COMP90024 ASSIGNMENT2

Melbourne City Analytics On Cloud

Web Application Design Report

THE UNIVERSITY OF MELBOURNE

Team 16

Cheng Qian, 962539

Qingmeng Xu, 969413

Tingqian Wang, 1043988

Zhong Liao, 1056020

Zongcheng Du, 1096319

Melbourne City Ananlytics On Cloud

Cheng Qian, Qingmeng Xu, Tingqian Wang, Zhong Liao, Zongcheng Du

University of Melbourne

Abstract

To carry out the city analytics on cloud, we choose a region of the Great Melbourne (under SA2 level) to study on. In this project, we design a software system architecture and developed a web application to illustrate three perspectives of citizens' lives based on the cloud computing on tweets over these regions and compared them to data of four main city's features.

Contents

1 Background

To carry out the city analytics on cloud, we choose the Great Melbourne region as our research region. In this project, we design a software system architecture and developed a web application to illustrate different perspectives of people's life based on the cloud computing on tweets in this region.

1.1 Scenarios Selection

More and more people like to post their stories, attitudes, interests, preferences, and opinions in the social media, and also obtain valuable information from it. When we see from the whole city, it is also meaningful to know people's interest and attitude arriving in the social media. In this project, we focus on three perspectives of citizens' lives over different suburbs. The three topics we choose from people's daily life are:

(1) alcohol

(2) coffee

(3) sports

We aim to get a distribution of related tweets counts under these three topics over the Great Melbourne region, and the sentiment propensity of each related tweets. Also, we compare the distribution of tweets counts with the data , collected from AURIN, of four main city's features which are:

(1) population

(2) proportion of poverty household

(3) proportion of residents with great life satisfaction

(4) median household income of the suburb

1.2 Tasks Allocation

To construct the complete systems of our web applications together, we allocate each member to responsible for each part of our project. And collaborative efforts and effective communication have greatly push us to complete the project.

- Back-end and Configuration: Cheng Qian, Qingmeng Xu
- Data Harvesting and Analysis: Zhong Liao, Tingqian Wang
- Front-end and Web Visualization: Zongcheng Du

1.3 Relevant Link

GitHub repository: https://github.com/ttcici/COMP90024-Assignment2.git
YouTube video of web-based frontend: https://youtu.be/wYs7xOKohko
YouTube video of Ansible deployment: https://youtu.be/gV7zMx3pJAo

2 System Design and Architecture

2.1 System Environment and Resources

All components of the system in this project, such as Front-end, Back-end, CouchDB, and Data Harvester are deployed on the UniMelb Research Cloud (URC) of Melbourne University. In URC, at most four nodes (or instances) are allocated to each team, along with total 250 GB of storage, therefore, we allocate 10 GB to instance-1, where we deploy Front-end and Back-end, and 80 GB to each of other three instances, where CouchDB cluster is deployed. Ansible is utilized in this project to make the allocation mentioned above through OpenStack, and set up the environments for those components on the cloud, while docker is used to deploy them on each node of the cloud.

2.2 System Architecture

The major architecture of the system is shown in the Figure 1 below.

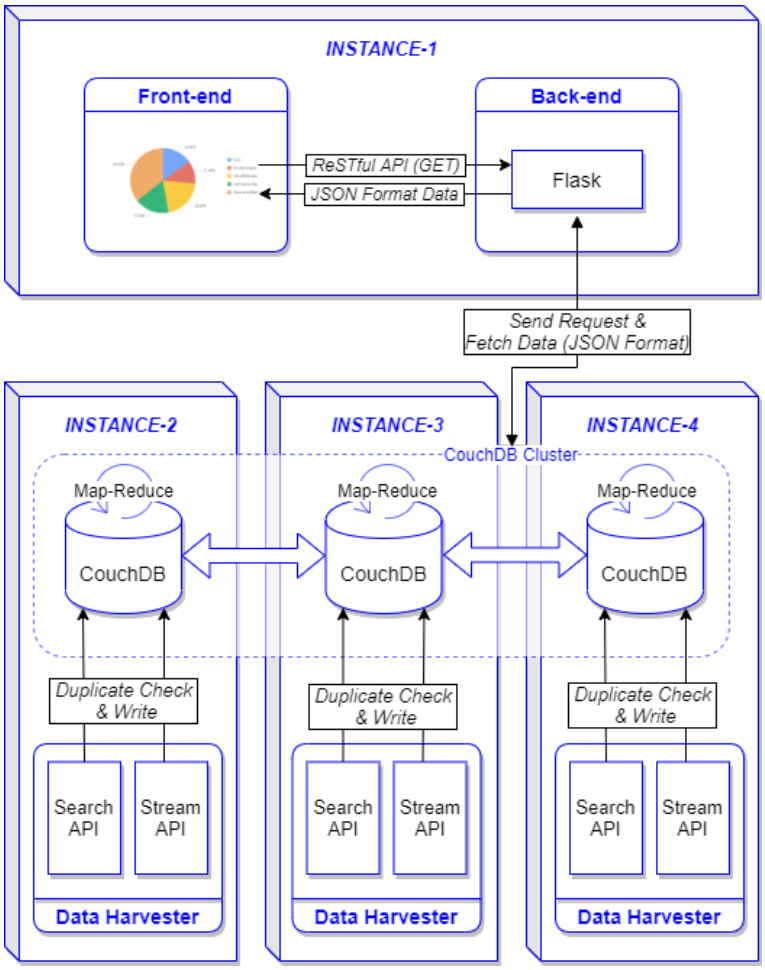


Figure 1: System Architecture

Instance 1 contains both Back-end and Front-end, and the Front-end process can communicate with the Back-end by sending the ReSTful-style GET request to the Flask server.After receiving and interpreting the request, the Flask server will send request to the CouchDB cluster to obtain the data required, and return to the Font-end.

In this system, the CouchDB is distributed into three nodes, namely instance-2, 3, and 4. The CouchDB server in each instance is connected to each other and together form a cluster. In order to ensure the horizontal scalability of a distributed system, each CouchDB server will store the same data as the others', performing as a replica, so that when one or two nodes are accidentally down, the data will not be lost. Using such architecture strategy, the reliability and availability of the database is strengthened.

When receiving requests from the Back-end, the CouchDB server will do the map-reduce operation using the predefined View stored in the server, and then returns the JSON format data.

Similar to the CouchDB server, the data harvester is running in each of the instance-2, 3, and 4. The Data crawled by the process, whether through Search API or Stream API, will be checked if duplicate before written into the corresponding CouchDB server in the same instance.

3 Configuration

3.1 Ansible Playbook

NeCTAR cloud only provides the Infrastructure as a Service, the creation and deployment of the instances needed to be done with respect to the project's requirements. In this assignment, we apply Ansible to automatically create the instances and set up the environments.

Ansible is an automatic operation and deployment tool which can free IT engineering from repeated and tedious tasks and save previous time. In our project, the maximum number of instances is limited to four, thus it might fail to show the advantages of Ansible which can configure and manage resources in large scale. Nevertheless, it shows the possibility of using Ansible to deploy on hundreds and thousands of servers at once.

Our project utilizes a three-step deployment to configure the services. Correspondingly, we use 3 Ansible playbooks (Figure 2): the first playbook creates four instances on the NeCTAR cloud and install the dependencies and resources that are required by all instances. This step consists of tasks like setting the security group and volumes. Those tasks are similar in all instances so that we could use one playbook to deploy that at once. The next is an ad hoc playbook whose functionality is to clone the team's GitHub repository. We treat it as a separate one because it is updated frequently during the development. When all instances are created, we get four corresponding IP addresses. The third playbook assigns different tasks to each instance. In our case, instance-1 maintains the web server and deploys the backend code while instance-2, instance-3, and instance-4 run the CouchDB and the data harvester.

