Analytics on the Cloud

Hilary McMeckan, 297180, **Trina Dey**, 1223966, **Ezequiel Gallo**, 1028532
Soham Mandal, 1190783, **Ankita Dhar**, 1154197

Abstract

Twitter has evolved over time from being a collection of random thoughts from individual users, into an organised source of information representing individuals, groups and corporations. This report explores the potential application of cloud infrastructure solutions to perform data collection and analysis on tweets to measure the real-time *well-being* of a population. To deploy our cloud infrastructure, we utilised a number of tools including Melbourne Research Cloud (MRC), Docker, Ansible as well as CouchDB for our database infrastructure. If *well-being* can be measured through Twitter data, this could have wide-reaching implications across a number of industries, in particular offering a markedly improved method of accessing, measuring and monitoring real-time datasets. Similarly, by accurately and efficiently monitoring trends on Twitter, Governments and other stakeholders can be better prepared and respond faster to emergent issues. Unfortunately our analysis found no statistically significant relationships between sentiment and other survey markers, indicating that the traditional methods of manual surveys and data collection still remains as the most accurate way to measure *well-being*. This data and analysis was exported and displayed via a front-end UI to allow for better visualisation of the analytical scenarios investigated..

Contents

1 Introduction

Life in Australia has changed immeasurably over the past 18 months as a result of the global pandemic and the subsequent shifts in human behaviour across not only Australia, but the globe. With enforced lockdowns and disruptions to the lives and habits of Australians, there has been an increasing amount of attention from Governments and statisticians on monitoring the *well-being* of the populous.

With the traditional method of capturing and analysing socio-economic and demographic trends occurring only every few years, there is a temporal disconnect between the state of the nation and the reference set of statistics. As the global pandemic has taught us, things can shift quickly, therefore, it is more important than ever to find innovative methods to accurately and efficiently ascertain *well-being* data and monitor trends in the population in real-time. As one of the most widely used social media platforms, with users that are able to post status updates instantaneously, Twitter has increasingly become a useful tool for measuring and analysing population trends [5].

Due to the sheer volume of tweets generated by Twitter users, and the requirement to efficiently capture and analyse these tweets, scalable computer processing power in the form of cloud computing is often required. For this project, our objective was the following:

- **(i)** Deploy a cloud-based solution that exploits virtual machines (VMs) on the Melbourne Research Cloud (MRC) and that can scale dynamically as required.
- **(ii)** Harvest tweets using both the *Streaming* and the *Search API* interfaces from each major city in Australia. Our amalgamated tweets should then be stored in a database ready to be used in our analysis.
- **(iii)** Develop a range of analytical scenarios using both the MapReduce functionality of CouchDB as well as additional natural language processing techniques. To supplement our Twitter data, we obtained relevant datasets from the Australian Urban Research Infrastructure Network (AURIN) [9].
- **(iv)** Design a front-end user interface (UI) that graphically visualises each of our analytical scenarios in a web-based application.

Although our chosen scenarios allowed us to illustrate the functionality our system, our specific motivation was to understand if Twitter data could be used to efficiently measure *well-being* and monitor real-time trends in the population. To investigate this, we hypothesised that the sentiment of tweets could be correlated to demographic and socio-economic characteristics used to measure *well-being* by the Victorian Government and that, potentially, sentiment analysis could be used as a substitute for these manually collected datasets. On the other hand, *topic modelling* allowed us to capture the top trends on Twitter, in order to better understand the essence of life in Australia after such a tumultuous year. Here, we performed two different topic modelling techniques: *Empath* and *Latent Dirichlet Allocation*.

This report details the specifications of our developed application including an overview of our system design (Section 2), an in-depth analysis of our cloud infrastructure (Section 3), database architecture (Section 4), comprehensive overview of the analysis performed (Section 6) and finally, technical details of our front-end interface which is responsible for visualising our results (Section 7). Throughout each section, we have also discussed the advantages of the technology utilised, the strengths and weaknesses of our approach as well as details on key challenges faced throughout our implementation.

Our appendices contain links to our GitHub (Appendix 10.1) and presentations (Appendix 10.2) as well as our step-by-step user guide to deploying our applications (Appendix 10.3).

2 System Architecture & Design

2.1 System Overview

In this section, we provide an overview of the structural, behavioural and functional aspects of our implementation. Figure 1 visually represents our system architecture, specifically illustrating the relationship between the software, hardware, network as well as the structural and functional dependencies between software and hardware. The lower-level designs are discussed in Section 2.2, providing a detailed inventory of hardware used, the applications installed and the interactions between various components over the network. Our system architecture needed to provide a solution to the following:

- **(i)** Harvesting tweets from Twitter
- **(ii)** Storing these tweets
- **(iii)** Perform analysis over these tweets
- **(iv)** Display the analysed data using a cloud-based UI

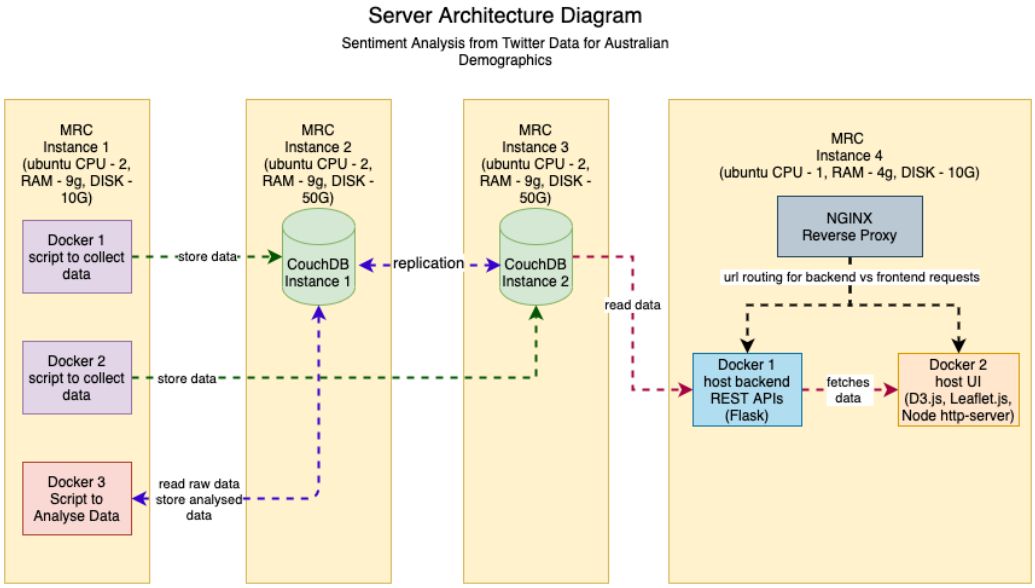


Figure 1: *Overview of server architecture*

2.2 Functionality

2.2.1 Cloud

Cloud infrastructure is the provision of on-demand computing services, such as virtual storage and processing power. Other common use cases include web-hosting. For this project, we were provided with the following hardware configurations: 4 server instances, 8 virtual CPUs (2 per instance), $32GB$ RAM ($8GB$ per instance), $500GB$ of data to be used as required and Available Flavours - 1 core with $4.5GB$ RAM, 2 cores with $9GB$ RAM, 4 cores with $16GB$ RAM. As such, one of our key requirements was to allocate these available resources across each server as efficiently as possible. As such, we adopted the following approach:

- Applications that run continuously like the Twitter harvester and database need a larger number of cores. Therefore, we allocated three instances with 2 cores - $9GB$ RAM each.
- To store large amounts of data, we needed to have an external data volume attached to the server. Therefore, an extra $50GB$ of data was added to the servers running CouchDB.
- The servers with front-end UI and back-end APIs utilised the least amount of resources since the load on these servers is based on traffic/number of hits on the website. Hence, this server was allocated only 1 core - $4.5GB$ RAM.
- Since resources are to be utilised efficiently, each server is further Dockerised to run multiple applications. The back-end and front-end are run on the same server with URL re-routing achieved by reverse proxy NGINX. Notably, the harvester and analysis can be run on the same server in separate Docker containers.

2.2.2 Database

Apache CouchDB is an open source NoSQL document database that collects and stores data in JSON-based document formats. In our design, we chose partitioned CouchDB with replication, deploying CouchDB in *standalone mode* rather than *clustered mode*. Although clustered mode improves the reliability of our database using sharding, internal balancing and replicating the database using internal HTTP calls, the performance of our CouchDB instance would be impacted when running over MapReduce queries over a clustered database. As such, we deployed CouchDB in *standalone mode* with the database created in *partitioned mode* on the same node and replicated over other instances.

2.2.3 Web-server

Flask Flask is a lightweight Web Server Gateway Interface (WSGI) web application framework. It is simple, with the ability to quickly scale its complexity. We used Flask in our design to host back-end APIs which connected with our CouchDB database. This fetched data from our MapReduce queries/different views and provided this data to the front-end server.

Node.js HTTP-server Node.js HTTP-server is a basic, zero-configuration command-line HTTP server. We used Node.js to host our front-end website which was comprised of HTML, Bootstrap for CSS, D3.js and Leaflet.js which enabled our many different kinds of visualisations.

NGINX In our design, we used NGINX as a web server to run over port 80. This removes the dependency on the internal TCP ports and re-routes the front-end vs back-end requests to different Docker containers using regex based URL patterns.

2.3 Scaling

Scaling is required when the load or the traffic increases in our application. To run the same applications on multiple servers, we needed a duplicate of the application. This could be done in two ways:

(i) By creating server instance image and replicating instances using server images. For this method, we would require more servers and depending on the load, we could still potentially over/under utilise the processing power.

(ii) By creating Docker container images and redeploying the containers on different ports.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allows many containers to run simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, removing reliance on what is currently installed on the host.

In our case, we can run multiple applications on the same server in different Docker containers. Most importantly, if we need to scale the harvester, front-end or back-end APIs, we can run the same Docker image on different ports and re-route/re-balance the load using NGINX. In case of the Twitter harvester, we can simply run multiple harvester images with different API keys.

Scaling can be done manually or dynamically based on the load, however, to scale dynamically would require load balancers and metric monitoring to identify when to scale up or down. Currently, load balancers are out of the scope of the project. However, we supported manual scaling of the Docker containers which can be done by changing the Ansible scripts and configurations files to suit any new requirements.

2.4 Security

MRC is a private cloud, based on OpenStack. When we launch our servers / instances they are only accessible from inside a private network, in our case qh2-*uom*-internal. Thus, to access the servers we needed to login to the University of Melbourne Virtual Private Network (VPN). Additionally, all connections to servers are made over secure shell protocol (SSH). SSH allows us to remotely login to the servers over command line.

As such, MRC servers uses *key pairs* - public and private keys generated with RSA Encryption to authenticate users. The public key is saved with the MRC. The private key is provided when the server is being launched and is used everytime we need to login to that server. This ensures that only those users holding the private key can login to the servers.

