more convenient to compare different LGAs on the basis of relative color codes. The problem with absolute scale was that in most of the cases many LGAs would have scores in a particular range with little variation, thus all of them would have been marked with the same color which would have been a poor visualisation. For example, sentiment scores range from -1 (red, meaning negative sentiment) to 0 (white, meaning neutral sentiment) to 1 (green, meaning positive sentiment). For a particular politician in a particular state, almost all the LGAs had the score very close to 0 i.e. 0.0234 or 0.003434, or 0.00254, etc. Hence, on an absolute scale all were marked with white, which visually did not portray any information.*

4. You can compare the sentiment score for the selected politician LGA-wise against 3 factors per LGA:
   1. Size of the economy
   2. Age-wise population distribution
   3. Population diversity considering country of origin

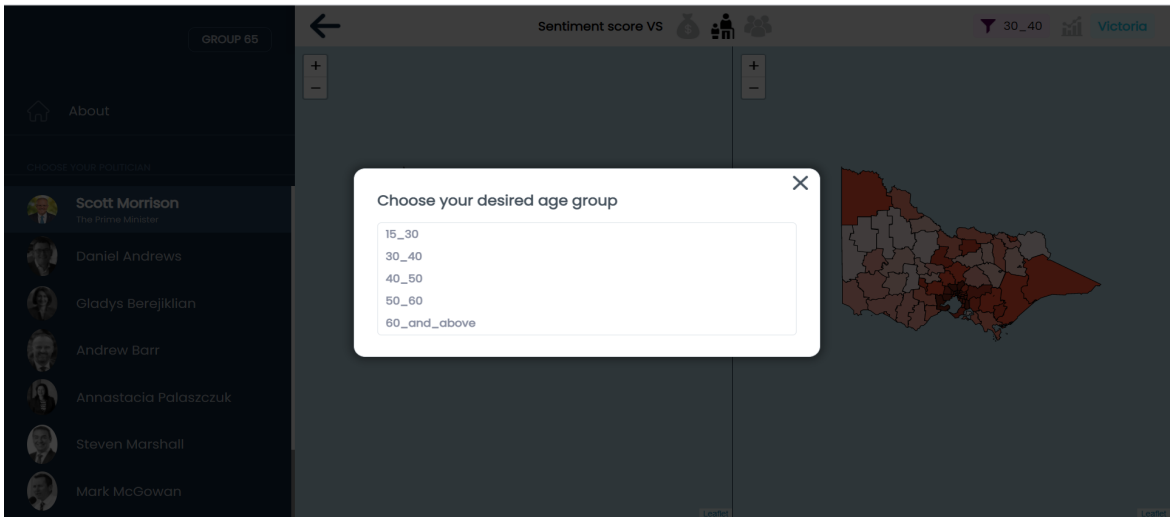


5. You can choose your desired filter in the last 2 factors. The `<CountryModal />` component is rendered here.

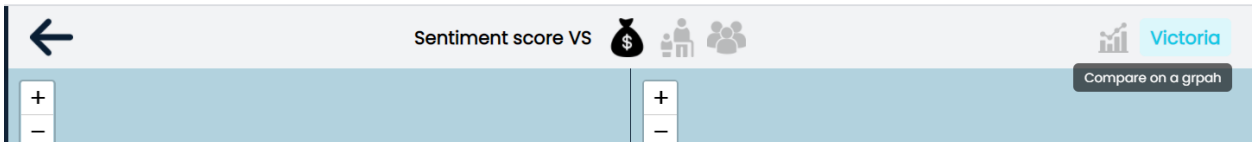
a. desired population fragment in the “Population diversity” section



b. age-groups in “Age wise population” distribution

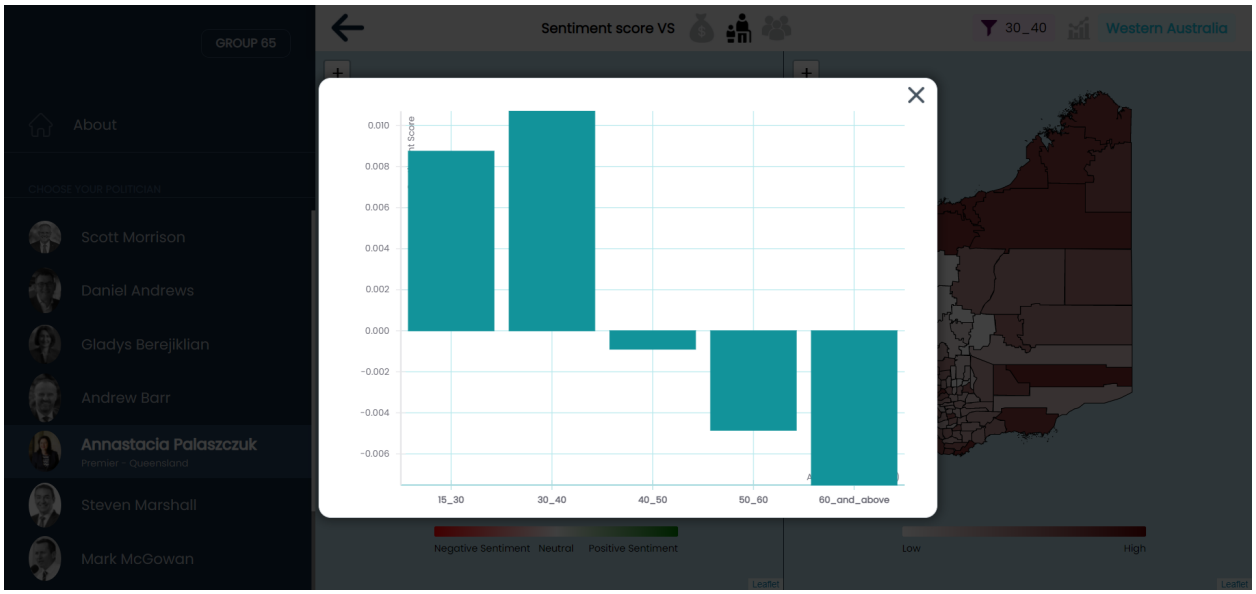


6. You can further compare the sentiment score against any of the 3 factors on a graph by clicking on the “graph” icon on the top right side of the screen. Click on the icon to render the `<ComparisonGraph />` component.