- Installation playbook: create and customise the instances including the volumes, security group and flavor etc.
- Github-repository playbook: clone the team's GitHub repository.
- Configuration playbook: deploy services such as the CouchDB and web server to the aiming instance.

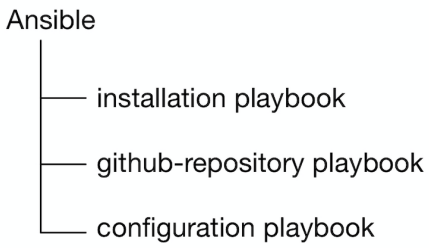


Figure 2: Ansible Playbook Structure

The benefits of multiple Ansible playbooks are: 1) dividing different tasks into separate playbooks give a clear view of functionalities of each playbook; 2) each playbook can be run independently as needed without interfering with other parts. As a result, we do not need to re-create the instances and re-install the dependencies once we have done the installation tasks. Besides, by isolating the frequently updated part such as cloning repository as a specific playbook, we avoid re-running other tasks that do not need to be updated and save lots of precious time.

3.2 Docker Container

Docker is currently the leading containerization technology which uses the recourse isolation features of the Linux kernel to allow independent containers to run within a single Linux instance. Meanwhile it is supported by the MacOS and Windows as well.

The docker container is kind of isolated from the outside environment and provides an identical internal environment. Therefore, the outer software and hardware configurations do not disturb the inner environment, and the applications running inside do not affect the outside as well.

Both `Dockerfile` and `docker-compose.yaml` are utilized in this project to build and run the container for different components. The Front-end and Back-end are directly deployed using `docker build` and `docker run` command with the corresponding `Dockerfile`, they are shown below in the Figure 3.

```
FROM node:latest

ENV http_proxy http://wwwproxy.unimelb.edu.au:8000
ENV https_proxy http://wwwproxy.unimelb.edu.au:8000

RUN mkdir /front-end
WORKDIR /front-end
ADD . /front-end

RUN npm install
RUN npm install express
EXPOSE 8080
CMD [ "node","server.js" ]
```

(a) Front-end Version

```
FROM python:3.7.3

ENV http_proxy http://wwwproxy.unimelb.edu.au:8000
ENV https_proxy http://wwwproxy.unimelb.edu.au:8000
ENV PYTHONUNBUFFERED 1

RUN mkdir /back-end
WORKDIR /back-end
ADD . /back-end

RUN pip install -r requirements.txt
EXPOSE 8000
CMD [ "nohup", "python3","app.py" ,"&" ]
```

(b) Back-end Version

Figure 3: Dockerfile

In the `Dockerfile`, the image that the container will be based on is specified, for instance, the Back-end is both built from `python` while the Front-end is based on `node` as shown in the first line of the Figure 3(a) and 3(b) respectively. Moreover, the environment parameters referred to as `http_proxy` and `https_proxy` are set for both Back-end and Front-end, so that these containers are able to be connected to the external network. The command behind the word `RUN` uses `pip` to install the prerequisite python libraries for the container, which will be executed during the building period. The `nohup` and `&` command behind `CMD` in the Back-end version specify that the back-end and front-end process will continue running when the user (in this case, us) logging out after setting up the container.

As for the `docker-compose.yaml`, it is used to set up the CouchDB nodes in our system, which is introduced in the next section.

3.3 CouchDB Set Up

There are two choices for CouchDB deployment. We could either build all nodes in one instance or create one node on one instance and then set up the cluster. In this project, we prefer the latter structure which is more stable compared with the former one. If one instance

fails, it will not affect other nodes and the application could still works as normal. Placing nodes on different places also helps mitigate the burden on each instance. Besides, each node is running inside a docker container to avoid being disturbed by external environment.

```yaml
version: "3"
services:

  couch_master:
    image: "couchdb:2.3.0"
    restart: always
    network_mode: host
    ports:
      - "5984:5984"
      - "5986:5986"
      - "4369:4369"
      - "9100-9200:9100-9200"
    volumes:
      - /home/ubuntu/couchdb/master/data:/opt/couchdb/data
    environment:
      - COUCHDB_USER=user
      - COUCHDB_PASSWORD=pass
      - NODENAME=172.26.133.141
    container_name: couch_node
```

Figure 4: Docker-compose File

To set up the CouchDB application, we first write a docker-compose file for each node as shown in the Figure 4. In this file, we specify the image used which is "couchdb:2.3.0". Then we set the "restart" as always, map the ports of container to the ports of the host and set the username and password etc. It is worth noting that we map the directory */home-/ubuntu/couchdb/master/data* on the instance to the directory */opt/couchdb/data* inside the container. Instead of copying the directory to the container, we mount it as a shared volume. The benefit is that the data stored in the folder will not vanish if the CouchDB fails. Changes to the directory */home/ubuntu/couchdb/master/data* will be synchronised inside the container as well. We use Ansible to automatically set up the CouchDB node on each instance.

Once the creation is done, we utilize two requests to add nodes to cluster:

```
curl -X POST -H "Content-Type: application/json"
http://<username>:<password>@<master_node_ip>:5984/_cluster_setup -d
'"action": "enable_cluster", "bind_address":"0.0.0.0", "username":
"<username>", "password":"<password>", "port": 5984, "node_count": "3",
"remote_node": "<remote_node_ip>", "remote_current_user":
"<remote_node_username>", "remote_current_password":
"<remote_node_password>" '
```

```
curl -X POST -H "Content-Type: application/json"
http://<username>:<password>@<master_node_ip>:5984/_cluster_setup -d
'"action": "add_node","host":"<remote_node_ip>","port": 5984, "username":
"<username>","password":"<password>"'
```

After adding nodes to cluster, we use the following command to complete cluster setup:

```
curl -X POST -H "Content-Type: application/json"
http://<username>:<password>@<master_node_ip>:5984/_cluster_setup -d
'"action": "finish_cluster"'
```

To verify that the cluster is successfully set up, we use the following command to check the membership of the cluster:

```
curl http://<username>:<password>@<master_node_ip>:5984/_membership
```

And we receive the following response:


```
{
        "all_nodes": [
                "couchdb@172.26.133.141",
                "couchdb@172.26.133.82",
                "couchdb@172.26.134.37"
                ],
        "cluster_nodes": [
                "couchdb@172.26.133.141",
                "couchdb@172.26.133.82",
                "couchdb@172.26.134.37"
]}
```


Figure 5: Cluster Setup Response

It means that we have correctly set up the CouchDB cluster.

3.4 User Guide for Testing

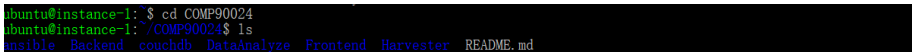
To test the deployment and end user usage of the system, certain verification can be done during each stage of the progresses mentioned above.

For the system environment setting up:

- After successfully running `run-installation.sh` file, which allocates the resources available on URC and creating the instances as well as the `Docker` for further use, the user can use the command `ssh -i Assign2-Annette.pem ubuntu@<instance-ip>` (where `<instance-ip>` is one of the four instances' IP address of our system, namely `172.26.132.129`, `172.26.133.141`, `172.26.133.82`, and `172.26.134.37`) to check whether the user is able to enter the instance located on the specific IP address, if is, as shown in the figure below, then the setting up is correct.