Lastly, the University of Melbourne makes each request through firewall and proxy servers. Therefore to access the internet or our servers launched on the MRC, we need to add the *proxy parameters* to the *environmental variables*.

2.5 Fault Tolerance

For a system to be *fault tolerant* it must continue operating properly (maintaining functionality) in the event of the failure of one of more of its components. A fault tolerant system should also be able to restart from the point of failure. We will discuss the two main concerns of our systems with respect to fault tolerance and how they were addressed:

(i) Instance Loss If any of the instances crash, we can resume the operation from the start by relaunching the instance using the Ansible script. While this approach was fine for most instances, there were difficulties when re-launching the Twitter Harvester. We sought to relaunch the server to fetch tweets from Twitter. The Harvester would fetch repeated tweets and result in an error while saving to CouchDB due to a prior document with same ID. To correct this, we stored the *last tweet fetched id* for each account in the CouchDB. Thus whenever the harvester system starts up, it would start fetching the tweets from the last saved *tweet ID* of that particular twitter account.

(ii) Data Loss Tweets were fetched from Twitter over a period of weeks and saved in to the database. When indexing the data for the first time, it took over 24 hours. Hence, in the event the database server crashes, a significant amount of resources and time are required to reconstruct the data. To rectify this, firstly, the volume remains intact, so we can recover the back-up data by attaching the same volume to the relaunched instance. If the volume crashes as well, our data is replicated on another CouchDB database. Therefore the data can be reconstructed.

Alternatively, we could have kept a daily snapshot of the data volumes. However this was not considered as two instances crashing together had a lower probability of occurring.

3 Cloud Infrastructure

Cloud computing is the on-demand availability of computer system resources, especially data storage and computing power. Non-local cloud infrastructure can assist with collating and storing data, performing large scale data analytics and the deployment and hosting of web-services. Clouds host the entire computational workload without the user's involvement in the acquisition of hardware and/or maintenance of the physical underlying assets.

3.1 Melbourne University Research Cloud (MRC)

The MRC provides Infrastructure-as-a-service (IaaS); it provides on-demand cloud services, such as computers, networking and storage, to researchers and students at the University of Melbourne. It offers similar functionality as other popular cloud providers, such as Amazon Web Services (AWS), with nearly $20,000$ virtual cores over variable VM sizes. It also offers other resources such as GPGPUs, load balancing, private networking, research software, etc. The MRC has 200 servers, and each server has between 32 and 512 cores.

The MRC is based on the open-source cloud computing platform *OpenStack*. OpenStack has many associated/underpinning services for deploying cloud systems. Specifically, we utilised the following services as part of this project:

- **Nova** for compute/virtualisation
- **Cinder** for block storage
- **Keystone** for security management

While the MRC is accessible using the browser-based dashboard which can be used to deploy instances, add volumes, set-up security groups, firewalls, etc., the *OpenStack-API* enabled us to directly configure and manage our cloud from the command line. This implementation further detailed in Section 3.2.

3.1.1 MRC Functionality

The MRC offers various tools to configure and set-up VMs required to perform computational tasks. The emulation of VMs is managed by software level *hypervisors* that divide and manage a large number of computational cores and resources in each emulated allocation. Below is a brief outline of each of the key functionalities of the MRC:

(i) **Instance** - Template for a VM

(ii) **Image** - A blank instance with an operating system (OS); MRC provides a number of official images e.g. Linux distributions (Ubuntu, Debian, CentOS, etc.), Microsoft Windows.

(iii) **Instance Snapshot** - Replica of existing instance. Snapshots preserve all the properties and configurations of an instance at the point the snapshot was taken. This helps to backup and recover instances as well as the possibility to redeploy an instance configuration again.

(iv) **Flavors** - Prescribes the size of a VM. For example, the `uom.general.2c9g` flavour has 2 virtual-CPUs (vCPU) and $9GB$ of RAM. Each flavor on the MRC has $30GB$ of storage allocated to the root disk.

(v) **Availability Zone** - Points to the physical location of the data centre. For the MRC, there is currently only one availability zone; `melbourne-qh2-uom`.

(vi) **Volume Storage** - Additional storage (in addition to the default root $30GB$ disk). For bigger projects, the MRC offers larger storage blocks that can be mounted to an instance. However, the permissible size depends on the allowance allocated for a project. For our project, we were allocated $500GB$ to be configured as necessary for our implementation.

(vii) **Networking** - Configures networking options for VMs. It is very likely that a VM will require internet access to install software/packages, obtain data, etc. hence, it will require a: network connection, service to authenticate login activities, and firewall to monitor and control traffic. These are further detailed below:

- **Secure Shell (SSH)** - A protocol used for encrypted communications between a local device and the computer. All activity on the MRC requires mandatory authentication.
- **Key-Pairs** - Key-pairs are used to authenticate each request to the MRC via the command line and consist of two keys, a *public* and *private* key. They can be generated manually or via the MRC dashboard. These key-pairs provides cryptographic strength authentication, allows user to maintain a secure channel and providing greater protection of sensitive research.
- **Firewall & Security Groups** - Helps monitor and managed network traffic. This allows us to configure various ports. For example, we declare port 80 for all HTTPS traffic and port 5986 for our incoming Twitter data feed into our CouchDB databases.

3.1.2 Strengths & Weaknesses of MRC

The main advantages of using *IaaS*, in our case the MRC, is being able to access shared computing resources quickly and efficiently. Traditionally, when requiring this type of infrastructure, a research team would need to acquire and setup servers. This equipment would have a three-to-four year asset life-cycle which might be longer than the life-time of the project. Not only is there significant cost associated with the initial procurement and installation costs, there is also the cost of overheads and ongoing maintenance, the potential to require more resources as the project progresses and the effort of trying to dispose of these assets once the project is completed.

This is where *IaaS*, and specifically the MRC, is enormously useful:

(i) Efficiency The MRC allows instances to be created in minutes. A large number of computers and storage can be connected in a virtual network and shared with all project collaborators, allowing for complex applications to be built with very low initial costs. Moreover, users can potentially access more powerful resources than what they would be able to procure individually.

(ii) Asset Management Utilising the MRC greatly reduces the administration burden as users no longer need to be concerned about the physical access to hardware, power requirements including spikes and outages, cooling, maintenance and upgrades. All hardware is centrally managed by the MRC team. As well, the MRC provides a dashboard which provides a user-friendly interface for interacting with MRC resources.

(iii) Flexibility Each instance can be configured as required. This includes specifying what operating systems and programs are installed, if services are exposed to the internet, hosting services such as a websites or databases, etc.

(iv) Scalability Since the MRC environment is dynamic, users can elastically scale required resources to meet demand. For example, if additional resources are required, a user can simply add these additional resources without the need to procure additional hardware or vice versa, can release additional computing power that is no longer needed.

(v) Asset Obsolescence All hardware will eventually be outdated and if the project extends beyond the useful life of the equipment, it will require project teams to overhaul systems and infrastructure to maintain speed and availability. Given MRC provides IaaS, their systems are constantly being updated with newer/faster technological solutions.

Comparison to HPC High-performance computing (HPC) systems, such as SPARTAN at the University of Melbourne, are an alternative for processing large volumes of data. HPC systems work well when the software runs on Linux, is able to be batch processed and does not require a graphical interface. However, given these constraints, HPC is far less flexible when compared to the MRC. In particular, HPC centrally handles system administration and program installations, i.e. these configurations cannot be changed by a user. As well, resources are shared and therefore users must wait in a queue for resources to become available.

Although the MRC has many benefits, it does have its limitations:

(i) OpenStack Given OpenStack is open-source, it is not as advanced as proprietary offerings such as those from AWS or Microsoft Azure. Being open-source, it is designed to operate on hardware without any specific requirements and as such, may not perform as well as other services specifically built for a task. That being said, these other services do come at a cost.

(ii) Access to Resources Given the large demand for resources, the MRC struggled at times to allocate resources for this project. In addition, the MRC only has one availability zone which is physically located in Melbourne. If a user is located overseas, this distance can lead to communication delays.

(iii) Security when Collaborating Each instance on the MRC is secured by a *key-value* pair. While this is not a problem for individual researchers, and is a property of the key authentication feature of OpenStack, in a team-based project, this private key must be shared impacting the security of our MRC instance.

3.2 Ansible

Ansible is an open-source configuration management and application-deployment tool. Basically, Ansible automates the multi-tiered deployment of complex systems as well as systematically handling changes so that the integrity of the system is maintained overtime. Specifically for our project, Ansible allowed for the automatic configuration and deployment of multiple instances without the need to manually create them individually. Providing a brief overview of how Ansible works:

- Ansible operates by combining nodes and releasing batches of small programs, called *Ansible Modules*. These modules contain configuration information for our remote emulated system.
- Ansible securely executes over SSH by default. Ansible does not require any client to be installed on the remote machine and hence is *agentless*. Given this, it is very convenient to deploy Ansible scripts from any system.

- Ansible utilises a playbook structure expressed in a simple human-interpretative language called YAML. It uses an inventory (.ini) file to store all recipients that need configurational or initialisation changes.

3.2.1 Implementation & Deployment

This section details the various files that contained in a standard Ansible folder structure.

(i) Variables This contains all the declared variables that are used across the various playbook sections. This file defines all the configurations for our desired system state. An example of this file is provided below.

```
# Common variables
availability_zone: melbourne-qh2-uom

# Volumes
volumes:
  - vol_name: database-1 <- variable_name: variable_value
    vol_size: 50
```

(ii) Inventory A `.ini` file that stores information about the exact servers/databases that need to be configured. We denote each one with IP addresses, range of IP Addresses or URLs of the site locations.

```
[harvester-instance]
172.26.139.26

[harvester-instance]
ansible_python_interpreter=/usr/bin/python3
ansible_user=ubuntu
ansible_ssh_private_key_file=new_key_sem.pem
ansible_ssh_common_args='-o StrictHostKeyChecking=no'
```

(iii) Roles Roles let you automatically load related vars, files, tasks, handlers, and other Ansible artifacts based on a known file structure.

(iv) Tasks These are specific instructions that perform a task - such as installing Docker or other dependencies such as pip when Python is installed.

(v) Playbook This is the most important file as it orchestrates how each of the different files above are utilised. The playbook structures the progression of all the other sections, where the order of calling tasks is preserved. This section also links the host variables into each of the other files. As the order of the roles matter, we cannot generate instances without first setting-up the commons and volumes.

```
---
- hosts: localhost
  vars:
   ansible_python_interpreter: /usr/bin/python3  vars_files:
    - host_vars/mrc.yaml
  gather_facts: true
  roles:
    - role: mrc-common
    - role: mrc-volumes
    - role: mrc-security-groups
    - role: mrc-instances
```

(vi) Collections Collections are a distribution format for Ansible content that can include playbooks, roles, modules, and plugins. Furthermore these can be accessed, installed from Ansible Galaxy.