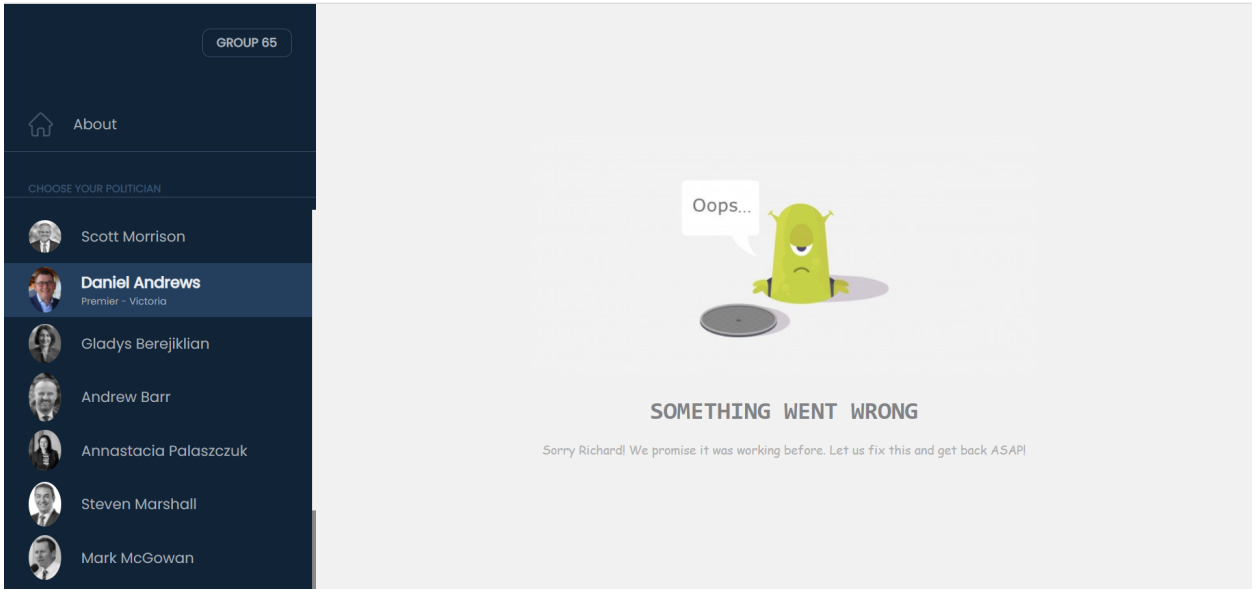


a. Comparison against economy size is done on a scatter plot as shown below

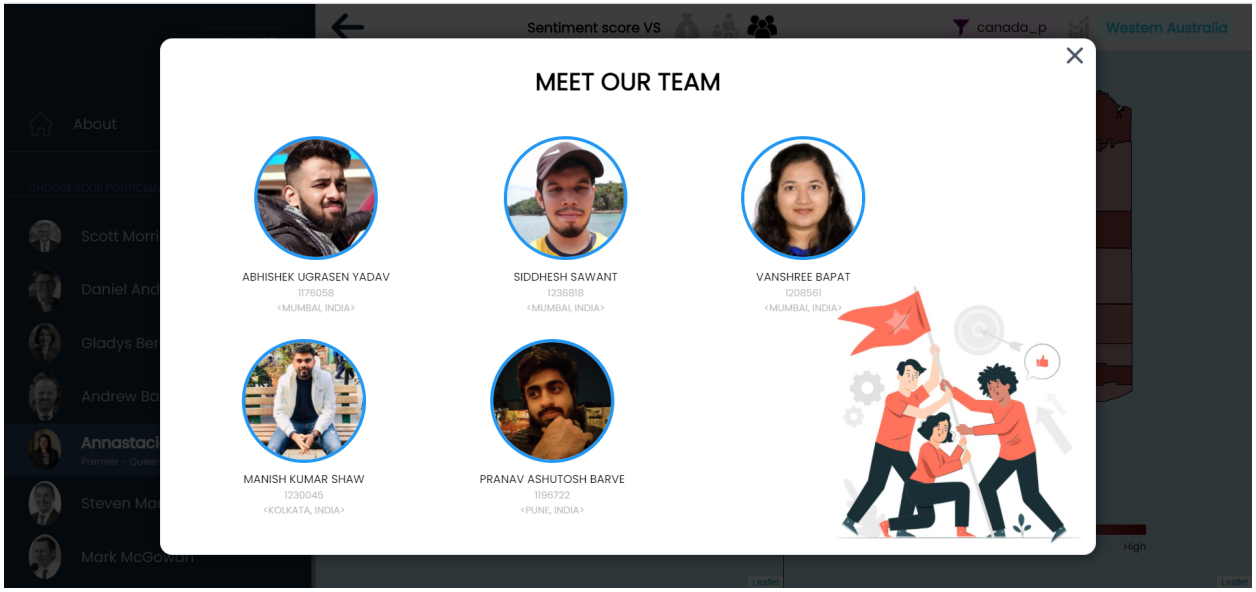
b. Comparisons against age groups and population diversity are done on a bar graph as show below:



7. We have also designed a page to display an error message when the API doesn't return the appropriate response.



8. On clicking on "GROUP 65" in the navigator, a modal opens which displays our team members with details like full name, student id, and location.



NOTE: *The maps are rendered using the spatialised JSON file from AURIN. However, these are not directly used. They are converted to GeoJSON format using an online tool because the React library we have used renders maps using only GeoJSON files.*

Docker

The webapp is also dockerised. You can follow the following steps to build an image and run the container:

1. Navigate to the root directory of the project: /comp90024-2
2. Enter the following command to build an image:

   ```
   docker build -t <image_name>:<tag> .
   ```

3. You can run the following command to check if the image is successfully build:

   ```
   docker images
   ```

   If you see your image listed in the output, then the image was built successfully.

4. Now, the next step is to build a container i.e a running instance of an image. Use the following command to build a container:

docker run -d -p 7000:7000 --name <container_name> <image_name>:<tag>,

for running the webapp on port 7000 of your instance. Since we redirect HTTP calls to port 7000, we suggest mapping container's port 7000 to instance's port 7000. If you wish to use some other port, you've to change it in the NGINX configuration file too.

5. You can confirm if the container was build successfully by running:

   docker ps

6. If you can see your container name in the output, you can visit the IP address of your host and see the webapp running!