- After running `run-github-repository.sh` file, which clones the necessary codes and files for the system maintained in `GitHub` to each existing instance, the user can enter the instances to check whether the file called `COMP90024` and its contents are correctly copied as shown in the figure below.

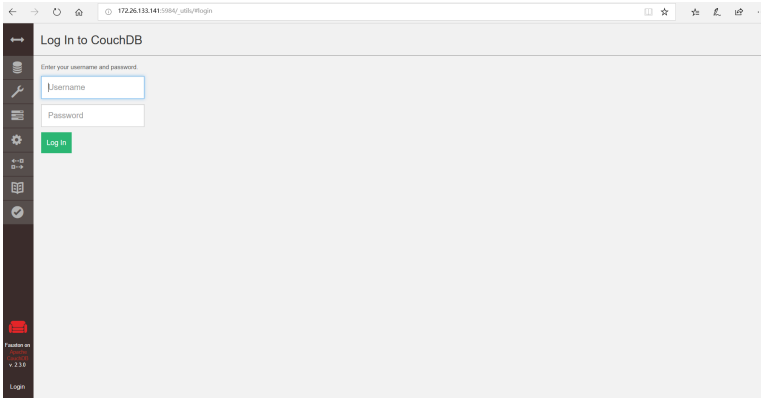
(a)



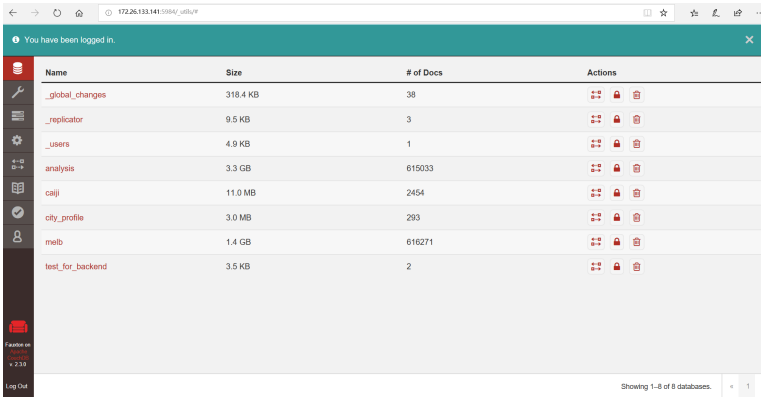
(b)

For the applications such as CouchDB cluster, the Back-end and the Front-end for user usage after running `run-configuration.sh` file:

- To check whether the CouchDB cluster is correctly installed and running, the user can type the IP address along with the port number in the browser to try to enter the CouchDB server's login page, the addresses (with port number) in this system are `172.26.133.141:5984/_utils/login`, `172.26.133.82:5984/_utils/login`, and `172.26.134.37:5984/_utils/login`. If the login page is displayed successfully as shown below, then input `user` as username and `pass` as password to login the CouchDB server. If the CouchDB is clustered correctly, the newly created database in any one of the instances located on the above three addresses will be duplicated to all other instances.

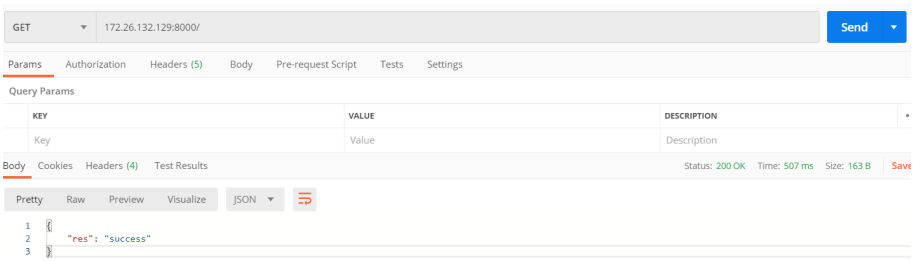


(c) Before Login

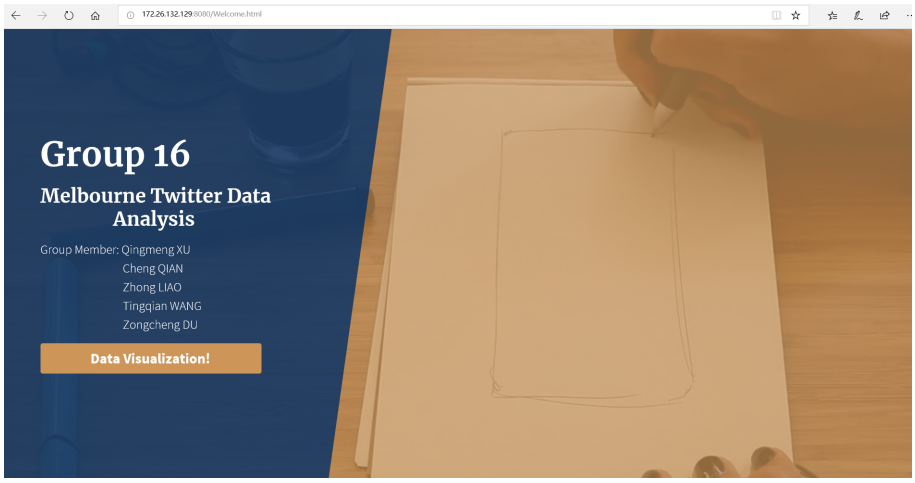


(d) After Login

- As for the Back-end, which is supposed to be able to access through `172.26.132.129:8000`, if set up successfully, using the postman to send a `GET` request to `172.26.132.129:8000` will receive a response message shown below.



- The Front-end is also deployed on instance-1 (`172.26.132.129`) as the Back-end, and can be visited through port `8080`. If it is running appropriately, typing `172.26.132.129:8080/Welcome.html` in the browser will lead to the Welcome page of our data visulisation system, as shown below.



4 Back-end

In this project, two major tasks of back-end are receiving requests from front-end, and sending back responses with data obtained from database. More specifically, the back-end process will be continuously listening on certain port, and waiting for the ReSTful-style requests, which is sent by front-end and contains the information about what data the front-end wants. After interpreting the requests, the back-end process will try to connect to the database, which is CouchDB in this project, and obtain the required data to send back to front-end through ReSTful design.

4.1 Flask

The framework utilized as back-end in this project is Flask, which is a python-based micro web framework. It is lightweight and easy to deploy, which means it is suitable for small web applications. Therefore, Flask is chosen as the back-end framework in this project, as processing ReSTful API and obtaining data from database do not require huge resources.

4.2 ReSTful Interface

In this project, the back-end server is deployed on instance 1 (IP address: 172.26.132.129) and can be accessed through port 8000, and two main interfaces are available for the front-end:

- For the first one, when receiving the GET request as

  `/database/<string:database_name>`

  from the front-end, where `database_name` is the name of the specific database, the back-end server will set up the connection with CouchDB and send the request to fetch all data in the required database, which will then be returned to front-end.

- The other one is used to obtain the map-reduced data from database through the predefined View in CouchDB, with the format of

  `/view/<string:database_name>/<string:view_id>/`
  `<string:view_name>/<int:group_level>`

  where `database_name` is the name of the specific database, `view_id` is the ID of the predefined View, `view_name` is the name of the specific View to be run, and `group_level` is the level of keys to be grouped.

5 Data Harvester

We target to harvest as many tweets with suburb geographic location information as possible to provide enough data for scenarios data analysis. The following flow chart shows the process of harvesting. All the works are executed by Python files named "*search.py*", "*stream.py*" and "*melb.py*" on the instance 2 (IP address: 172.26.133.141).