3.2.2 Advantages of Using Ansible

Deploying complex cloud systems requires a lot of repetitive actions. Ansible makes this easier by automating the process to deploy solutions on the cloud. Ansible has quite a positive reputation amongst DevOps professionals for its straightforward operations and simple management capabilities. Although Ansible might not seem necessary for small deployments, it can still provide many benefits as it is easy to forget the configuration of an instance and manual processes are prone to errors. Ansible eliminates these issues as it *codifies* the instance configuration and makes the process idempotent. Ansible is also *agentless* which means the deployable instance need not have Ansible installed leading to greater flexibility. Ansible can extracts inventory, group, and variable information from OpenStack, therefore, we can deploy our entire instance set for our project using just Ansible playbooks and the OpenStack API. Furthermore, Ansible follows a known file structure that is easy to understand and can be version controlled. Although we could employ snapshots in CouchDB, these snapshots are monolithic

and provide only a record of the current state, not what has changed. Ansible scripts record the changes made over time via *source control*. Thus steps are repeatable whenever required to launch new instances from any point in time.

3.3 Containerisation

Containers allow developers to bundle an application with everything it requires to run, such as libraries and other dependencies, and ship it as one package. Docker is currently the leader in containerisation technology. Docker is open-source and automates the development, deployment and running of applications inside isolated containers.

3.3.1 Implementation & Deployment

In deploying our applications, we utilised both VMs and containers where our base computation unit was a VM with containers deployed on top of it. Please refer to Figure 1 which illustrates our VMs and Docker architecture. To build and run each Docker image, we used *Docker-Compose*, a tool for defining and running multi-container Docker applications. Docker-Compose gave us the flexibility to define the configurations for our Docker containers. We were able to define Docker container tags, ports, path files, etc. in our `docker-compose.yml` file. We have provided an outline below of this implementation:

(i) Uninstall any old and/or incompatible versions of Docker running on the instance.

```
- name: uninstall old docker
    tags: 'docker'
    become: yes
    apt:
        name: ['docker', 'docker-engine', 'docker.io']
        state: absent
```

(ii) Install Docker and Docker-Compose. To do this, add the GPG keys to the Ubuntu server, set the path for the *Debian* package and then install.

```
- name: Add docker apt repo key
    tags: 'docker'
    become: yes
    apt_key:
        url: http://download.docker.com/linux/ubuntu/gpg
        state: present
    environment: '{{proxy_env}}'

- name: Add docker apt repo key and update cache
    tags: 'docker'
    become: yes
    apt_repository:
        repo: "deb http://download.docker.com/linux/ubuntu/ focal stable"
        mode: '644'
        update_cache: yes
        state: present
    environment: '{{proxy_env}}'

- name: Install docker
    become: yes
    apt:
        name: docker
        state: present
    environment: '{{proxy_env}}'
```

(iii) To deploy a Docker container, all necessary dependencies and installations for the application must be contained within the container. We defined this using a *Dockerfile* which comprised of a base image of the Docker container. The Dockerfile details the steps to install dependencies, creates a directory within the container, copies files and lastly, the performs the command to run the Docker container. For example, if a user wanted to run a Python script, the base container should have Python and pip installed. Installing pip allows other installations was the container has been created.

```
FROM python:3
RUN mkdir harvesterApp
COPY TwitterHarvester.py   harvesterApp
RUN pip install tweepy
RUN pip install couchdb
WORKDIR harvesterApp
CMD ["python3", "TwitterHarvester.py"]
```

```
version: '3.3'

services:
    harvester:
        build:
        context: .
        dockerfile: Dockerfile
```

3.3.2 Advantages of Containerisation

There are many advantages to utilising virtualisation but using VMs alone can lead to a duplication of resources such as server processors, memory and disk space. This also means that there is a limit on the number of VMs that can be supported. Thus, the proliferation of containerisation. These containers share a single OS as well as associated drivers, binaries and libraries, thus removing this duplication of resources. Each container now only holds the application and related binaries.

Containers allow users to horizontally scale individual services easily without the overhead associated with VMs. In addition, containers provide consistency and speed: they provide a consistent environment for an application, starting from the same point each time, and startup time for containers is faster, a user can quickly run a new process on a server since the container image is pre-configured. As well, Docker is layered file system, every change made to the image becomes a new layer and these file layers are cached allowing for changes to be tracked. Although containers have their strengths, users should still consider what is the best choice for their application based on guest OS, size of the task, life-cycle of the application as well as communication and security requirements.

3.4 Issues & Challenges

This section details the primary challenges we faced during deployment of our cloud architecture:

(i) Adding a dynamic host inventory for floating IP

When servers are launched, they are dynamically assigned an *auto-floating-IP*. This means the IP address is not fixed and can change each time an instance is launched. Therefore, this address cannot be hard coded within scripts. To automatically extract this information, we needed to read the output from the command creating the instance before the next command was executed. To do this, we associated the IP address with a variable to be accessed later. Lastly, we needed to use the `add_host` command to create the inventory dynamically.

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

```
- name: setting the os_instance variable
set_fact:
    os_instances_ips: '{{ os_instance.results }}'

- name: add host
    add_host:
        name: '{{ item.openstack.public_v4 }}'
        groups: '{{ item.openstack.name }}'
        ansible_python_interpreter: /usr/bin/python3
        ansible_ssh_user: ubuntu
        ansible_user: ubuntu
        ansible_ssh_private_key_file: new_key_sm.pem
        ansible_ssh_common_args: '-o StrictHostKeyChecking=no'
    loop: '{{os_instances_ips}}'
    when: item.openstack is defined
```

(ii) Attaching volumes at runtime and updating necessary configuration files

The volumes that are created at the runtime needed to be attached to the server. We were able to run the commands directly on the server terminal or through shell command in the Ansible script.

```
ubuntu@frontend-instance:~$ sudo mkfs -t ext4 /dev/vdb
ubuntu@frontend-instance:~$ sudo mkdir /data
ubuntu@frontend-instance:~$ sudo mount /dev/vdb /data
ubuntu@frontend-instance:~$ vi /etc/fstab
/dev/vdb /data auto defaults 0 0
:wq!
```

(iii) Adding proxy variables in the launched instance environment

To access the internet/intranet from any server launched, we needed to add *proxy variables* by editing the `/etc/environment` file and proxy entries. We achieved this by defining the variables in the `vars` file and adding in Ansible scripts.

```
proxy_env:
    HTTP_PROXY: http://wwwproxy.unimelb.edu.au:8000/
    HTTPS_PROXY: http://wwwproxy.unimelb.edu.au:8000/
    http_proxy: http://wwwproxy.unimelb.edu.au:8000/
    https_proxy: http://wwwproxy.unimelb.edu.au:8000/
    no_proxy: localhost,127.0.0.1,localaddress,172.16.0.0/12,
        .melbourne.rc.nectar.org.au,.storage.unimelb.edu.au,.cloud.unimelb.edu.au

- name: setup proxy in etc environment
    become: yes
    lineinfile:
        dest: /etc/environment
        state: present
        line: "{{ item.key }}={{ item.value }}"
        regexp: "^{{ item.key }}="
        insertafter: "^PATH\$"
    with_dict: "{{ proxy_env }}"
```

(iv) Setting-up the proxy variables for Docker containers

Similarly to the above issue, we needed to add proxy variables to access the internet/intranet from within the Docker containers. This was achieved by updating the variables in files:

`/etc/systemd/system/docker.service.d/http-proxy.conf`; and

`~/.docker/config.json`

The daemon-service was then reloaded and the Docker service restarted.

```
- name: add docker proxy settings
    become: yes
    copy:
        src: http-proxy.conf
        dest: /etc/systemd/system/docker.service.d
        owner: root
        group: root
    when: docker_proxy_file.stat.exists == false

- name: add docker container proxy settings
    become: yes
    copy:
        src: config.json
        dest: ~/.docker
        owner: root
        group: root
    when: docker_config_json.stat.exists == false

- name: reload daemon
    become: yes
    shell:
        cmd: systemctl daemon-reload
    environment: '{{proxy_env}}'

- name: restart docker
    become: yes
    shell:
        cmd: systemctl restart docker
    environment: '{{proxy_env}}'
```

(v) Adding IP address to javascript files

It was a challenge to add the IP address in the javascript files for our backend API URLs. To manage this, we defined a separate javascript file called ip.js and added a single variable `ip_addr` which we override using our Ansible scripts. All our URLs were then formatted to be `http://" + ip_addr + "/api/dashboard/{urlpath}`.

```
- name: updating a js file for frontend with ipaddress created
    become: yes
    lineinfile:
        path: roles/frontend-docker-install/tasks/frontend-ui/code/static/js/ip.js
        state: present
        line: "ip_addr=\"{{ item.openstack.public_v4 }}\""
        regexp: "^ip_addr="
    loop: '{{os_instances_ips}}'
    delegate_to: localhost
```

4 CouchDB

CouchDB is an open source NoSQL database deployment tool that stores data in JSON-based document formats. It is a storage engine purpose-built for cloud and multi-database infrastructures. CouchDB was best suited for this project given:

(i) Efficient Storage Unlike relational databases, CouchDB stores a collection of documents in JSON format. These JSON formats contain various fields and attachments for easy storage. This format is flexible as there are no size limitations and since CouchDB is a NoSQL database, the data structures stored are highly customisible and flexibile.

(ii) Scalability The design of CouchDB makes it extremely adaptable when partitioning databases and scaling data onto multiple nodes. Data can be easily partitioned and replicated while still balancing read and writes.

(iii) No read locks In most relational databases, the row of data being changed is locked until the modification has been processed or rolled-back. CouchDB uses multi-version concurrency control (MVCC) to manage concurrent access to databases. It does so by versioning and appending documents in real-time so that read requests always see the most recent database snapshot. This removes bottlenecks as databases can be accessed without restricting users.

(iv) Accessibility & Reliability A CouchDB database can be accessed from anywhere using its RESTful API and through which all CRUD (create, read, update, delete) operations are accessible. CouchDB is extremely flexible and

can be installed and run on various operating systems given it is written in Erlang, a general-purpose, concurrent, garbage-collected programming language and runtime system. This also makes CouchDB a very durable and reliable storage engine.