Manually Running the Server

You can also start the server manually by navigating to /comp90024-2/app and running the following commands:

1. Install third-party dependencies:

   npm i

2. Set the appropriate environment variables. We've created a ".env_sample" file for reference. You can copy content of .env_sample to your .env file and then set your environment variables.

   cp .env_sample .env

3. Run the server:

   npm run dev

4. If the above command gives no error, you can visit the IP address of your host and see the webapp running!

Couchdb

Apache CouchDB is a document-oriented NoSQL database that is open source. It stores data in JSON, queries in JavaScript using MapReduce, and provides an API over HTTP. As per our architecture, we have installed CouchDB in three instances (alpha, beta and gamma). To install CouchDB we used Docker by running the following commands.

We used ibmcom/couchdb3 image with version 3.1.1. Here we use the docker run command to start the container having an image of CouchDB. We specify ports to open with -p attribute. The 5984 port is used for connections with the outside world and port 9100-9200 are used by other CouchDB instances to communicate internally. We also specify environment variables such as username, password and secret key(cookie). We mount a volume to the container using -v attribute. Run the below command to fetch a CouchDB image and start a container.

```
docker run -d -p 5984:5984 \
-p 9100-9200:9100-9200 \
--name couchdb${node} \
--env COUCHDB_USER=${user} \
--env COUCHDB_PASSWORD=${pass} \
--env COUCHDB_SECRET=${cookie} \
--env ERL_FLAGS="-setcookie \"${cookie}\" -name \"couchdb@${node}\"" \
-v /mnt/vdb/couchdb:/opt/couchdb/data \
ibmcom/couchdb3:${VERSION}
```

You can check whether couchDB is running or not by running below command

```
curl http://${node}:5984
```

After the CouchDB container was running on all three servers we created a cluster. To set up the cluster we referred to the CouchDB official documentation page. The link to the documentation is given below.

https://docs.couchdb.org/en/latest/setup/cluster.html

After the setup was complete. We created a total of 10 databases. All 8 Premier's/Chief Ministers and the Prime Minister have their separate databases and the last database named as ‘all_lga’ lists all the 544 LGAs in Australia as per ABS, 2016.

Databases

Name	Size	# of Docs
abarrmla	19.3 MB	2408
all_lga	121.3 KB	544
annastaciamp	283.5 MB	77272
danielandrewsmp	158.7 MB	149530
fanniebay	3.3 MB	324
gladysb	234.9 MB	72504
gutweinteam	10.2 MB	1014
markmcgowanmp	25.3 MB	21137
marshall_steven	90.5 MB	9015
scottmorrisonmp	261.9 MB	331031

A document of all_lga looks like:-

The state_territory key stores the state/territory which the LGA belongs to and lga_name16 stores the LGA name as defined by ABS, 2016.

A typical document of a database for any of the 9 politicians looks like

```
{
  "_id": "005cb3f4a25927504d1ca75a7801a8e9",
  "_rev": "3-3554aa9499909f275e5c218a22d91be6",
  "tweet_id": "1360870847599157252",
  "tweet": "@ScottMorrisonMP Why haven't vaccines started to be rolled out. Another lie - another failure.",
  "user_id": "777333275660476416",
  "date": "2021-02-14",
  "place": "",
  "geo": "",
  "near": [
    "Buloke ",
    "Campaspe ",
    "Brimbank "
  ],
  "language": "en",
  "sentiment_score": {
    "neg": 0.216,
    "neu": 0.784,
    "pos": 0,
    "compound": -0.5106
  },
  "state_territory": "VIC"
}
```


The key 'tweet_id' stores the id of the tweet which is used later on to handle duplicate tweets. The 'tweet' key contains the text of the tweet. The 'user_id' key stores the id of the user who tweeted and the 'date' key stores the date at which the tweet was made. The 'place' key and 'geo' key is used to store the geolocation of the tweet but unfortunately, only 1% of tweets has that data. Next is the 'near' key which stores the LGA name from which the tweet was harvested. A tweet can be harvested from LGAs that are in close vicinity. The 'language' key contains the language of the tweet. Next is the 'sentiment_score' key which was crucial in our analysis and stores the sentiment score for that particular tweet. The last key is 'state_territory' which records from which state the tweet was tweeted.

Map Reduce

The map stage is the first phase of the procedure. We output key/value pairs while mapping on CouchDB. Our Map function emits key value as tweet_id and value as the document. To check the duplicate tweets we needed to emit tweet_id in the map function. Given below is our map function:

```
function (doc) {
  emit(doc.tweet_id, doc);
}
```

The reduce function accepts two parameters key and value. Since we needed the document to append the LGA name in the near key we have return values(document) in the reduce function:

```
function (keys, values, rereduce) {
  return values
}
```

Harvester

Collection and extraction of the data is one of the key aspects of our project. Our method of gathering substantial amounts of relevant data, cleaning and managing it to produce required results has been described below.

The data extraction process began by using the Twitter APIs to fetch data specific to our problem statement. It requires extraction of data specific to some of the major politicians of Australia that includes the Prime Minister of Australia, the Premiers/Chief Ministers of the states and of the territories. Our initial methods of extraction conceptually included the tweets tweeted by these politicians, replies on these tweets, tweets of their followers regarding them as well as tweets tagging them. In order to calculate the sentiment score based on the Local Government Area, getting the location of these tweets was important.

However, due to limitations of the APIs as well as the narrow scope of the project, the tweets that we could extract using this method were not only insufficient, but also the data generated was sparse, that is a lot of parameters (most importantly location) were blank. Less volume of data along with sparsity would have led to less accurate results.

Although there were workarounds to extract sufficient volume of data, our problem statement heavily relies on the location of tweets, extracting which using the Standard Twitter Search API was not feasible since 2% of all the tweets are geotagged in general and our topic is too specific to get sufficient amount of geotagged tweets.

In order to resolve these issues we proceeded to use a scraping tool called ‘Twint’ to gather the required data. Twint is a Twitter scraping tool developed in Python that allows scraping tweets

from - or - addressed to specific users, related to specific topics, trends, locations, languages, time period etc. It allowed us to overcome the volume limitation of the Twitter APIs as well as avoid sparsity in the data gathered. Twint also provided extraction of data directly in a json file which helped us initially for testing. The Twint Scraping tool can be referred from github (https://github.com/twintproject/twint) and follows a relatively easy installation process.