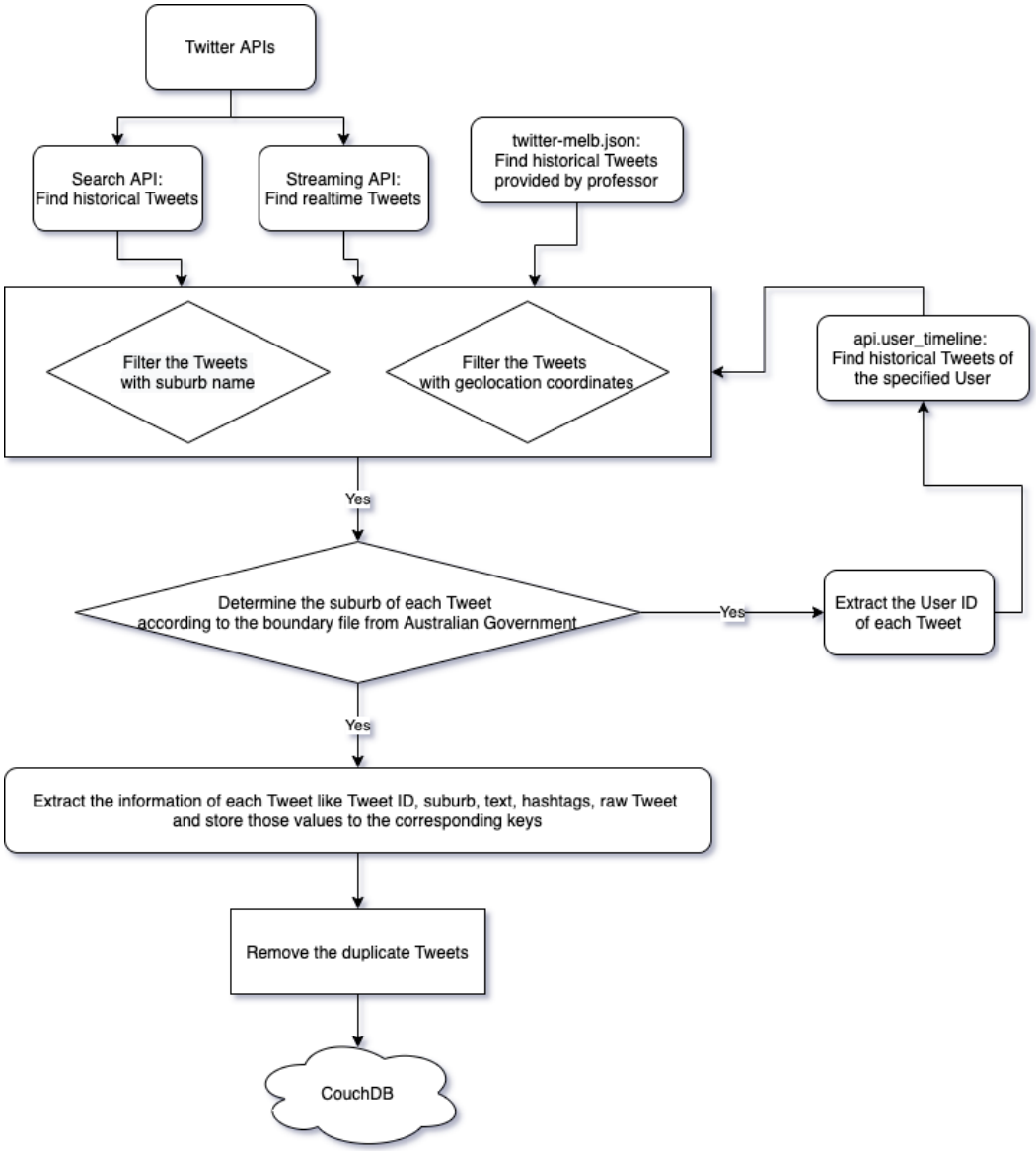


Figure 6: Harvester Flow

5.1 Search API and Streaming API

To access Twitter's API, we need to create and verify the authentication by running the following code:

```
import tweepy

# Authenticate to Twitter
auth = tweepy.OAuthHandler("CONSUMER_KEY", "CONSUMER_SECRET")
auth.set_access_token("ACCESS_TOKEN","ACCESS_TOKEN_SECRET")


api = tweepy.API(auth)


# test authentication
try:
    api.verify_credentials()
    print("Authentication OK")
except:
    print("Error during authentication")
```

Search API and Streaming API are two main methods to gathering the tweets among all the APIs provided by Twitter Developer Labs. The difference between these two APIs is that search API can only find the tweets that are posted several days ago while streaming API returns the real-time tweets. In order to get the full text for analysis, the parameter *tweet_mode* should be set as "*extended*" for search API. For streaming API, we can find the full text from the value of the key "*extended_tweet*" in tweet data. When the search API hit the seven days limit, the file "*search.py*" will be terminated, while the file "*stream.py*" will constantly harvest the real-time tweets.

5.2 Tweets Processing

Firstly, we need to determine the geographical scope of the target place for tweets harvesting. The geographical scope declared differently in the two APIs:

- For search API, the scope should be declared by a centered coordinate and a radius to circle the target place.
- For stream API, we use a box with four vertex coordinates to cover the target place.

Also, we need to set the language of the tweets as English before running the python script of search API or streaming API.

In order to facilitate storing data in CouchDB, we will change the harvested twitter to json format named *tweetJson*. And then the followings are two ways to filter the geographical location information about suburb of the tweet:

(1) We need to check the value in the key "*place_type*" of *tweetJson*["*place*"]. If the place type is neighbourhood, it implies that the key "*name*" of *tweetJson*["*place*"] is the suburb name. In this case, if the suburb name is contained in the boundary file from Australian Government, the tweet will be stored in the database.

```
if tweetJson["place"]:
    if tweetJson["place"]["place_type"] == "neighborhood":
        place = tweetJson["place"]["name"]
        for suburb in boundary["features"]:
            if suburb["properties"]["vic_loca_2"] == place.upper():
```

(2) The boundary file from Australian Government contains a set of polygon boundary coordinates for each suburb. So if the key value of "*coordinates*" is not null in *tweetJson* and the coordinate belongs to one of the suburb in the boundary file, then the tweet will be stored in the database.

```
if tweetJson["coordinates"]:
    longitude = tweetJson["coordinates"]["coordinates"][0]
    latitude = tweetJson["coordinates"]["coordinates"][1]
    coordinate = Point(longitude, latitude)
    for suburb in boundary["features"]:
        for multipolygon in suburb["geometry"]["coordinates"]:
            if Polygon(multipolygon[0]).contains(coordinate):
```

5.3 Challenging

We have faced three major challenges during the process of harvesting.

5.3.1 Duplicate Tweets

When we first stored the harvested data in the database of CouchDB, we found some duplicate data exist. These duplicate tweets possibly occur in the following situations:

(1) Using **api.search** or **api.user_timeline** to harvest the historical tweets.

(2) Harvesting the historical tweets from the "*twitter − melb.json*" file.

(3) For a tweet, both the name of the suburb and the coordinate together match the geographic location information in the boundary file from Australian Government.