4.1 Database Structure

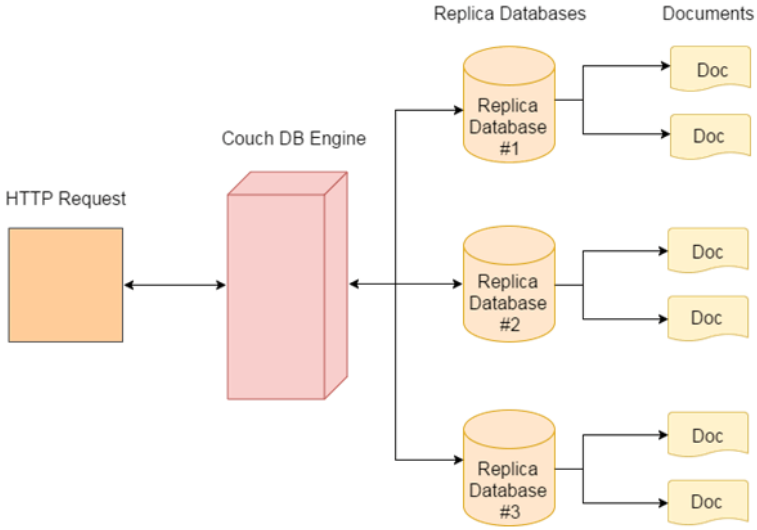


Figure 2: *CouchDB Architecture* [1]

The architecture of CouchDB is described below:

(i) CouchDB Engine - Core of the system which stores internal data, documents and views. This engine is based on B-trees where data is accessed by keys or key ranges which map directly to the underlying B-tree operations.

(ii) HTTP Request - Used to create indices and extract data from documents. It is written in *JavaScript* (JS) and allows for the creations of views using MapReduce queries.

(iii) Document - Stores data

(iv) Replica Database - Used for replicating data to a local or remote database and synchronising design documents

4.2 Implementation & Deployment

4.2.1 Replication

A key feature of CouchDB is bi-directional replication which enables synchronisation of data across multiple servers. This replication enables users to maximise system availability, reducing data recovery times by simplifying backup processes. This is a key feature of our database to improve fault tolerance and mitigate the risk of data loss.

4.2.2 Views

CouchDB uses *views* as the primary tool for running queries over the stored document files. These views allow a user to filter documents, creating powerful indices that help data to be quickly located. These indices also establish relationships between documents. Furthermore, CouchDB views are built dynamically and do not directly affect underlying document stores, meaning there is no limitation to how many different views can be stored. Lastly, these views are created inside of special design documents allowing them to be replicated across multiple database instances.

4.3 MapReduce

Another key feature of CouchDB is the availability of Apache *MapReduce*. This allows us to perform filtering, sorting and reduce/summary operations over our databases. For example, from our database containing harvested tweets, we were able to count and return the total tweets per state. MapReduce does this by marshalling distributed servers and managing communications and data transfers between them to collate this information into a single view. Of particular importance, MapReduce is able to parallelise queries leading to more efficient processing time especially for large datasets. Figure 3 illustrates the MapReduce query diagrammatically. The following MapReduce queries were executed as part of this project:

[1] https://www.javatpoint.com/couchdb-tutorial

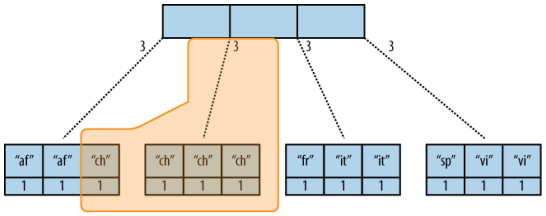


Figure 3: *CouchDB Map Reduce View* [2]

(i) Count by Lat/Lng Maps all the latitudes/longitudes and place names per tweet and then reduces the function to give count against each unique latitude, longitude and place name. This helps plot the map cluster showing where tweets are originating from different parts of Australia.

(ii) Count by State Maps all the latitudes and longitudes to a state and then reduces the function to give count against each state.

(iii) Count by Day Maps all the latitudes/longitudes to a state and day-of-week and then reduces the function to give count against each unique state and day-of-week combination.

(iv) Count by Hour Maps all the latitudes/longitudes to a state and hour-of-day and then reduces the function to give count against each unique state and hour-of-day combination. This is the example MapReduce query provided below.

(v) Count by Lat/Lng Reduced Maps all the latitudes/longitudes and place name with state and then reduces the function to give count against each unique latitude/longitude and place name. We then filter the reduced count by state. This helps plot the state-wise map clusters showing where tweets originated from across different parts of Australia.

```
function (doc) {
  state = 'None'
  if(doc.coordinates_lat && doc.coordinates_lng){
    if(-39.060782 <doc.coordinates_lat &&  doc.coordinates_lat  <-34.135311 && 140.923152 <
    ↪  doc.coordinates_lng && doc.coordinates_lng < 149.909968 )
      state = 'VIC';
    if(-34.470930 <doc.coordinates_lat &&  doc.coordinates_lat  <-28.16951 && 140.906307 <
    ↪  doc.coordinates_lng && doc.coordinates_lng < 154.362534 )
      state = 'NSW';
    if(-28.948032 <doc.coordinates_lat &&  doc.coordinates_lat  <-12.610675 && 137.971640 <
    ↪  doc.coordinates_lng && doc.coordinates_lng < 153.571355 )
      state = 'QLD';
    if(-25.928942 <doc.coordinates_lat &&  doc.coordinates_lat  <-11.409829 && 129.071842 <
    ↪  doc.coordinates_lng && doc.coordinates_lng < 138.080128 )
      state = 'NT';
    if(-35.199090 <doc.coordinates_lat &&  doc.coordinates_lat  <-13.277255 && 113.023758 <
    ↪  doc.coordinates_lng && doc.coordinates_lng < 128.887790 )
      state = 'WA';
    if(-38.033372 <doc.coordinates_lat &&  doc.coordinates_lat  < -25.450329 && 128.859295 <
    ↪  doc.coordinates_lng && doc.coordinates_lng <141.075410 )
      state = 'SA';
    if(-43.354260 <doc.coordinates_lat &&  doc.coordinates_lat  < -40.675674 && 144.678355 <
    ↪  doc.coordinates_lng && doc.coordinates_lng < 148.238422  )
      state = 'TAS';
    if(doc.created_at && state != 'None') {
      emit([state,doc.created_at.substring(0,3)], 1);
    }
  }
}
```

[2] https://docs.couchdb.org/en/latest/ddocs/views/intro.html

From the above example, we can see that we first filtered by state, where the state for each tweet was determined by the latitude and longitude. The `emit()` function was then used to index based on state and hour-of-day from the *created_at* feature. Finally, the "reduce" parameter detailed the type of operation to be performed. The final output of this view gave us the number of tweets per state per hour-of-day.

4.4 Issues & Challenges

There were a number of challenges when working with CouchDB:

(i) One of the major issues that we faced during our development was the long wait times during data indexing for each MapReduce view. Given our database had approximately 4.5 million entries, indexing took almost 4 hours to complete.

(ii) With every change to our view, multiple indexed files would be created, causing *out-of-memory* issues at one point crashing our database instance. To rectify this, we had to mount the appropriate volume size, move all `.couchDB` files, change the data directory in CouchDB settings file `default.ini` and restart the CouchServer.

(iii) We also has issues with re-reducing queries. We could not find any proper documentation and as such, we were not able to use partitioning to it's full extent.

5 Data Collection

As mentioned previously, the objective of our analysis was to investigate scenarios that allowed us to better understand everyday life in Australia. To do this, we harvested Twitter data and mined datasets from AURIN [9] related to our analytic scenarios. This section details the various datasets utilised in our analysis as well as how these datasets were collected/accessed.

5.1 Tweet Harvesting

In the digital age, social media is one of the most abundant sources of data, especially when trying to gauge the current sentiment of a population. The Twitter API is a tool used by researchers to interact with and retrieve Twitter data in order to analyse conversations occurring on Twitter. In this project, we fetched Twitter data from two main sources:

(i) Live Twitter feeds through the Twitter API. To access the Twitter API, we used the open-source Python package *tweepy*.

(ii) Twitter Corpus - A large collection of Twitter data collected by the University of Melbourne.

5.1.1 Collecting tweets through Twitter API

Although we could have filtered real-time tweets according to keywords, we chose to process feeds without any search text in order to get a wide breadth of tweets over which to perform sentiment analysis and topic modelling. However, to obtain non-duplicated datasets across each major city in Australia, we needed to add a geo-location filter, in terms of latitude and longitude, to each Twitter harvester as well as our fabricated variable, *last fetched id* which has been discussed previously. This boundary box extended across each city, providing a radius from which to harvest a selection of tweets over the last 14 days. To access the Twitter API, each team member was required to generate user and project access tokens/keys. In addition, due to the limitation of our sentiment analyser, we only harvested English language tweets.

To combat the rate limits imposed by the Twitter API, we ensured our harvesters did not exceed the request rate per time interval [10]. Given these limits, we also implemented our Twitter harvester as early as possible, allowing us to collect over 2 million tweets during this period. We only read and extracted certain fields from each tweet. There are many fields, we highlight some of the most important fields below:

- ***tweetid*** - ID of each tweet as provided by Twitter
- ***createdat*** - date/time tweet created
- ***text*** - tweet text
- ***source*** - source of the tweet, e.g. browser, mobile App
- ***lang*** - language of tweet (in our case, set to English)
- ***quoted*** - flag indicating a quoted a tweet
- ***retweeted*** - whether the tweet is retweet
- ***replied*** - whether this tweet is a reply to earlier tweet
- ***coordinates*** - {longitude, latitude} where the tweet originated
- ***place*** - location of the user's account. This can provide the location where the user is originally from.
- ***hashtags*** - hashtags used in tweet
- ***usermentions*** - all the users mentioned in the tweet

5.1.2 Twitter Corpus

Given we wanted to average sentiment by geographical area, we needed to source tweets containing latitudes and longitudes. Although this data was prevalent when Twitter was first introduced since location geo-tagging was on by default, more recently, tweets tend to not contain this information. As such, from the Twitter data we harvested, only 44 tweets contained latitude and longitude coordinates. Therefore, in order to perform our analysis, we needed to access additional data where these coordinates were provided. Here, we accessed this additional data through the *Twitter Corpus* and *bigTwitter* dataset.

5.2 AURIN Data Mining

AURIN is an online *workbench* that provides access to thousands of multi-disciplinary datasets, from hundreds of different data sources [9]. Specifically, AURIN houses data on demographics, social indicators, economic activity/productivity, health and livability as well as many other topics. Datasets can be searched by many parameters including keywords, year and *aggregation level* (i.e. the geographical statistical areas as defined by the Australian Bureau of Statistics (ABS)). Our aim was to find the best datasets that would leading indicators of *sentiment*. All the following datasets were accessed through the AURIN Portal [3].

5.2.1 *VicHealth* Indicators Survey

This survey was conducted by *VicHealth* in 2011 and aimed to gain a comprehensive picture of health and well-being in Victoria [11]. These datasets contained the aggregated responses by LGA to a number of different survey questions. Since *well-being* is a very subjective measure of an individual's quality of life, we utilised a number of different metrics to try and accurately measure *well-being*:

Lacking Time Percentage of individuals who always/often lack time to spend with family/friends.