Installation of Twint

Local Installation

In order to install twint locally, we had to clone the git repository and install its dependencies. Installation of Microsoft Visual C++ is also required to run this tool. On its completion, we proceeded to install twint using the command: `pip3 install twint`
Installation was verified by running basic Twint commands on GitBash

Virtual Installation

Similar to local installation, the twint repository was cloned on the instances, however getting the tool up and running was not an easy task due to multiple proxy issues. On modifications in the base twint code, we were able to start scraping tweets on the instances. We also had to configure proxy settings in our data harvester program as follows:

```
c.Proxy_host = "wwwproxy.unimelb.edu.au"
c.Proxy_port = 8000
c.Proxy_type = "http"
```

Extraction and Storage of Data

Data can be extracted from Twint and stored directly into a json file. This can be done either through command line or through a script written in Python. After trying out the working of Twint through the command line we proceeded to write a script 'harvest.py'. This script included the extraction of required fields, iterating the scraping of data over defined time intervals, storing unique tweets in the CouchDB, calculating and storing the sentiment score of the extracted tweets.

Extraction of required fields

Twint provides flags to filter and extract data as per user requirement. Few flag examples are as below

```
Username            (string) - Twitter user's username
Near                (string) - Near a certain City
Since               (string) - Filter Tweets sent since date
```

```
Until                (string) - Filter Tweets sent until date
To                   (string) - Display Tweets tweeted to the specified user.
```

The -to flag was used to scrape the tweets addressed to each politician:

```
c.To = screen_name #screen_name= 'scottmorrisonmp'
```

The screen name is fetched from the names.json file present on each of the instances that contains the data of respective politicians assigned to each harvester. This allowed us to scrape for four politicians simultaneously and automatically continue to the next one on completion of scraping of the first four.

The “near” parameter helped us get location specific tweets and it was based on the 544 Local government Areas of Australian States and Territories as defined by ABS 2016. It basically used each LGA as a value for the near parameter.

```
c.Near = formatted_lga_name #formatted_lga_name= 'Buloke'
```

Filtering out the required data made the scraping faster and was stored in a customized way as below:

```
c.Custom["tweet"] = ["id", "tweet", "date", "geo", "near", "language", "user_id",
"place"]
```

Iteration through time intervals

In order to avoid discontinuity in the scraping process, we had included an array in the script that contains time periods of one month each. So, at a given time, the extraction was carried out for person A, for a specific month for all the LGAs. Once completed, it moved on to the next month. Currently we have considered data between the time period ‘’ and ‘’.

```
c.Since = since #since=2020-01-01
c.Until = until #until=2020-02-01
```

Storing unique tweets and sentiments in the Database

Extraction using twint leads to generation of duplicate tweets. In order to maintain a clean database, it was required to remove this redundancy and store only the unique tweets. This was done by first checking the existence of a newly scraped tweet in the database and if present, the tweet was then discarded. However, the near parameter of the duplicate tweet was checked and if the LGA was different than the original tweet, the LGA was appended to the original tweet.
If the tweet was not present in the database, it was added as a new document and included the sentiment score calculated for it. The sentiment score is calculated using the NLTK python library.

```
sia = nltk.sentiment.SentimentIntensityAnalyzer()
```

```
filtered = {
    'tweet_id': getattr(extracted, 'id_str'),
    'tweet': getattr(extracted, 'tweet'),
    'user_id': getattr(extracted, 'user_id_str'),
    'date': getattr(extracted, 'datestamp'),
    'place': getattr(extracted, 'place'),
    'geo': getattr(extracted, 'geo'),
    'near': [getattr(extracted, 'near')],
    'language': getattr(extracted, 'lang'),
    'sentiment_score': sia.polarity_scores(getattr(extracted, 'tweet')),
    'state_territory': place["doc"]["state_territory"]
    }
this_premier_db.save(filtered)
```

This was the working of the harvest.py explained in brief. On deploying this script on the instances along with the names.json files, following commands were used to execute the scripts and monitor them:

Execution: `sudo nohup python3 <filename.py> &`
"&" allows running the harvester as a background process and "nohup" allowed us to continue scraping even after our connection with the servers was discontinued. The output generated was stored in a file named nohup.out

Monitoring: `sudo tail -f nohup.out`
The output as and when generated keeps on appending in the initial outputs present in the nohup.out file. In order to keep a track on the scraping, using this command allowed us to check the latest status.

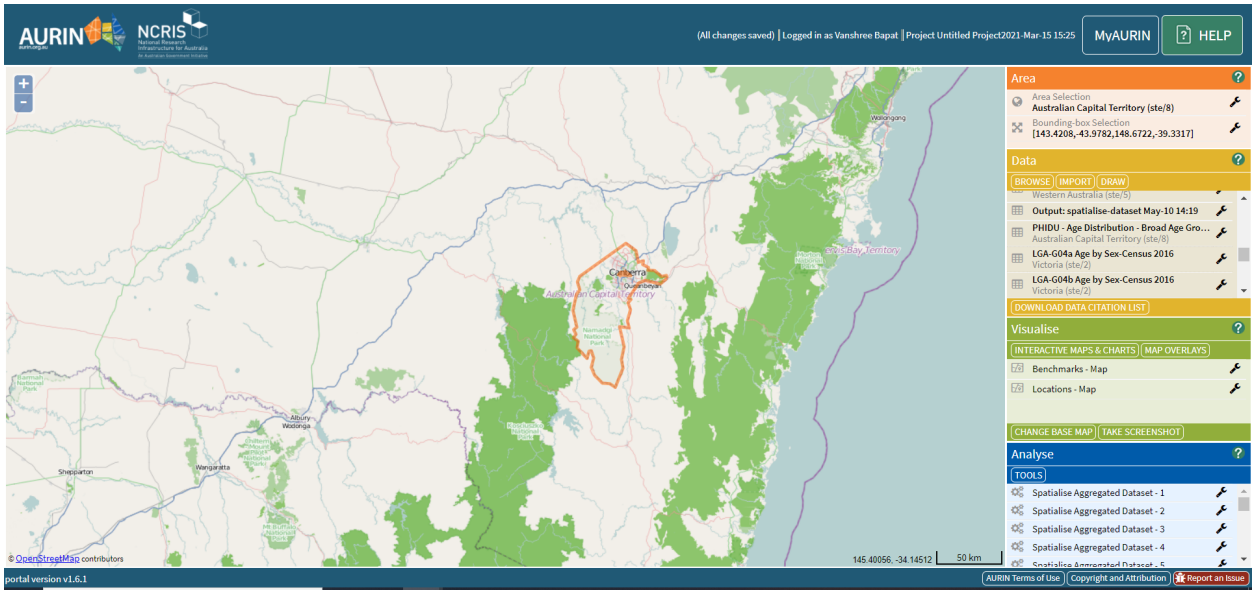
As the scraping continued, we came across another difficulty of the scraping stopping midway through, due to multiple errors. It was not feasible to begin the entire process again as it would have been an unnecessary consumption of resources and time. In order to avoid this, a script ‘check.py’ was developed and deployed on the instances. At the end of the harvester script, a code snippet storing the LGA name and the date parameters of the tweets collected in a .txt file has been written. The check.py script restarts the harvester script in case the scraping stops midway and the last date parameter is considered to start the process again, thus effectively limiting the duplicate scraping.

```
while True:
    if count==0:
        time.sleep(1)
        count=+1
    else:
        time.sleep(300)
    if not is_running(check_filename):
        count+=1
        print(count)
        os.system("nohup python3 "+check_filename+" &")
```