To avoid the occurrence of duplicate tweets, instead of using a random ID allocated by the CouchDB, we decided to take "*tweet.id*" which is unique for each tweet as the key value, "*_id*", of the harvested data. So that each tweet data will be stored with an exactly unique key ID. The newer the tweet, the larger the tweet ID. In this case, a ResourceConflict exception will be raised when a tweet with same ID that has already stored in the database. To catch this error, we put the statement **db.save()** into the **try** block. Once the error is caught, the **except** block will be executed to prevent duplicate storage. In this way, the duplicate tweet will be ignored without interrupting the save process.

```
# Check if the tweet already exists in the CouchDB to prevent duplicate storage
except couchdb.http.ResourceConflict:
    print("Duplicate tweets found and ignored.")
```

5.3.2 Twitter API Rate Limit

There is a rate limit for twitter API, which limits the number of calls to 180 every 15 minutes and the number of tweets from each call to 100 as maximum. That is to say, we can only process 18000 tweets/15 mins. To deal with this issue, we set a **try except** block to catch the error triggering by rate limit. We have 5 API accounts which were created by each of our group member so that when one of the API reach the rate limit, it will switch to the next API automatically. Also, we have set a clock to record the usage time for each API. If the API has reached the rate limit within 15 minutes, the statement **time.sleep()** will let the python script sleep for a while until the rate limit is over.

```
# If rate limit is hit, switch to the next API
except tweepy.error.TweepError as e:
        # Error Messages
        print(e)
        print("#" + str(i + 1) + " API hits rate limit (or other error), switching to next API.")
```

5.3.3 Low harvesting efficiency for search API and streaming API

When we only use the search API, due to the seven days limit, we can only get less than 2500 tweets with suburb geographic location information, equivalent to a rate of 300 tweets per day. The harvesting efficiency of streaming API is similar to search API. However, we need a great number of tweets to support us to analyze our scenarios. So we need to improve our efficiency for tweets harvesting. And we try to fix the limitation by following ways:

(1) Gathering the data with geographic location information that meet our expectation from the "$tweet - melb.json$" file. Nearly 0.4 million tweets are stored in the database.

(2) Using **api.user_timeline** to find the historical tweet of the specified user, which is unlimited. Because we consider that the user who post a tweet in one of the suburb of Great Melbourne may have posted his/her other tweets before in the same suburb or somewhere nearby.

These two solutions did help us largely speed up the rate of data harvesting, especially the second solution above. After we adjust the second solution into a part of streaming API, we can store about 10 thousand tweets per day, which means that the rate of harvesting is 32 times faster than before.

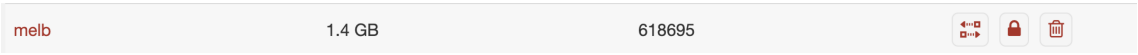


Figure 7: The "melb" Database

As for June 26, we obtained nearly 620,000 Twitter data with geographic location information and store them in the database named **"melb"**, among which about 220,000 tweets were harvested by search API and streaming API. The huge improvement of efficiency will make the results of our analysis more accurate.

6 Data Processing

Data from CouchDB as a harvesting end will be read and process by the following analysis flow:

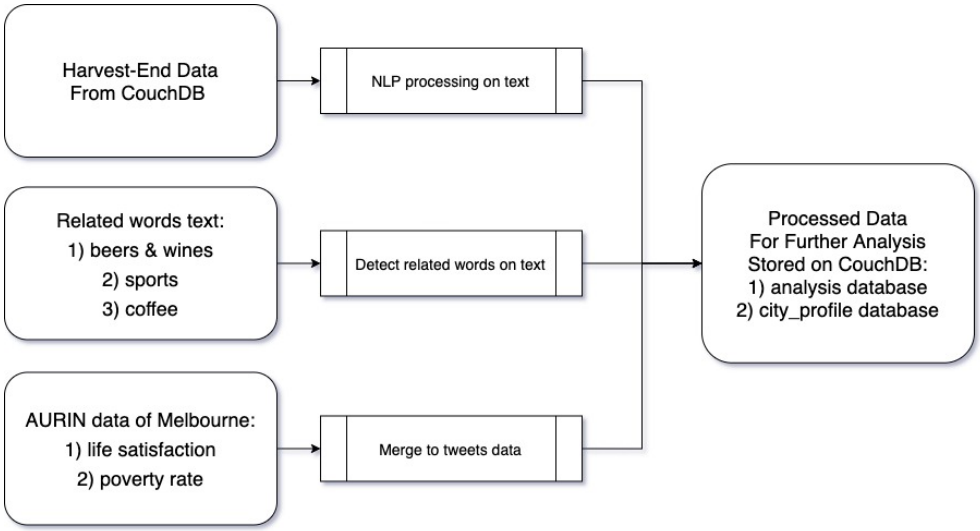


Figure 8: Data Processing Flow

6.1 NLP Processing

Being initialized into a list object with 'id', 'suburb' and 'text' in Python, tweets are processed in order by lower-cased, tokenized, lemmatized, and sentiment analysis. Tokenize and Lemmatized are built in NLTK package, and the SentimentIntensityAnalyzer is based on the VADER lexicon that is also built in NLTK package.

6.1.1 Lemmatize Method Defined

After downloading 'wordnet' from nltk package, lemmatisation is applied on each word in a text with two steps:

- Starting from steaming a word as a verb. If the word was lemmatized, then return the lemmatized word as a result.
- If the result remains unchanged, steaming the word as a noun.

For example, 'drunks' was lemmatized as a verb, resulting in a word still as 'drunks'. Then pass the condition and to be lemmatized as a noun, return a word as 'drunk'. Though 'drunk' is past participle of 'drink', it was not steamed to 'drink' based on the algorithm of Porter's algorithm [**Porter**, **1980**], however, it can be steamed to 'drunk' as a noun.

```
lemmatizer = nltk.stem.WordNetLemmatizer()

#lemmatisation of the words from text
def lemmatisation(word):
    lemma = lemmatizer.lemmatize(word,'v')
    if lemma == word:
        lemma = lemmatizer.lemmatize(word,'n')
    return lemma
```

6.1.2 Sentiment Analysis

Sentiment Analysis is employed on each text. Text, being analysis based on VADER (Valence Aware Dictionary for sentiment Reasoning), will have scores for each propensity of 'compound', 'neutral', 'positive', and 'negative'. We choose the propensity with the highest scores as the sentiment label of the text.

```
#SentimentIntensityAnalyzer
nltk.download('vader_lexicon')
def IdentifySentiment( sentence ):
    sia = SentimentIntensityAnalyzer()
    ps = sia.polarity_scores( sentence )
    sentiment = max(ps, key = ps.get)
    return sentiment
```

6.2 Related Words Detection

We choose three topics of people's city life in this project. There will be three text files as three sets of collection of related words under each topic. Take 'coffee' as an example, it has related words all in lower case, including:

- Seed words: coffee, cafe, etc.
- Coffee types: flat white, cold brew, etc.
- Coffee bean: robusta, arabica, etc.
- Coffee shop in Melbourne: market lane coffee, industry bean, etc.

Each of the related word is appended into specific list object after lower-cased and lemmatized. The related word lists are ready to compared with tweets texts, so as to detect the related words in the texts if there are one or more. If the coffee related words exist in a tweet text, we will update a pair of key and value as $\{"cf\_exist" : 1\}$ to that tweet data. If the coffee related words don't exist, the value will be 0. This rule is also employed under alcohol and sports related words detection.

6.3 AURIN Data

We obtained the Great Melbourne region data of poverty rate, median household income and life satisfaction from AURIN Data Portal. And process the data to be ready to used as following:

(1) For the 'poverty rate' data, we choose the poverty rate that represents the proportion of people who have equivalised disposable household income after housing costs that is below half median equivalised disposable household income after housing costs. For the people's who can afford beers and wines, coffee, and sports events or goods should largely at least have no huge difficulty in dwelling. We merge this data to processed tweets data by suburb name.