Inadequate Sleep Percentage of individuals who report sleeping less than 7 hours on average on a typical weekday.

Time Pressure Percentage of individuals who report always or often feeling rushed or pressed for time.

Adequate Work-life Balance Percentage of employed individuals who disagreed/strongly disagreed that work and family life often interfere with each other.

It should be noted that these datasets were only available by LGA. Unfortunately, LGAs are *non-ABS structures*; they are defined by the State and Territory governments and released annually. For the purpose of this analysis, this was not an issue but if we wished to make comparison across years, this would prove difficult given the boundaries of LGAs can be redefined if significant changes to the areas have occurred and comparisons year-on-year may lead to *false equivalencies*.

5.2.2 Estimates of Personal Income 2010-2015

This dataset presents aggregates values of *personal income* over the financial years 2010-11 to 2014-15 by LGA [3]. Specifically, we used *median personal income* by LGA for 2011. There is evidence that subjective well-being (i.e. sentiment) improves with increased income, although this does vary with age and level of education. We utilised this dataset to test if this correlation exists with our calculated sentiment.

5.2.3 *Department of Health & Human Services* LGA Profiles 2011

This dataset contains profiles of each LGA based on a number of demographic and socio-economic characteristics for 2011 [4]. Specifically, we used the *unemployment rate*, total offences per 1000 population, as well as proportion of people who: did not finish Year 12, attained higher-educational degree, are food insecure, believe the LGA has good facilities (e.g. shops, libraries, hospitals, etc.), did not meet physical activity guidelines, reported poor/fair health and are experiencing high/very high levels of psychological distress. We chose these dataset as we believed that there might be a correlation between these metrics and sentiment.

5.2.4 Population & People - National Region Profiles 2011-2014

This dataset contains regional profiles on population at the LGA level for 2010-2014 [8]. Specifically, we used *median age* by LGA for 2011. *Age* is important to take into account as, for example, using the *unemployment rate* alone could potentially be misleading as it could simply reflect an atypical age distribution. As such, many metrics should be looked at in conjunction with *median age*.

5.3 GeoJSON

In order to render the polygons of each of the LGAs within our choropleth, a geoJSON containing all polygons was downloaded from AURIN. This geoJSON was created using the *spatial tools* in AURIN and creating a *spatialised aggregated dataset* containing the polygons for each LGA. This was then loaded into our CouchDB database.

[3] https://portal.aurin.org.au/

6 Data Analysis

6.1 Overview of Scenarios

We considered a number of different scenarios in our analysis:

(i) Tweet Dashboard This dashboard was developed to gain a better understanding of the harvested tweets. We wanted to be able to visualise the basic characteristics of our data through a convenient graphical interface that could be periodically updated as more data was processed. This allowed us to monitor our tweet harvester to ensure it was working appropriately, as well as providing an overview of when/where the tweets were collected. The dashboard uses views created in CouchDB using MapReduce queries. It is then rendered in the front-end UI through Flask. Please refer to sections 4 and 7 for more details on MapReduce and front-end UIs respectively.

(ii) Sentiment Analysis to Gauge *Well-being* In order to estimate the *well-being* of Victorians, Government departments collect and analyse many different demographic and socio-economics data points. Although some of these are easily collected, many require expensive surveys to be performed in order to collect the required information and the data is often outdated before it is even analysed. That's where sentiment analysis comes in. We hypothesised that the sentiment of tweets could be correlated to these demographic and socio-economic characteristics and that, potentially, sentiment analysis could used as a substitute to measure *well-being*. Since this data is already collected, it would allow Governments to understand the *well-being* of its citizens without the need for manual surveys. Sentiment analysis was performed across Victorian LGAs using tweet text, combined with their geo-location.

(iii) Topic Modelling What better way to improve our knowledge of life in Australian cities than to identify the most popular topics currently being discussed by Australians on Twitter. Given the significant number of tweets collected, we identified these topics using two *topic modelling* techniques: (a) lexical categorisation using *Empath* [2] and (b) Latent Dirichlet Allocation (LDA) [1]. These two techniques approach the problem very differently and given this unsupervised setting, executing both allowed us to compare results and understand the accuracy of our outputs.

6.2 Notation & Terminology

Below we define some of the key terminology and notations used throughout this section:

- **Token** - individual linguistic units, usually a word denoted as w
- **Document** - a sequence of n tokens denoted by $w = (w_1, w_2, \ldots, w_n)$
- **Corpus** - a collection of m documents denoted by $D = \{\boldsymbol{w_1}, \boldsymbol{w_2}, \ldots, \boldsymbol{w_m}\}$

6.3 Sentiment Analysis to Gauge *Well-being*

As mentioned above, we hypothesised that the sentiment of tweets could be used as a substitute to measure the current *well-being* of Victorians. Since Twitter data is readily accessible, it would allow Governments to understand the well-being of its citizens in real-time, without the need for these manual surveys. Firstly, we collected a number of datasets from AURIN that we expected to be correlated with sentiment and have been used by the Victorian Government to historically measure *well-being* [11]. Please see section 5 for more details on these datasets.

6.3.1 Tokenization

In order to perform our sentiment analysis, our tweet text first needed to be tokenized, i.e. broken into individual linguistic tokens. These tokens are usually words but can represent other things such as emojis. Tokenization was performed using the *TweetTokenizer* from *nltk* [7]. This tokenizer was best suited for this dataset given the unique linguistic units observed in tweets. Firstly, *TweetTokenizer* removes all user handles (i.e. tokens starting with @), replaces any repeated character sequences greater than 3 with only 3 characters and, importantly, keeps emojis as a single token. In addition, all stop words (based on list from *nltk*) were removed, all urls and numbers were replaced with *<url>* *<number>* tags respectively.

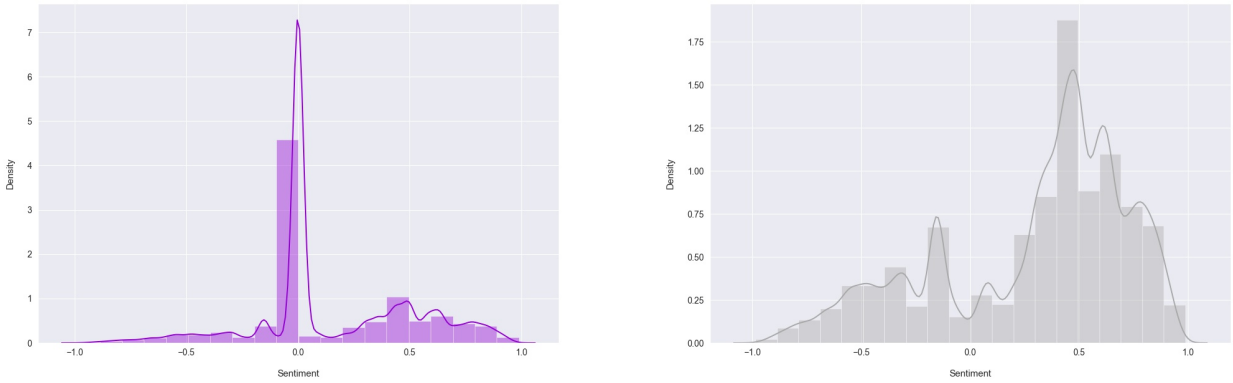
6.3.2 Sentiment Analysis using VADER

Sentiment analysis was performed by measuring grammatical and syntactical conventions that express and emphasise sentiment polarity. Specifically, we utilised *VADER*, a lexicon and rule-based sentiment tool specifically optimised for microblog-like contexts such as Twitter [6]. Not only does *VADER* take into account punctuation and capitalisation to measure intensity but, by analysing preceding *trigrams*, it can also understand:

- Degree modifiers that change the intensity of a statement, e.g. *"good"* versus *"extremely good"*; and
- Conjunctions that indicate a shift the sentiment polarity, e.g. *"the movie was good but..."*

Of particular importance, VADER is able to interpret emojis which are directly correlated with sentiment and are widely used throughout social media. Figure 4 below shows the distribution of sentiments scores. The polarity of sentiment scores are measured from -1.0 to 1.0 where negative and positive scores indicate negative and positive sentiment respectively, with 0

indicating neutral sentiment. We can see that a significant number of tweets have neutral sentiment. When removing those tweets with neutral sentiment, we can see that overall, the tweet sentiment is skewed positively and there are few tweets at the extreme polarities.



(a) *All sentiment scores*

(b) *Without sentiment equal to zero*

Figure 4: *Distribution of sentiment scores*

6.3.3 Allocating Tweets to LGAs

Given we are trying to the compare to data aggregated to LGAs, we only used tweets where latitude and longitude was recorded. This allowed us to map each tweet to an LGA and compare against our AURIN datasets. This allocation process was performed using *Shapely* which allowed us to easily generate polygons and determine if said polygon contained the point/tweet.

6.3.4 Correlations with AURIN Datasets

Figure 5 is a heatmap of the correlations between each of the features. This heatmap measures the statistical relationship, whether causal or not, between each random variable. Looking at sentiment, it can be seen that most features are not strongly correlated. Surprisingly, those metrics one would expect to best represent sentiment, such as adequate work-life balance, adequate sleep, lack of time with family/friends and feeling pressured, all have quite low correlations. The strongest correlation with sentiment is with *degree of psychological distress* with a correlation of -0.33

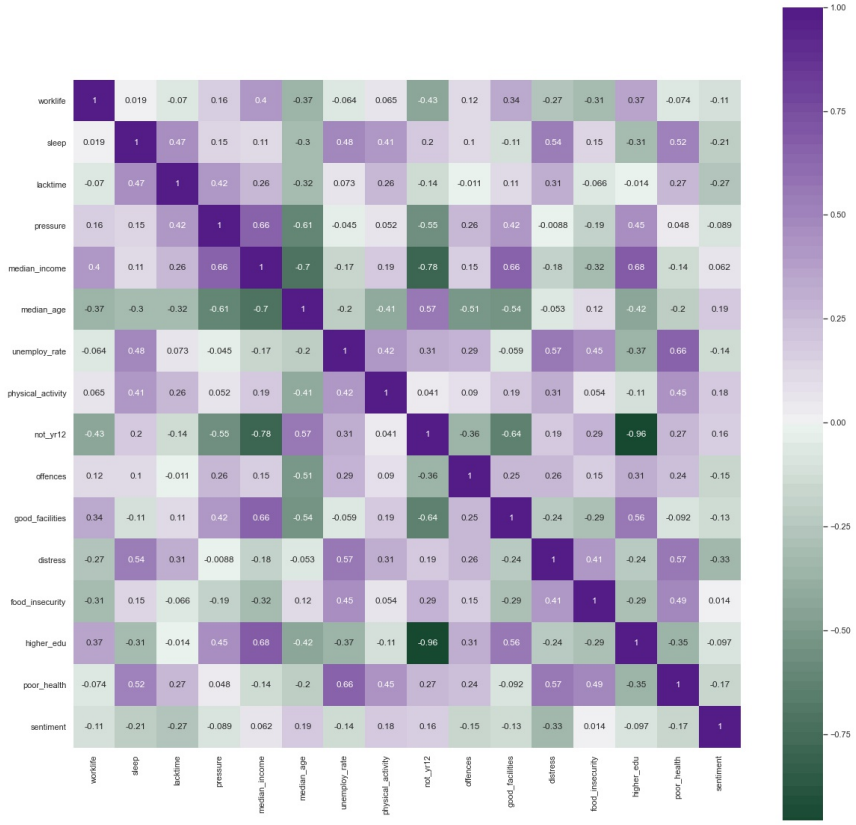


Figure 5: *Heatmap showing the correlation between each feature*

Interestingly, when looking at correlations between other features, there are a few observations that should be highlighted:

- As expected, there is a negative correlation between the proportion of students that do not finish Year 12 and the number of people that pursue higher education. They are nearly perfectly negatively correlated (-0.96). Median income is also significantly negatively correlated (-0.78) with the proportion of students not finishing Year 12.
- Poor health is highly correlated with a number of factors including lack of adequate sleep (0.52), unemployment rate (0.66) and not meeting minimum physical activity requirements (0.45).
- Pursuing higher education tends to lead to a higher median income (0.68).
- Feeling of pressure increases as median income increases (0.66) and decreases with age (-0.61).
- LGAs with higher median incomes tend to have better facilities (0.66).
- Less offences tend to occur in LGAs where the median age is older (-0.51).

6.3.5 Statistical Significance

Although we did identify correlations between the features and sentiment, a formal test was conducted to make our conclusions about statistical significance. To formally test our hypothesis, we used an *F-test*. This is a statistical test where the test statistic

has an F-distribution under the null hypothesis. Here, we tested the hypothesis that there is a statistical difference between the fit of the only-intercept model and model containing each feature.

Utilising `f_regression` from *sklearn*, we calculated both the *F-scores* and *p-values* of each of the features and found that the null hypothesis can only be rejected for one of the features, *psychological distress*, at the 5% significance level. Unfortunately, this means that our hypothesis, that sentiment analysis over Twitter data can be used as a substitute for measuring community *well-being*, is incorrect.

6.4 Topic Analysis

As mentioned above, topic modelling provides a method for automatically summarising documents, here allowing us to summarise our large collection of tweets. We define a topic as a set of tokens that, when taken together, suggest a shared theme. Topic modelling allows us to discover the key and potentially hidden themes of the collected tweets. These techniques basically work by measuring how relevant tokens, within each document, are to each topic. To allow us to better understand the performance of our topic modelling, we implemented two different techniques: (i) lexical categorisation and (ii) LDA. Given these techniques are unsupervised, executing both methods allowed us to compare results and understand the accuracy of our outputs.

6.4.1 Empath

High quality lexicons allow us to analyse language across a broad range of topics. As such, lexical categorisation was performed to identify the top-10 topics across our collection of tweets. This analysis was done using *Empath* [2], a pre-trained lexical categoriser. The *Empath* lexicon was built by extending a deep learning skip-gram network to capture words in a neural embedding. This embedding learns associations between words by calculating the cosine similarity of these words in vector space. The clusters identified map these words into 200 lexical categories. Figure 6 illustrates this process diagrammatically.

Figure 6: *Illustration of Empath lexicon training process* [2]

Empath was applied to our tokenized tweets and returned a score against each topic, essentially providing the likelihood of a tweet being associated with a topic. Given this, we took the category with the highest score as the topic of the tweet. We then counted the occurrence of each topic and identified the top-10 most commonly occurring topics. From this analysis, the top-10 topics were (in descending order): weather, vacation, attractive, domestic work, optimism, wedding, driving, social media, sleep and dance. This is displayed as a doughnut chart in our Tweet Dashboard as seen Figure 8.

6.4.2 Latent Dirichlet Allocation (LDA)

LDA is a generative probabilistic model for the collection of discrete data such as text corpora [1]. The goal of LDA is to find *"short descriptions"* that accurately describe members of a collection, e.g. a document can be summarised by a number of topics and each token is attributable to one of these topics. The probability that a token forms part of a topic is calculated using *Bayesian* methods and *Expectation Maximization* (EM). Basically, within a topic, certain terms will be used much more frequently than others and we use this fact to perform topic modelling.

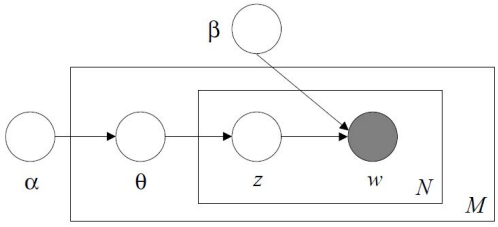


Figure 7: *Graphical model representation of LDA where outer plate represents each document and the inner plate represents the repeated choice of topics and tokens within a document* [1]

As part of this analysis, we were required to perform some additional augmentations to our dataset:

(i) Lemmatize Reduces inflectional forms of a word to a common base form, e.g. walked, walking become walk. This was performed over our tokenised tweet text using part-of-speech tagging and WordNet lemmatizer from *nltk*.

(ii) BOW vs. TF-IDF Topic modelling was performed using two different sets of features: bag-of-words (BOW) and term frequency-inverse document frequency (TF-IDF). A BOW is a collection of all tokens across all documents in a corpus with their respective occurrence count. Although BOW has been found to be an adequate representation, TF-IDF better reflects the importance a token has within a document as it increases proportionally each time a token appears in a document but is adjusted by to reflect the number of documents it appears in.

After performing the above, we trained our LDA model; this was done using *gensim*. LDA initially randomly assigns each token to one of k topics. Here, k has to be chosen beforehand which is quite difficult given Twitter covers a multitude of topics. We tuned this hyperparameter by running the analysis multiple times, stopping when the lowest *coherence* was achieved. *Coherence* measures the relative distance between words within a topic and is generally used as a methods of scoring the performance an LDA model. LDA calculates the probability of each token belonging into a topic using the following equation: $P(w|t) = P(t|d)P(w|t)$. After running this analysis, topics are output with the combination of words that best describe the topic. Below is an example of such output:

```
Word:  0.340*"<number>" + 0.058*"c" + 0.056*"today" + 0.054*"wind" + 0.054*"humidity" +
     0.051*"temperature" + 0.049*"barometer" + 0.048*"hpa" + 0.036*"°" + 0.030*"h"
```

Clearly this topic relates to weather, which also happened to be the most common topic identified using *Empath*. Although other topics were clear, some of the grouped words were puzzling. Table 1 compares the performance of our LDA model using both BOW and TF-IDF feature sets. Clearly, in this instance BOW features are preferred.

Table 1: *Performance of Topic Modelling Techniques*

Feature Type	**BOW**	**TF-IDF**
uMass Coherence	-6.62	-7.36

Although LDA performed well, we still preference *Empath* due to a number of reasons. Unlike *Empath*, topic labels are not automatically provided, only the tokens that make-up that topic. As such topics need to be assigned by the user and can require specialised knowledge depending on the subject matter. Another disadvantage of this technique is that the number of topics needs to be specified before training. This number needs to be correct in order for the model to perform well and therefore, without any underlying knowledge regarding these tweets, the model needed to be trained multiple times to assess performance and select the correct number of topics. With an ever changing set of tweets, this makes our analysis process more repeatable with *Empath*.

7 Front-end User Interface

In this section, we detail our front-end architecture used to visualise our analytical results. This front-end UI is a web-based application that utilises *Flask* and several Python and JavaScript libraries.

7.1 Design

The front-end architecture leverages both Flask to create back-end API endpoints and Node.js HTTP-server to render our HTML pages. Flask, a Python library for micro web application frameworks, allows a user to keep any application simple and scalable. For our application, each Flask endpoint accessed a MapReduce query from CouchDB. Through specific functions, each dataset from CouchDB was transformed to match the format required for visualisation. In this project, we specifically used *Plotly.js* and *Leaflet.js* to visualise our data.

MRC Instance The instance in MRC where our CouchDB database is located.

```
base_url = http://172.26.130.129:5984
```

CouchDB How our CouchDB database is accessed.

```
headers = {'Authorization': "Basic YWRtaW46bXJjcGFzc3dvcmRjb3VjaA=="}
couch_url = "http://admin:mrcpasswordcouch@172.26.130.129:5984/"
couch = couchdb.Server(couch_url)
```

MapReduce Each MapReduce *view* is then accessed through the following requests method.

```
r = requests.get(url, headers=headers)
data = json.loads(r.text)
```

7.1.1 Cross-Origin Resource Sharing (CORS)

When Flask is spun-up, a Node.js server accesses the Flask IP server where the data is being stored. To enable the communication between the Node and Flask servers, we implemented CORS. CORS is an HTTP-header based mechanism that allows a server to specify origins (e.g. domains, ports, etc.) other than its own from which it is permitted to read information from a web browser. As we are leveraging Flask to create end-points, we need CORS to allow a Node.js server to communicate with these end-points to access our data. Basically, CORS is a bridge between both servers and the browser.

```
from flask_cors import CORS
CORS(app, resources={r"*": {"origins": "*"}})
```

7.1.2 Node.js

We leveraged a Node.js server by calling an HTTP-server from the command line and then visiting the local host. This is a fast and secure way to test our application, which is also suitable for deployment. By spinning up the Node.js server, the client can pick-up the data from the Flask end-points, simplifying the communication between front and back-end.

7.2 Java Script

In each HTML (where we are rendering data from an end-point), we capture the data by assigning a variable to each end-point in JS. An example of this is provided below. Each of these end-points' payloads are processed asynchronously with D3 and, leveraging a series of JS libraries, we are able to create meaningful data visualisations.

```
var state_count = "http://" + ip_addr + "/api/dashboard/stateCounts"
var map_data = "http://"+ ip_addr +"/api/dashboard/locationCounts"
var daily_count = "http://"+ ip_addr +"/api/dashboard/dailyCount"
var hourly_count = "http://"+ ip_addr +"/api/dashboard/hourlyCount"
```

7.2.1 D3.js

D3 is a powerful open-source JS library mainly used to create graphs. However, D3 is also a suitable tool to handle calls to document object models (DOM), therefore seamlessly capturing HTML elements that we can then add JS functions to. We are utilising version 5.5 of D3.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/5.5.0/d3.js"></script>
```

The `.json()` method is a simple tool to retrieve data from each end-point. Chained with the `.then()` method, we can asynchronicity:

```
d3.json(api_variable).then((object) => {}
```

Thus, most DOM manipulations in our application are done through D3 which also incorporates simple functionality to create new HTML elements and pass new data across:

```
d3.select("#selState").selectAll("option")
          .data(statesObject)
          .enter()
          .append('option')
          .html(statesObject => statesObject.state);
```

7.2.2 Plotly.js

Plotly is an open-source JS library that allows a user to easily create a wide variety of graphs. The key elements (i.e. objects) of a Plotly function are *trace*, *layout* and *HTML Div*. *Trace* defines the axis, passes the data and defines the graph type. *Layout* is where the visuals of the graph can be manipulated. Finally, the *HTML div* element points indicate where in the web page we want our graph rendered.