The harvested data could be monitored through CouchDB’s Fauxton interface running on port 5984 on ” /_utils” path using the instance’s ip addresses.

Australian Urban Research Infrastructure Network (AURIN)

As the system is specific to the Australian continent, we needed to fetch the appropriate data from a verified source. The Australian Urban Research Infrastructure Network (AURIN) is a one-stop network that hosts thousands of datasets across multiple topics and provides tools to extract, analyse and visualize data as per our requirements.



Our project displays the sentiment score across 3 parameters: age, economic distribution and population diversity. After setting the appropriate level of area selection, the data specific to these was fetched from the following datasets:

1. **Economy Size**

   In order to help understand the distribution of the economic status across the LGAs of Australia by considering the attribute “Size of Economy” we used the “RAI - Market Size Indicators (LGA) 2011, AURIN” dataset.

RAI - Market Size Indicators (LGA) 2011

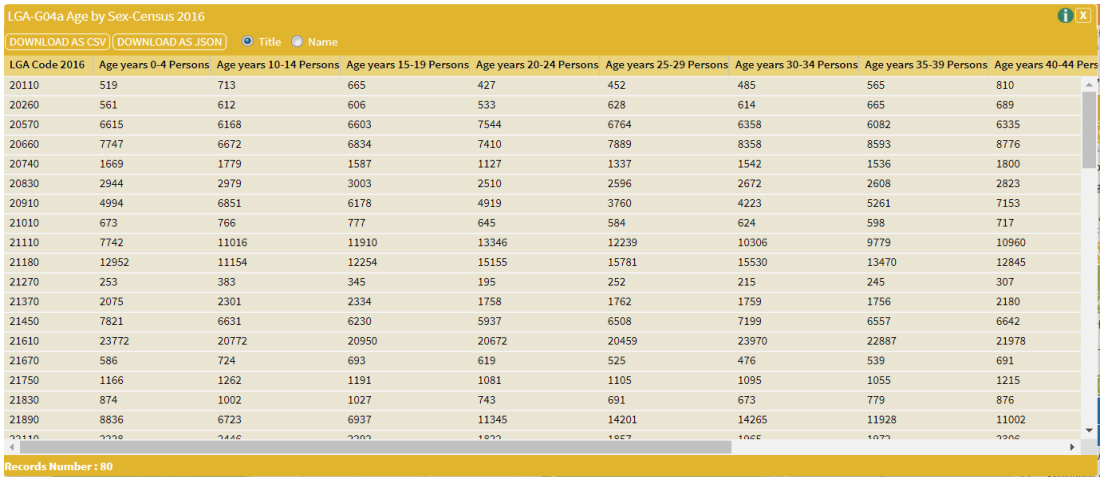
DOWNLOAD AS CSV | DOWNLOAD AS JSON | Title | Name

LGA Code	LGA Name	Size of Economy	Size of Economy Ranking	Size of Economy Standard Deviati	Market Size Indicator Standard De	Market Size Indicator Ranking	ogc_fid	State
20110	Alpine (S)	5048135000	402.000	-0.460	-0.430	392.000	153	Vic
20260	Ararat (RC)	6562775000	287.000	-0.190	-0.350	355.000	154	Vic
20570	Ballarat (C)	4043460000	465.000	-0.640	-0.560	413.000	155	Vic
20660	Banyule (C)	11178840000	75.000	0.650	1.320	50.000	156	Vic
20740	Bass Coast (S)	8518155000	105.000	0.170	0.270	118.000	157	Vic
20830	Baw Baw (S)	8518155000	105.000	0.170	0.270	118.000	158	Vic
20910	Bayside (C)	15710155000	22.000	1.480	1.440	47.000	159	Vic
21010	Benalla (RC)	5048135000	402.000	-0.460	-0.430	392.000	160	Vic
21110	Boroondara (C)	13281120000	62.000	1.040	1.080	57.000	161	Vic
21180	Brimbank (C)	12936170000	68.000	0.970	2.250	16.000	162	Vic
21270	Buloke (S)	6562775000	287.000	-0.190	-0.350	355.000	163	Vic
21370	Campaspe (S)	5033630000	412.000	-0.460	-0.580	426.000	164	Vic
21450	Cardinia (S)	20364140000	12.000	2.330	3.060	10.000	165	Vic
21610	Casey (C)	20364140000	12.000	2.330	3.060	10.000	166	Vic
21670	Central Goldf...	4043460000	465.000	-0.640	-0.560	413.000	167	Vic
21750	Colac-Otway ...	5408890000	387.000	-0.400	-0.560	407.000	168	Vic
21830	Corangamite ...	5408890000	387.000	-0.400	-0.560	407.000	169	Vic
21890	Darebin (C)	11178840000	75.000	0.650	1.320	50.000	170	Vic

Records Number : 79

2. **Age Group of the Population**

The data set used for considering the distribution of age groups across the LGAs, we used the “LGA-G04[a,b] Age by Sex-Census 2016, AURIN” data sets. It includes age groups in an interval of 5 years from 0 years to 100+ years. We modified and combined the extracted data as required by our problem statement. This information is based on the 2016 Census.