(2) For 'median household income' data, we also merge it to the processed tweets data by suburb name.

(3) For 'life satisfaction' data, the number of people were counted and recorded under each of the satisfaction score levels which are ranging from 10 to 100. We choose the satisfaction score from 80 to 100 to represents great satisfaction, and sum the counts of people under these targeted fields to represent the number of people with grate satisfaction of life. Besides, we also merge the population of each suburb together with satisfaction data to the tweets data by suburb name.

6.4 MapReduce on Processed Data

After the processing, we will obtain a data with a format as following:

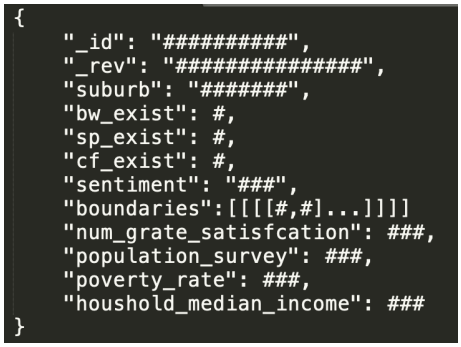


Figure 9: Processed Data Format

Based on this processed data stored in CouchDB, we can apply the MapReduce Analysis tool on the database for each data.

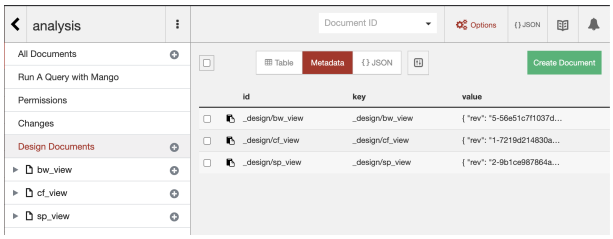


Figure 10: Design Documents

Besides, we can get the results of the tweets counts for each topic by a comment of

curl -X GET
http://172.26.132.129:8000/view/analysis/<View_ID>/<index_name>/<reduce_group_level>

Take getting the counts result of coffee related tweets as an example, executing the comment will return a result like this:

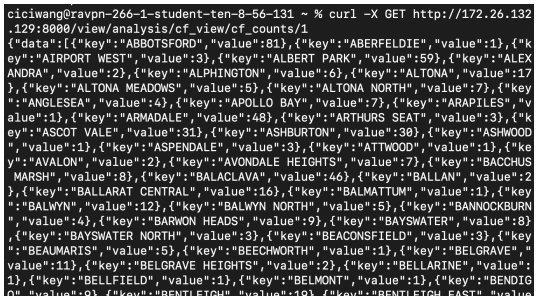


Figure 11: GET the MapReduce Results

6.5 Challenging and Discussion

Initially, our processing program has following shortages:

(1) Run locally, which also face a challenge of network instability. Since our data has a size of more than 1GB, if the local network gets disconnected, our program will stop and need to be restart by hand. This is not efficient and also occupy too much of the local running memory.

(1) Program stop at the moment when no more new tweet is updated into the database. In this case, the program can't restart automatically when more new tweets data reach the database.

Realizing these two problems, we consider to put the processing script on the instance and start to run by a comment on the instance, and put the core of data processing part in a while loop which end with a sleep of 86400 seconds. The program succeed in running in instance 4 (IP: 172.26.134.37), and can also run after the sleep of one day to process the updated tweets data from harvesting.

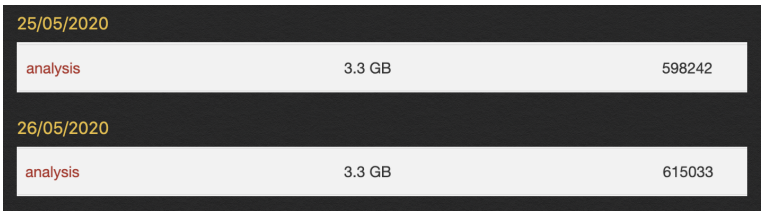
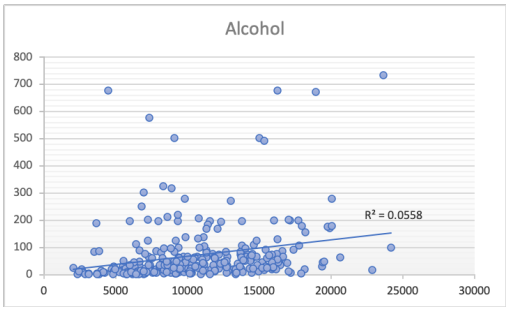


Figure 12: After One Day Sleep of the Processing

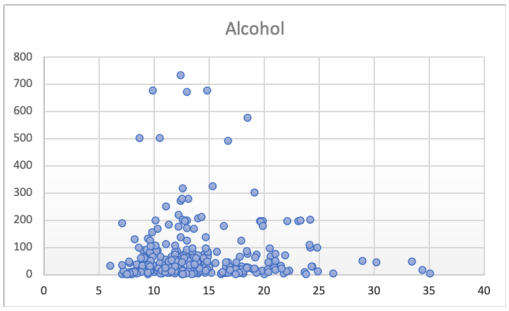
7 Scenario Analysis

We have explored an analysis on the data. One of the results shows that the number of tweets in exactly Melbourne region is always the highest all over the suburbs. For example, the alcohol related tweets counts are over 4500 in exactly Melbourne region while that of other regions are all under 1000. So we exclude the tweets counts in Melbourne and include other suburbs in our data exploratory analysis in this section.

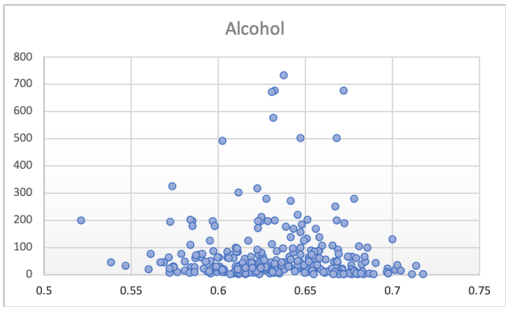
Tweets Related to Alcohol Patterns like number of tweets related to alcohol against population show no significant correlation between this two items. Against the 'low-income families proportion', the suburbs with high number of tweets are those have low proportion of low-income families. Against the ' resident satisfaction', the suburbs with grate residents satisfaction are clustered in the proportion between 60% to 65%, but only few of them seem to have large number of tweets related alcohol. Against the 'average income', the suburb' average income seems to be evenly distributed between $1200 to $2400, and the suburbs with number of tweets related to alcohol seems to centering in the middle of average income.