LGA-G04a Age by Sex-Census 2016

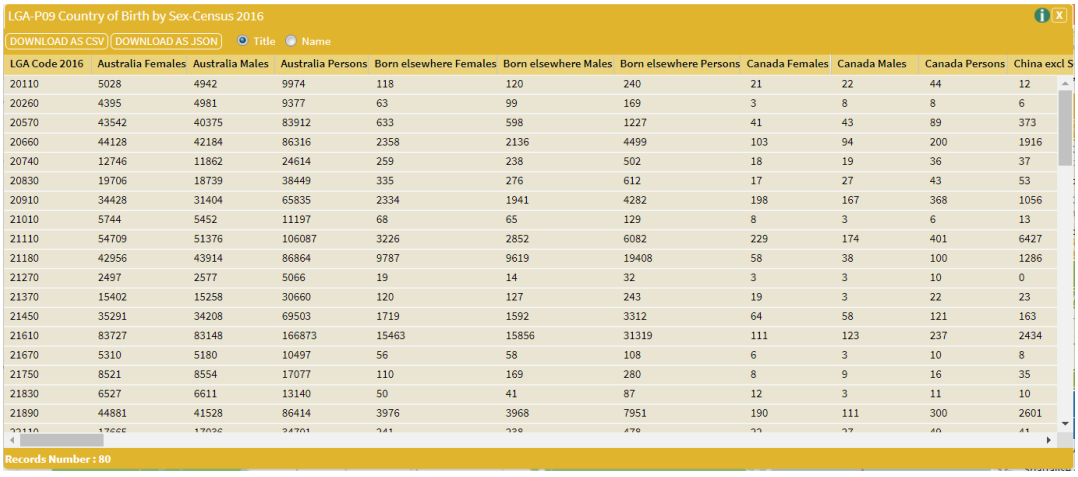
DOWNLOAD AS CSV | DOWNLOAD AS JSON | Title | Name

LGA Code 2016	Age years 0-4 Persons	Age years 10-14 Persons	Age years 15-19 Persons	Age years 20-24 Persons	Age years 25-29 Persons	Age years 30-34 Persons	Age years 35-39 Persons	Age years 40-44 Pers
20110	519	713	665	427	452	485	565	810
20260	561	612	606	533	628	614	665	689
20570	6615	6168	6603	7544	6764	6358	6082	6335
20660	7747	6672	6834	7410	7889	8358	8593	8776
20740	1669	1779	1587	1127	1337	1542	1536	1800
20830	2944	2979	3003	2510	2596	2672	2608	2823
20910	4994	6851	6178	4919	3760	4223	5261	7153
21010	673	766	777	645	584	624	598	717
21110	7742	11016	11910	13346	12239	10306	9779	10960
21180	12952	11154	12254	15155	15781	15530	13470	12845
21270	253	383	345	195	252	215	245	307
21370	2075	2301	2334	1758	1762	1759	1756	2180
21450	7821	6631	6230	5937	6508	7199	6557	6642
21610	23772	20772	20950	20672	20459	23970	22887	21978
21670	586	724	693	619	525	476	539	691
21750	1166	1262	1191	1081	1105	1095	1055	1215
21830	874	1002	1027	743	691	673	779	876
21890	8836	6723	6937	11345	14201	14265	11928	11002

Records Number : 80

3. Population Diversity

To understand and map the distribution of the diversity of the population across the LGAs, we used the “LGA-P09 Country of Birth by Sex-Census 2016” dataset. It contains the total number of people belonging to a specific country and currently residing in Australia thus allowing us to understand the population diversity in Australia. This information is based on the 2016 Census.



LGA-P09 Country of Birth by Sex-Census 2016

DOWNLOAD AS CSV | DOWNLOAD AS JSON | Title | Name

LGA Code 2016	Australia Females	Australia Males	Australia Persons	Born elsewhere Females	Born elsewhere Males	Born elsewhere Persons	Canada Females	Canada Males	Canada Persons	China excl S
20110	5028	4942	9974	118	120	240	21	22	44	12
20260	4395	4981	9377	63	99	169	3	8	8	6
20570	43542	40375	83912	633	598	1227	41	43	89	373
20660	44128	42184	86316	2358	2136	4499	103	94	200	1916
20740	12746	11862	24614	259	238	502	18	19	36	37
20830	19706	18739	38449	335	276	612	17	27	43	53
20910	34428	31404	65835	2334	1941	4282	198	167	368	1056
21010	5744	5452	11197	68	65	129	8	3	6	13
21110	54709	51376	106087	3226	2852	6082	229	174	401	6427
21180	42956	43914	86864	9787	9619	19408	58	38	100	1286
21270	2497	2577	5066	19	14	32	3	3	10	0
21370	15402	15258	30660	120	127	243	19	3	22	23
21450	35291	34208	69503	1719	1592	3312	64	58	121	163
21610	83727	83148	166873	15463	15856	31319	111	123	237	2434
21670	5310	5180	10497	56	58	108	6	3	10	8
21750	8521	8554	17077	110	169	280	8	9	16	35
21830	6527	6611	13140	50	41	87	12	3	11	10
21890	44881	41528	86414	3976	3968	7951	190	111	300	2601

Records Number : 80

Ansible

Ansible is an automation tool for deployment. Ansible is used for deploying and setting up a virtual machine on Melbourne Research Cloud. All the code is written in YAML format. The ansible code is divided into small snippets of codes which each one having certain specific functions. We have written Ansible scripts to deploy both the Web application and the Harvester.

Host vars
This folder stores all the data required to build a machine. All the data related to volume names, flavours, instance names are present. Data regarding which security groups are attached to be instances is stored here.

Inventory
There is an inventory file that stores all the variables required to run the ansible code on the host. The IP addresses of instances are stored here. Given below is our inventory file and we have added a group called ‘webservers’ and variables for ssh-ing into the instance.

```
[webservers]

[all:vars]
ansible_ssh_common_args="-o StrictHostKeyChecking=no"
ansible_user=ubuntu
ansible_ssh_private_key_file=~/.ssh/master_key.pem
```

Setting Local Host
Basic packages need to be installed on the localhost machine required to run ansible script. ‘pip’ is installed and updated. ‘Openstacksdk’ is also installed on the localhost machine.

Security-group
Security groups are created in this file and attached to the instances.

```
security_groups:
  - name: ssh
    description: "Group for SSH access"
    protocol: tcp
    port_range_min: 22
    port_range_max: 22
    remote_ip_prefix: 0.0.0.0/0

  - name: http
    description: "Group for HTTP"
    protocol: tcp
    port_range_min: 80
    port_range_max: 80
```

```
          remote_ip_prefix: 0.0.0.0/0
```

Since we are using Nginx as our proxy server we only need these two ports to be open in our web servers.

Create-instance
A single instance is created in this file. All the parameters required to create the instance are fetched from the data file. After the instance is created, the host is connected to the remote machine with an openssh connection.

```
- name: Create an instance
  os_server:
    name: '{{ item.name }}'
    image: '{{ instance_image }}'
    key_name: '{{ instance_key_name }}'
    flavor: '{{ instance_flavor }}'
    availability_zone: '{{ availability_zone }}'
    security_groups: '{{ sg_names }}'
    volumes: '{{ item.volumes }}'
    auto_floating_ip: yes
    wait: yes
    timeout: 600
    state: present
  loop: '{{ instances }}'
  register: os_instance
```

After an instance is created we store the IP in this webserver group. This is dynamic addition and is called in-memory inventory.

```
- add_host:
    name: "{{ item.openstack.public_v4 }}"
    groups: webservers
  loop: '{{ os_instance.results }}'
  when: item.openstack is defined
```

When the server is up and running we are now ready to deploy our application, we achieve this by adding:

```
- hosts: webservers
  vars:
    ansible_python_interpreter:  /usr/bin/python3.6
  vars_files:
    - host_vars/mrc.yaml
  gather_facts: true
  roles:
     - role: configure-server
```

```
    - role: deploy-webapp
    - role: deploy-harvester
```

For all the IPs in the ‘webservers’ group, the roles configure-server, deploy-webapp and deploy-harvester are called.

Configure Server

First, whenever we create a new instance we configure the server by adding proxy settings and updating the server to the latest packages.

```
- name: update environment file
  become: true
  blockinfile:
    dest: /etc/environment
    block: |
      http_proxy=http://wwwproxy.unimelb.edu.au:8000/
      https_proxy=http://wwwproxy.unimelb.edu.au:8000/
- name: Create directory for docker proxy settings
  become: true
  shell: |
    mkdir -p /etc/systemd/system/docker.service.d
    touch /etc/systemd/system/docker.service.d/http-proxy.conf
- name: Add Proxy settings to Docker
  become: true
  blockinfile:
    dest: /etc/systemd/system/docker.service.d/http-proxy.conf
    block: |
      [Service]
      Environment="http_proxy=http://wwwproxy.unimelb.edu.au:8000/"
```

After setting Proxy settings we restart and reload the docker service to ensure settings have been applied.

```
- name: Docker Daemon Reload
  become: true
  shell: |
    systemctl daemon-reload
    systemctl restart docker
```

After we set all proxy settings we update and upgrade the instance.

```
- name: "apt-get update"
  become: true
```

```
  apt:
    upgrade: yes
    update_cache: yes
    cache_valid_time: 86400
```

Deploy Web-App

To deploy our web application first we clone our github project.

```
- name: clone a private repository
  become: true
  git:
    repo:
"https://{{gituser}}:{{gitpass}}@gitlab.com/AbhishekYadav142110/comp90024-2.git"
    dest: '{{web_path}}'
    accept_hostkey: yes
    force: yes
    version: master
```

Since we are cloning a private repository we cannot pass password directly, therefore we used ansible vault to save our password. Adding secret keys to Ansible vault can be found in the below documentation:

https://docs.ansible.com/ansible/latest/user_guide/vault.html

After cloning a GitHub repo we check our docker daemon is running and deploy our webapp in a docker container.

```
- name: Ensure docker daemon is running
  service:
    name: docker
    state: started
  become: true
- name: Build Docker Image
  become: true
  shell: 'docker build -t {{image_name}} {{web_path}}'
- name: Remove container if Present and start New the Container
  become: true
  shell: |
    docker stop {{container_name}} || true && docker rm {{container_name}} || true
    docker run -d --name {{container_name}} -p {{container_port}}:{{container_port}}
{{image_name}}
```

Finally we setup our nginx server as given below:

```
- name: "install nginx"
  become: true
  apt:
    name: ['nginx']
    state: latest
- name: delete default nginx site
  become: true
  file:
    path: /etc/nginx/sites-enabled/default
    state: absent
- name: copy nginx conf
  become: true
  shell: 'cp {{web_path}}/nginx-default /etc/nginx/sites-enabled/default'
```

After Nginx is installed we restart our application and then our application is ready to use.

```
- name: restart Services
  become: true
  shell: |
    systemctl restart nginx
    docker restart {{container_name}}
    service nginx reload
```

Deploy Harvester

To deploy Harvester first we clone our project repository and the twint repository

```
# Cloning Repository
- name: clone a Project repository
  become: true
  git:
    repo:
"https://{{gituser}}:{{gitpass}}@gitlab.com/AbhishekYadav142110/comp90024-2.git"
    dest: '{{web_path}}'
    accept_hostkey: yes
    force: yes
    version: master
- name: clone a Twint repository
  become: true
  git:
    repo: "https://github.com/sidsawant25/group65-twint.git"
    dest: '{{twint_path}}'
    accept_hostkey: yes
    force: yes
```

Next, we install all the dependencies required for the harvester.

```
# Installing Dependencies
- name: Install pip3
  become: true
  apt:
    name: ['python3-pip']
    state: latest
    update_cache: yes
- name: Install Dependencies
  become: true
  shell: 'pip3 install {{twint_path}} -r {{twint_path}}/requirements.txt'
```

Then in the last step we start our harvester using the ‘nohup’ command.

```
# Stop Existing and Start New Harvester
- name: Kill harvester if Already Running
  become: true
  shell: "pkill -f 'python3 {{web_path}}/harvester/harvestmultithreading.py' || true"
  ignore_errors: true
- name: Start Harvester
  become: true
  shell: 'nohup python3 {{web_path}}/harvester/harvestmultithreading.py
&>{{web_path}}/harvester/nohup.out &'
```

With Ansible, the deployment becomes very easy. The creation of instances and the setup was done very easily and quickly.