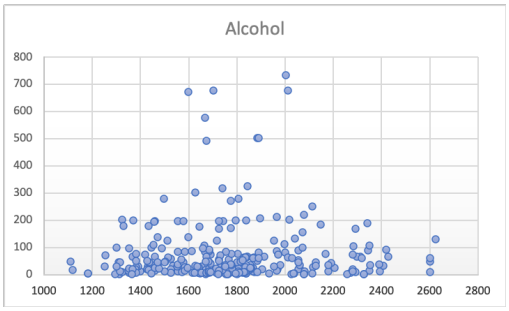
(a) against Population



(b) against Low-income Families Proportion



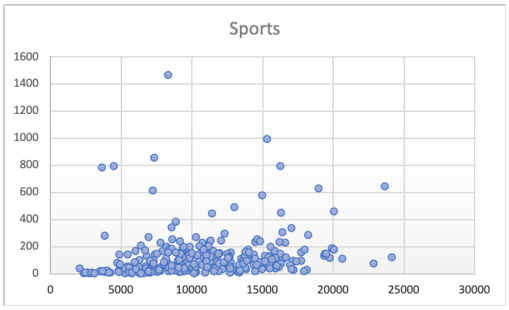
(c) against Residents Satisfaction



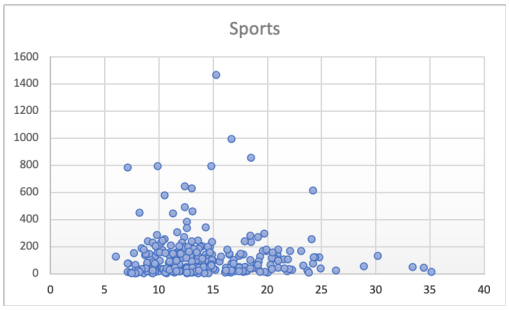
(d) against Median Income

Figure 13: Number of Tweets Related to Alcohol

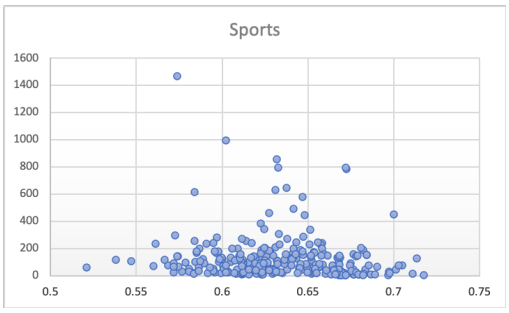
Tweets Related to Sports The sports related tweets number against the low-income families proportion seems to have similar pattern with alcohol related tweets counts have.



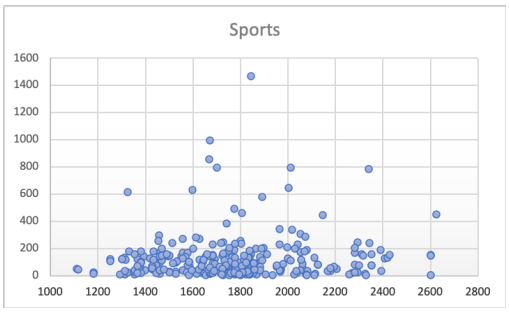
(a) against Population



(b) against Low-income Families Proportion



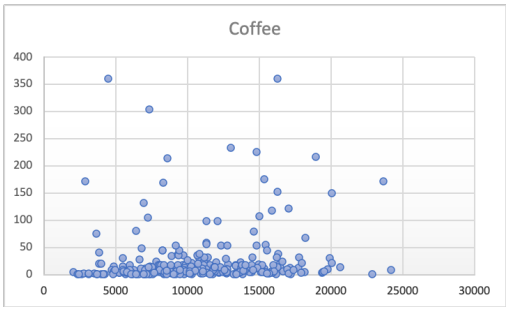
(c) against Residents Satisfaction



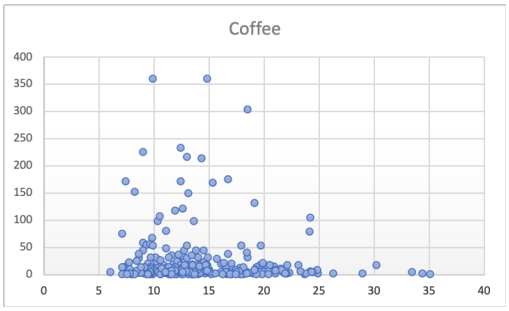
(d) against Median Income

Figure 14: Number of Tweets Related to Sports

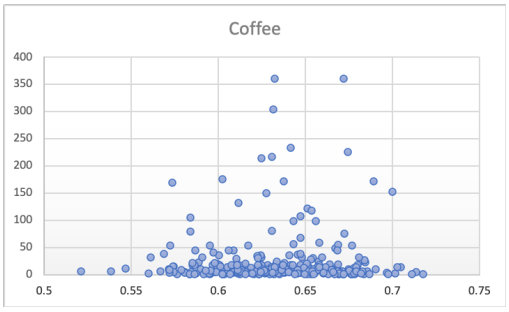
Tweets Related to coffee The coffee related tweets number against the low-income families proportion seems to have similar pattern with alcohol and sports related tweets counts have. Besides, seen from the plot (c), the suburbs with high number of coffee related tweets seems to be a little bit clustered to those with grate proportion of grate residents satisfaction.



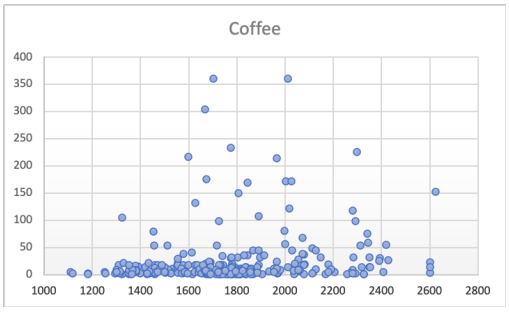
(a) against Population



(b) against Low-income Families Proportion



(c) against Residents Satisfaction



(d) against Median Income

Figure 15: Number of Tweets Related to Coffee

8 Web Visualization

8.1 Front-end Design

Web visualization is the process of merging data from different sources and displaying them visually. Our page is mainly composed of two parts: welcome page and analysis page. The welcome page shows the composition of our team members. There is a "data visualization" button that takes the user to the analysis page. The original intention of designing this page is to make webpage access more standardized and the user experience more friendly. The analysis page contains a huge map of Great Melbourne. The GeoJson data of Victoria Local Government Area map using to determine the boundaries is retrieved from Australia government (2019).

8.2 Visualization

On the analysis page, we generated three heat maps based on alcohol Twitter statistics, sports Twitter statistics and coffee Twitter statistics. The user can intuitively feel the difference between the areas according to the depth of the colour. When the user places the cursor on the map, he will obtain various data about this area, including the name of the area, statistical information, median income, population, residents' satisfaction, and the proportion of low-income families. The last four of these data come from AURIN (Australian Urban Research Infrastructure Network), which provides accurate and reliable data about cities in Australia. The following figure shows a demo using coffee information:

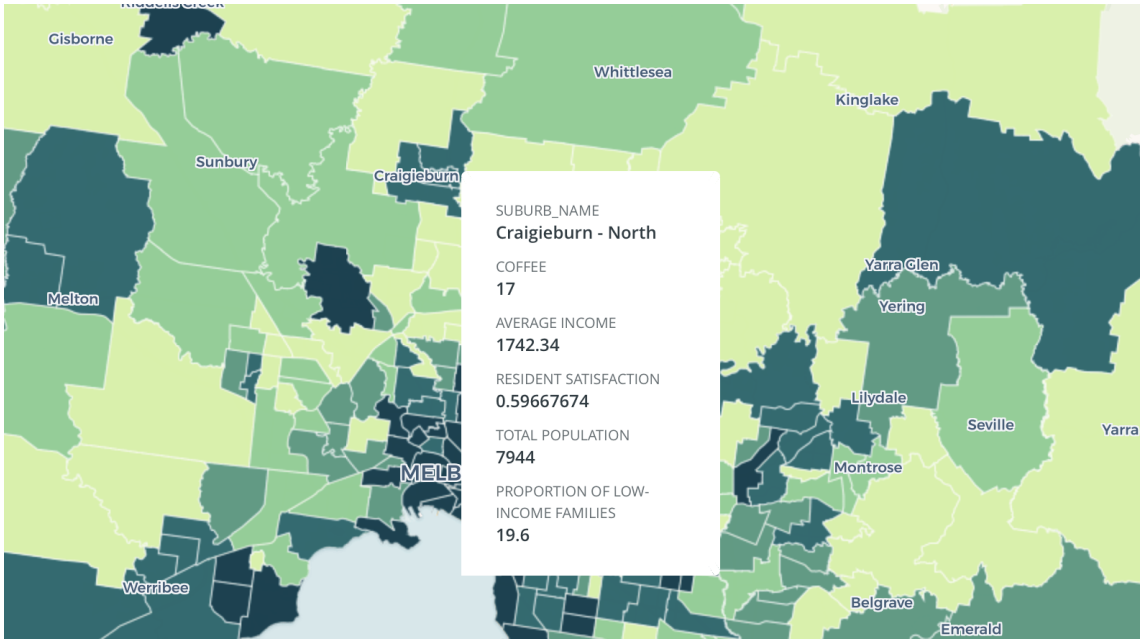
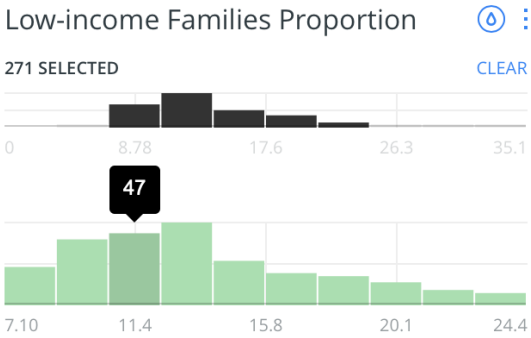
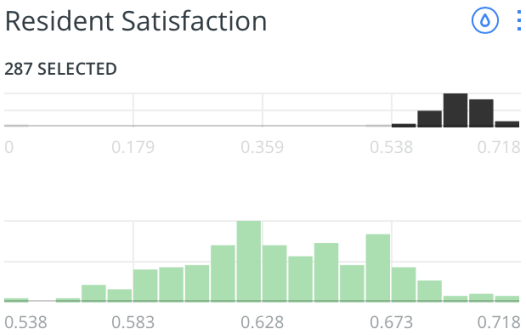


Figure 16: Analysis Page

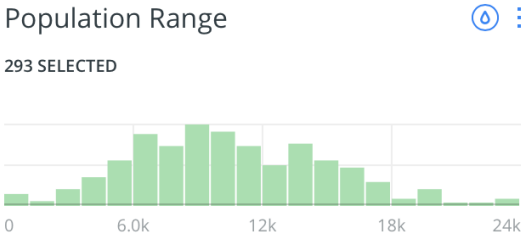
Since we convert AURIN data into a histogram on the page for observation, users can easily feel the relationship between alcohol Twitter data and various AURIN information:



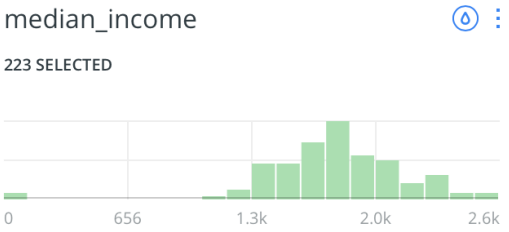
(a) Low-income Families Proportion



(b) Resident Satisfaction



(c) Population Range



(d) Median Income

Figure 17: AURIN Histograms

In addition, we provide a function that allows users to set filters. For example, the user can view the suburbs where the resident satisfaction is between 61 % and 65 %:

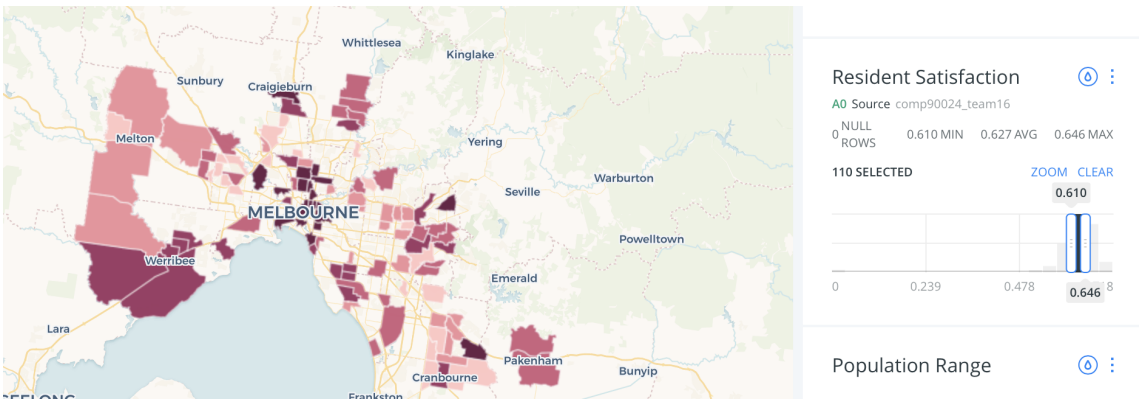


Figure 18: Customer Filter

When doing data visualization, we found that Twitter's statistics are obviously abnormal in certain places, such as airport, racecourse, CBD, etc. The official demographics of the airport area are 0, but Twitter statistics are higher than in most areas. According to the common sense of life, this is not difficult to understand, because the active users in these areas are mainly floating population. When analyzing these areas, we need to pay special attention.

9 Reflection

In the era of Big Data, analytic tools have been largely employed in the visibility of social media users. Raised by the demand of high-capacity networks, low-cost computers and storage devices as well as the widespread adoption of hardware virtualization, distributed system and autonomic computing, cloud computing has come into being.

Computing
Deployed on UniMelb Research Cloud (URC), our system can operates multiple tasks and distributed data on 4 instance nodes at the same time with 8 VCPUs and 360GB RAM in total, leading our processing on data analytics much more efficiently and steadily.

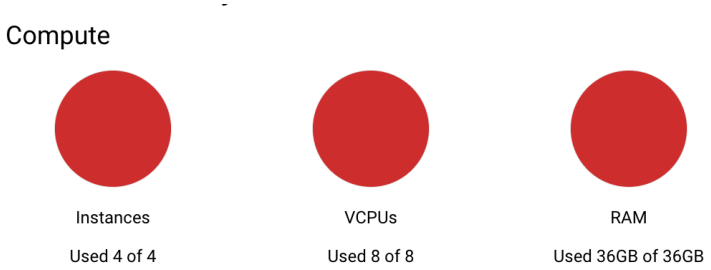


Figure 19: Use of URC Capacity

OpenStack
Based on the OpenStack cloud, it is much more convenient for us to deploy our entire system and source. However, it is also true that most of the operations throughout OpenStack rely on a message broker which is hard and impossible to scale. When a single point of failure in the architecture happens on a message broker, it is already possible to put the whole deployment into tremendous trouble.

Ansible
Apart from the challenges and system design choices we mentioned in each section, there are still something we believe we could do better if having more time.

With the help of Ansible, we tried to apply automation during every stage of our development. However, we could do more and better if we have more time. In our current application, for instance, every time we update our code in GitHub repository, we need to manually run the Ansible playbook (github-repository.yaml) to synchronize change on our instances. If we have more time, we could deploy a CD (Continuous Deployment) system using tools like Jenkins. This system can monitor the status of our repository, every time a new request emerges, it will automatically update the code on the instances. Similarly, we could do a better automation to manage docker containers. When we change the functionalities in container, instead of stopping and re-start the containers by hand, Jenkins can automatically do that for us if we set up the configuration in advance.