Result and Analysis

Correlating with economy size

SInce the API returns a single sentiment score for every LGA which represents the general sentiment of the population in that LGA and we also have a single value for economy size for every LGA from AURIN, we have simply plotted sentiment score against economy size on a scatter plot without any preprocessing unlike the rest of the cases described below.

Correlating with age wise population distribution

There are 5 age-groups we have considered:

- 15-30 years
- 30-40 years
- 40-50 years
- 50-60 years
- 60+ years

Given a particular politician, for a particular group in a particular LGA of a particular state/territory the sentiment score is calculated as follows:

1. Multiply the population count in this age group for this LGA with the average sentiment score of this LGA. This gives the sum of the sentiment scores for the population of this age group for this LGA. Also, keep the population count of this age group in this LGA.
2. Repeat step 1 for every LGA and keep adding the resulting values: total sentiment score and total count.
3. After all the LGAs are considered, divide the total sentiment score by the total count. This gives an approximation of the general sentiment score for this age group for the selected state/territory.

Since the tweets scraped do not give the age of the tweeter, approximating the sentiment score by summing up the proportionate sentiment scores according to an age group is the best estimate that we could use.

Correlating with population diversity

We have considered the country of origin of an individual here. 36 countries are taken into account. The approximate sentiment score of the population having a common country of origin is calculated using the same process described for age-wise population distribution.

Inferences

Scott Morrison - Prime Minister

Scott Morrison seems to have a moderate sentiment across Australia tending towards low and even negative in Victoria. He has higher popularity in Western Australia, Northern Territory and Queensland.

- It seems that despite having a relatively higher popularity across these states, the economic size distribution plot shows extreme trends with scores either being really high or really low.

- If the age graph is taken into consideration, few states like Queensland, Western Australia, or Tasmania show a gradual increase in the sentiment score with increase in age, states like Victoria and Northern Territory show a reverse trend.
- Considering the states with cities having major population, like Queensland, NSW, Victoria, SA, and WA, it can be observed that for most of these states, the population diversity graph gives a low sentiment score and is moderate in some states like New South Wales and South Australia.

Daniel Andrews: Premier, Victoria

Daniel Andrews seems to be a well-liked personality in Victoria. The overall sentiment about him is quite positive across the whole state.

- If we consider the graph for economy size, it can be seen there is mixed sentiment in the areas with lower economic size, however the areas with higher economy have a lower sentiment.
- The sentiment score across all age groups is similar ranging between 0.108 to 0.113 which indicates a happy population.
- Daniel Andrews is also popular among the diverse population of Victoria. Comparing the sentiments of all the states, it can be seen Mr. Andrews is well liked by the people he serves.

Gladys Berejiklian: Premier, New South Wales

Considering the sentiment score of New South Wales regarding their premier Gladys Berejiklian, it is difficult to recognize a pattern as the sentiment score across the LGAs is quite mixed, though overall indicating a positive sentiment.

- If we consider the economic distribution, there is again a mixed sentiment, hinting towards low sentiment score, even negative in some cases.
- The age wise distribution bar shows a generalized graph with scores between 0.031 to 0.033 for all age groups which is relatively low.
- The diverse population of NSW again has a mixed sentiment regarding their premier with scores going as low as 0.01 for few countries.

Most of the States have similar sentiments regarding Gladys Berejiklian.

Andrew Barr: Chief Minister, Australian Capital Territory

Not much can be said about the Chief Minister of ACT, due to lack of data available from Canberra. But it is interesting to note that he is one of the politicians with the highest sentiment score across all states, even more than the Prime Minister of Australia. His highest score being in South Australia, it can be observed that he has quite a positive sentiment across all parameters of age, economy, and diversity.

Annastacia Palaszczuk: Premier, Queensland

According to the twitter users of Queensland, there seems to be room for improvement within the government of politician Annastacia Palaszczuk. The sentiment score obtained is one of the most negatively distributed sentiment across states with respect to their premiers.

- Considering the economy size, the sentiment score is mixed tending towards neutral, but also negative in a few cases.
- The age Distribution does not show much variance between different age groups. For Western Australia, this graph displays quite negative sentiments with increase in age group.
- An interesting trend can be noted in the population diversity graph. Considering all the states which have the major Australian cities, the sentiment score for Annastacia Palaszczuk is quite high.

Steve Marshall: Premier, South Australia

Steve Marshall has a moderate sentiment score across Australia. The sentiment score based on the tweets for Steve is consistent across all the three parameters which we have considered.

- Steve is a premier of South Australia and his average sentiment score is 0.1379 which is the lowest for him as compared to the other regional areas.
- His average sentiment score in Western Australia is quite impressive, especially among the middle class people.

Mark McGowan: Premier, Western Australia

We cannot observe any specific trends for Mark McGowan. He has an average sentiment score of 0.14 across Australia.

- Although Mark McGowan has a moderate positive sentiment score, the score for the Srilankan community was negative in Western Australia against him.

Michael Gunner: Chief Minister, Northern Territory

He has a moderate sentiment score across Australia. He has the highest sentiment score in New South Wales, especially from big cities.

- In the northern districts, there is not much variance between the average sentiment scores of different age groups.
- Huge variation can be seen in the sentiment scores from people from different countries, but the maximum scores are positively distributed with a very few negative averages. It can be said that he has a chance of improvement against the negative scores of people from the United States of America in northern territory.

Peter Gutwein: Premier, Tasmania

He is the premier of Tasmania but has a comparatively high sentiment score in other states of Australia.

- He has a high sentiment score in almost all age groups across Tasmania. He has a high sentiment score in areas of moderate to high population density in Tasmania. There are no such other trends spotted in the data regarding him.
- There is not much variation in scores of people from different countries, but there are some negative sentiment scores from the population of people from certain countries with a very tiny population in Tasmania.

Contribution

	Frontend	Backend	Docker	Harvester	Ansible	CouchDB	AURIN	Testing	Documentation
Abhishek Ugrasen Yadav	✔	✔	✔				✔	✔	✔
Siddhesh Sawant			✔	✔	✔	✔			✔
Vanshree Bapat				✔			✔	✔	✔
Manish Kumar Shaw			✔				✔	✔	✔
Pranav Barve					✔	✔			✔