```
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
```

Each of these elements are then passed through to each new plot:

```
Plotly.newPlot("mydiv", trace, layout);
```

We implemented Plotly to render the following graphs in our Twitter Dashboard. Each of these graphs renders data from a different flask end-point and is manipulated through a unique function:

- Tweet Count per State
- Topic Modelling
- Daily Tweet Count for a chosen state
- Hourly Tweet Count for a chosen state

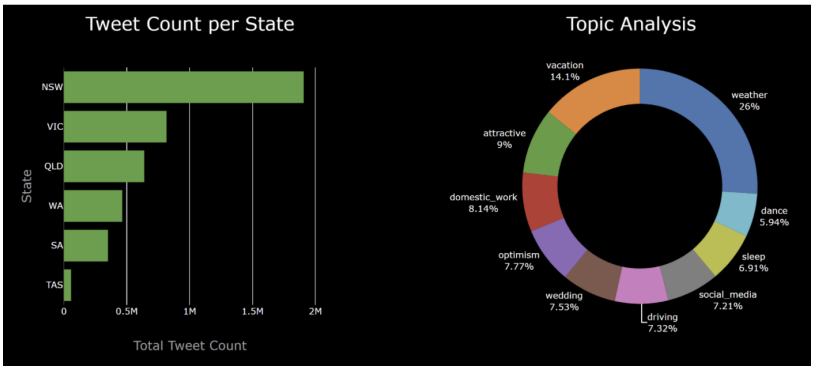


Figure 8: *Screenshot from Twitter Dashboard of tweet count per state and topic modelling*

7.2.3 Leaflet.js

Leaflet is an open-source library used to render interactive maps. All maps in our application are implemented using this library.

```
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.3.3/dist/leaflet.css"
    integrity="sha512-Rksm5RenBEKSKFjgI3a41vrjkw4EVPlJ3+OiI65vTjIdo9brlAacEuKOiQ5OFh7cOI1bkDwLqdLw3Zg0cR
    ↪ JAAQ=="
    crossorigin=""/>
<script src="https://unpkg.com/leaflet@1.3.3/dist/leaflet.js"
    integrity="sha512-tAGcCfR4Sc5ZP5ZoVz0quoZDYX5aCtEm/eu1KhSLj2c9eFrylXZknQYmxUssFaVJKvvc0dJQixhGjG2yXW
    ↪ iV9Q=="
    crossorigin=""></script>
```

Maps are created via different layers. The map is instanced through the `L.map(\div",L.tileLayer())` method where *HTML div* is referenced and the first layer passed through. The initial layer of the map is called *tileLayer* and assigned to a variable. Leaflet provides access to a wide variety of map styles which are accessed through a unique ID and subsequent API URL. These values are then passed to the corresponding attributes in the *L.tileLayer* object as seen below:

```
var darkmap = L.tileLayer("https://api.mapbox.com/styles/v1/mapbox/{id}/tiles/{z}/{x}/{y}?access_token={
↪ accessToken}",
↪ {
    id: "dark-v10",
    accessToken: API_KEY
});
```

Here, an API call is made to *Mapbox*, the client that holds the basemap and where more layers will be passed and rendered on the web page. From the many available layers in Leaflet, we are using a total of two layers across our application. These layers are added to the map variable created with the `L.tileLayer()` method. Via pre-processing the data and transforming it into a geoJson format, we allowed for maximum interoperability between Flask and Leaflet.

Vector Layer - *circleMarker* Accessing the *circleMarker layer* required the `markerclust.js` script below. This layer was created using the `L.geoJSON()` method and pointToLayer parameter. circleMarker attributes were chosen to reflect the radius and colour of these markers. A circleMarker object was then returned for each data point (latitude and longitude) on the map.

```
<script type="text/javascript"
↪  src="https://unpkg.com/leaflet.markercluster@1.0.3/dist/leaflet.markercluster.js"></script>
```

To enhance the visualisation, we included the `onEachFeature` parameter that bound pop-ups to the map and added them to the layer. This process was implemented for the maps in *Maps Dashboard* and *Dashboard* pages, an example screenshot for which can be seen in Figure 9.

```
var twitts = L.geoJSON(twitData, {
            pointToLayer: function (feature, latlng) {
              var geojsonMarkerOptions = {
                  radius: feature.properties.count/10000,
                  fillColor: getColour(feature.properties.count),
              };
              return L.circleMarker(latlng, geojsonMarkerOptions)
            },

            onEachFeature: onEachFeature
          }).addTo(myMap);
```

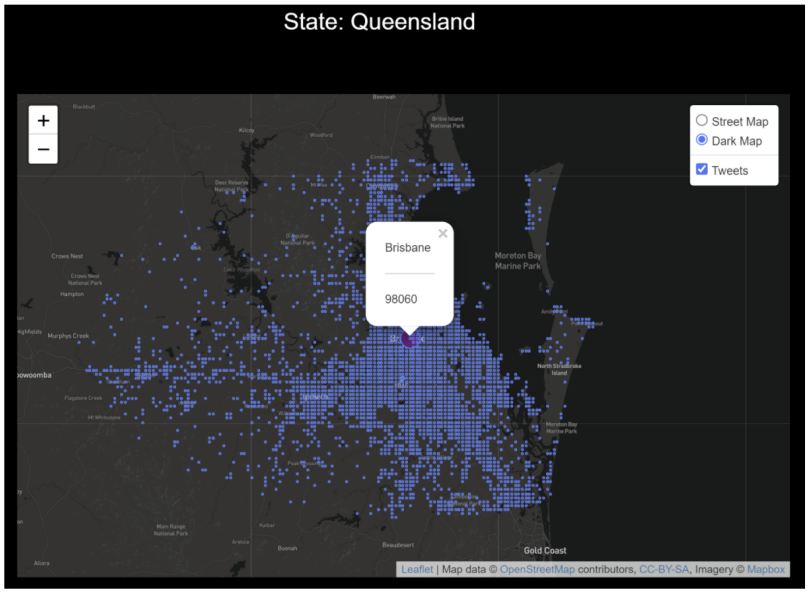


Figure 9: *Screenshot of map with circle marker*

Choropleth Plugin To enable the Choropleth plugin, an additional choropleth JS script was required for the HTML. We injected the following scripts into the HTML:

```
<!-- Leaflet-Choropleth JavaScript -->
<script type="text/javascript" src="static/js/choropleth.js"></script>
<!-- OUR JS -->
<script type="text/javascript" src="static/js/choropleth_logic.js"></script>
```

The sentiment analysis was visualised through this choropleth plugin available through Leaflet. Once the `L.tileLayer()` was initialised, we created and attached the choropleth layer to the map through the `L.choropleth()` method. The key element inside this object is the *valueProperty* key. In our choropleth, sentiment score determines what colour will be visualised based on the colour scale given to the object.

```
geojson = L.choropleth(data, {

    // Define what  property in the features to use
    valueProperty: "sentiment",

    // Set color scale
    scale: ["#d1d9d9","#9fe6a0", "#aa2ee6"],

    // Number of breaks in step range
    steps: 10,

    // q for quartile, e for equidistant, k for k-means
    mode: "q",
    style: {
      // Border color
      color: "#fff",
      weight: 1,
      fillOpacity: 0.8
    },
```

Through the `onEachFeature` function inside our `L.choropleth` object, we added *event listeners* like *mouseovers* and *bindPopUp* to display data endemic to the highlighted region.

```
// Binding event listeners to the choropleth layer.
onEachFeature: function(feature, layer) {
  layer.bindPopup("Sentiment value: " + feature.properties.sentiment + "<br>Median Income:<br>" +
    "\$" + feature.properties.median_income);
  layer.on({
    mouseover: highlightFeature,
    mouseout: resetHighlight
  });
}
}).addTo(myMap);
```

Figure 10: *Screenshot of choropleth*

7.3 HTML

7.3.1 Bootstrap

Bootstrap is a front-end JS and CSS framework that allows us to add responsiveness and structure to HTML documents. We implemented a series of *bootstrap classes* like *Navbar*, *Columns*, *Grids* and *Cards*, to display our visualisations. Bootstrap requires both CSS and JS scripts.

```
<!-- Bootstrap CSS -->
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/css/bootstrap.min.css" rel="stylesheet"
↪  integrity="sha384-wEmeIV1mKuiNpC+IOBjI7aAzPcEZeedi5yW5f2yOq55WWLwNGmvvx4Um1vskeMj0"
↪  crossorigin="anonymous">

<!-- Bootstrap JS-->
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"
↪  integrity="sha384-IQsoLXl5PILFhosVNubq5LC7Qb9DXgDA9i+tQ8Zj3iwWAwPtgFTxbJ8NT4GN1R8p"
↪  crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0/dist/js/bootstrap.min.js"
↪  integrity="sha384-lpyLfhYuitXl2zRZ5Bn2fqnhNAKOAaM/0Kr9laMspuaMiZfGmfwRNFh8HlMy49eQ"
↪  crossorigin="anonymous"></script>
```

7.4 Web Pages

Our front-end is comprised of five pages:

(i) Home Where we describe the type of analysis we implemented in our application

(ii) Choropleth Where we render the sentiment analysis through our choropleth visualisation

(iii) Dashboard Provides a summary and overview on the amounts of tweets through various levels of abstraction: state, hour, day, location and most tweeted topics.

(iv) Map Dashboard Designed to speed-up visualisations of tweet by location for each state. Via indexing our database and segregating the data per state, we are able to load each region faster.

(v) Team Where we provide the profiles of each of the team members behind this project.

7.5 NGINX

Proxying is often used combat a number of issues such as spreading the load among several servers and pass requests over protocols other than HTTP. When NGINX proxies a request, this request is sent to a specific proxied server, where responses are fetched and sent back to the client. It is possible to proxy requests to an HTTP server or a non-HTTP server which can run an application developed with a specific framework using a specified protocol. To pass a request to an HTTP proxied server, the proxy pass directive is specified inside a location.

In our case, the back-end API was running on Flask server port 5000, whereas the front-end was running on Node.js HTTP-server port 8080. Both of these are inside the Docker containers which can run on separate ports. Thus, we could not hard code these port numbers in our HTML or JS scripts. Hence, we added NGINX Server with reverse proxy to resolve the dependencies on the ports (5000 and 8080) and perform URL rerouting to the required Docker containers where the front-end and back-end was deployed. To do this, we had to install the NGINX server using Ansible and then add the configuration files defining the proxy pass directives:

```
- name: install nginx
    become: yes
    apt:
        name: nginx
        state: present

- name: copy code file
    become: yes
    copy:
        src: nginx.conf
        dest: /etc/nginx/nginx.conf
        owner: root
        group: root
```

```
location /api/dashboard/ {
    add_header Access-Control-Allow-Origin *;
    proxy_pass http://127.0.0.1:5000\$request_uri;
}

location /dashboard/ {
    add_header Access-Control-Allow-Origin *;
    proxy_pass http://127.0.0.1:8080\$request_uri;
}
```

7.6 Issues & Challenges

One of our main challenges was managing speed. The map in our dashboard is rendering all tweets across Australia which is approximately 4 million plus data points. In order to circumvent this issue, we decided to re-index the database for each state and create a new web page, *Maps Dashboard*, with separate maps for each state. In this regard, we are loading a fifth or less of data points in each map, considerably speeding-up loading time.

We also encountered issues with different versions of JS libraries. We were using a combination of syntaxes out of which a particular method to load data asynchronously through anonymous function was deprecated in older versions of D3 and therefore we were not able to access the data. We discovered that the `.then()` method was the correct syntax in the latest version of D3.

8 Conclusions & Future Work

In this report, we discussed and evaluated the use of a cloud-based solution to harvest tweets from each major city in Australia. We then developed and performed a range of analytical scenarios to test the hypothesis that Twitter data could be used efficiently to measure *well-being* and monitor trends in the population.

We managed to successfully deploy a cloud-based solution that exploited VMs and containers on the Melbourne Research Cloud (MRC), used both the *Streaming* and *Search API* interfaces to harvest tweets and then stored our consolidated tweets in our CouchDB database. Following on from this, we performed additional analysis using MapReduce, a programming model used to analyse large quantities of data through a filter, sort and reduce method which was ideal for our dataset and use case.

A number of different scenarios were considered and compared for the analysis, culminating in the use of a **Tweet Dashboard**, **Sentiment Analysis to gauge *well-being***, and **Topic Modelling**. In particular, **Topic Modelling** we used of two differing techniques: (a) lexical categorisation and (b) Latent Dirichlet Allocation (LDA). Comparisons to AURIN datasets yielded a set of interesting features and the correlations yielded some interesting insights. However, in the end, only one feature was found to be statistically significant and therefore we concluded that unfortunately sentiment analysis over Twitter data could not be used as a substitute for measuring and monitoring community *well-being* in real-time.

Finally, we summarised our analytical results in a web-based front-end UI that utilises *Flask* and several Python and Java Script libraries. This front-end architecture renders each HTML page using a combination of CSS, Bootstrap and NGINX, as well as *Plotly.js* and *Leaflet.js* which were the main libraries we used to visualise our analysis. In the end, our front-end consisted of $five$ pages covering the type of analysis performed *(Home)*, sentiment analysis visualisation *(Cholopleth)*, a tweet dashboard containing key characteristics of our collected tweets *(Tweet Dashboard)*, indexed data segregated by state *(Map Dashboard)*, and profiles of our team members *(Team)*.

Across the report we encountered a number of issues and challenges, however were able to navigate these difficulties and source effective methodologies for defining and testing our hypothesis. In future, in order to better capture and analyse sentiment and undertake topic modelling, it would be beneficial to increase the volume of data harvested substantially, as well as having multiple collection instances over a longer period of time so as to reduce the risk of point-in-time biases or event-driven anomalies skewing the results.

9 References

[1] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent Dirichlet Allocation. *Journal of Machine Learning Research 3* (2003), 993–1022.

[2] FAST, E., CHEN, B., AND BERNSTEIN, M. S. Empath: Understanding topic signals in large-scale text. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2016), CHI '16, Association for Computing Machinery, p. 4647–4657.

[3] GOVERNMENT OF THE COMMONWEALTH OF AUSTRALIA - AUSTRALIAN BUREAU OF STATISTICS. LGA Estimates of Personal Income - Employee Income 2010-2015, 2017.

[4] GOVERNMENT OF VICTORIA - DEPARTMENT OF HEALTH AND HUMAN SERVICES (DHHS). VIC DHHS - Local Government Area Profiles Data (LGA) 2011, 2012.

[5] HASAN, A., MOIN, S., KARIM, A., AND SHAMSHIRBAND, S. Machine learning-based sentiment analysis for twitter accounts. *Mathematical and Computational Applications 23*, 1 (2018).

[6] HUTTO, C. J., AND GILBERT, E. VADER: A parsimonious rule-based model for sentiment analysis of social media text. *Proceedings of the 8th International Conference on Weblogs and Social Media, ICWSM 2014* (2014), 216–225.

[7] LOPER, E., AND BIRD, S. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1* (USA, 2002), ETMTNLP '02, Association for Computational Linguistics, p. 63–70.

[8] OF THE COMMONWEALTH OF AUSTRALIA AUSTRALIAN BUREAU OF STATISTICS, G. Lga population & people - national regional profile 2010-2014, 2016.

[9] SINNOTT, R., BAYLISS, C., BROMAGE, A., GALANG, G., GRAZIOLI, G., GREENWOOD, P., MACAULAY, A., MORANDINI, L., NOGOORANI, G., NINO-RUIZ, M., TOMKO, M., PETTIT, C., SARWAR, M., STIMSON, R., VOORSLUYS, W., AND WIDJAJA, I. The australian urban research gateway. *Concurrency and Computation: Practice and Experience 27* (04 2014), 358–375.

[10] TWITTER. Rate limits: Standard v1.1, 2021.

[11] VICTORIAN HEALTH PROMOTION FOUNDATION. VicHealth Indicators Survey 2011. Tech. rep., VicHealth, Melbourne, Australia, 2011.

10 Appendix

10.1 GitHub Repository

The Github repository can be found here. This repository is not public but we can add the required members as needed.

10.2 Presentation

Cloud Architecture https://www.youtube.com/watch?v=3ITSOFSybeM

Analysis and Front-end UI https://youtu.be/M3r1tvIB5qU

10.3 User Guide

To launch the applications on MRC, please repeat the following steps:

(i) From the Github repo found here, clone the repository onto your local machine.

(ii) Create a key-pair on MRC under our project unimelb-comp90024-2021-grp-19 and downloaded the private key. Rename this file to new_key_sem.pem and add to the root of the directory cloned from Github.

(iii) Install python3, pip3 on your machine if not already installed. You can read about installing python3 and pip3 on the internet, but usually all latest UNIX/LINUX based OS comes with python3 installed.

(iv) Install Ansible using pip

```
sudo pip3 install ansible
```

(v) Download the OpenRC file from the MRC dashboard. You can find it at the top right corner on the Menu under User section and add to the root of the directory cloned from Github.

(vi) One by one you can run the .sh files for deploying CouchDB, harvester and analysis, frontend and backend instances. Use the below commands in the sequence.

```
sh run_couchdb.sh
```

This step will require two passwords - one you have generated from MRC and other is sudo password of your local machine, to run some of the step of Ansible playbook in root mode.

(vii) Once the CouchDB instance is created, some of the manual steps are done. This can be automated as well through Ansible. To attach the volume we login to MRC instance and follow some commands:

```
ssh -i ~/.ssh/new_key_sem.pem ubuntu@<ip-address>
sudo mkfs -t ext4 /dev/vdb
sudo mkdir /data
sudo mount /dev/vdb /data
vi /etc/fstab
```

Go to insert mode, add below line and save the file

```
/dev/vdb /data auto defaults 0 0
```

Stop CouchDB and Continue with following commands to update the data folder for CouchDB

```
sudo systemctl stop couchdb
sudo mkdir /data/couchdbData
sudo chown -r couchdb:couchdb /data/couchdbData
sudo vi /opt/couchdb/etc/default.ini
```

Go to Insert mode and Update path of these two variables

```
database_dir = /data/couchdbData
view_index_dir = /data/couchdbData
```

Save the file and exit vi and then start CouchDB via command

```
sudo systemctl start couchdb
```

(viii) To create the Map Reduce View, open the CouchDB url by substituting correct ip address to the link and entering credentials:

```
url - http://ip-address:5984/_utils/
user id - admin
password - mrcpasswordcouch
```

(ix) Once logged into CouchDB, go to twitterfeed database, and add a document. You can copy paste the document from Github Repo folder mapReduceDocuments/AllMapReduceViews.json. This will create all the map reduce views we have used in our project.

(x) Setup Replication from CouchDB fauxton UI to the other CouchDB server forked.

(xi) Once we have CouchDB up and running, we need to change the IP Address of the CouchDB in the files:

- `ansible/roles/frontend-docker-install/tasks/backend-api/app.py`
- `ansible/roles/harvester-docker-install/tasks/harvester/TwitterHarvester.py`
- `ansible/roles/harvester-docker-install/tasks/analysis/analysis.py`

and then run the harvester and front-end script one-by-one.

```
sh run_harvester.sh
sh run_frontend.sh
```

Once both the scripts has successfully run, you can see the UI dashboard by adding the correct IP of the frontend server instance to the URL `http://ip-address/dashboard/dashboard.html`.

10.3.1 Creating Instances on MRC

Below is a detailed description of how instances were created on the MRC:

(i) Login into the MRC dashboard using university credentials and select the project.

(ii) Generating Key-Pair

Compute $\rightarrow$ *Key pairs* $\rightarrow$ *Create a Key Pair*

This process can be performed using the dashboard. Executing the above steps generates and returns a private key. A key-pair can also be generated on a user's local machine and then the public key mapped on the MRC to their private key.

(iii) Launching an Instance

Compute $\rightarrow$ *Instance* $\rightarrow$ *Launch Instance*

To performing this process on the MRC dashboard, the following steps were followed:

(a) *Details*

- Provide a suitable *Name* and *Description* for the instance.
- Select *Availability Zone*. Given the MRC only has one one zone, `melbourne-qh2-uom` is set by default.

(b) *Source* - Add the OS image for your instance from a list of available templates.

(c) *Flavor* - Select the RAM and core configuration from the list of available options; should choose the most suitable option for the instance given your processing requirements.

(d) *Security Groups* - Enable the default and SSH security group.

- To add a new Security Group, follow:

  Network $\rightarrow$ *Add New Security Group*

  Provide the port number and other details.

- This new security group will be available in the instance configuration settings.

(e) *Public Key* - Add generated public key (as discussed above).

(f) Launch Instance

(g) *On local* - move the private key in the `.ssh` folder, if key was generated using MRC dashboard. Then using the terminal, login to the server using the following command:

```
ssh -i .ssh/ pvtkey.pem ubuntu@ <instance-ip-address>
```

10.4 Additional Graphs from Analysis

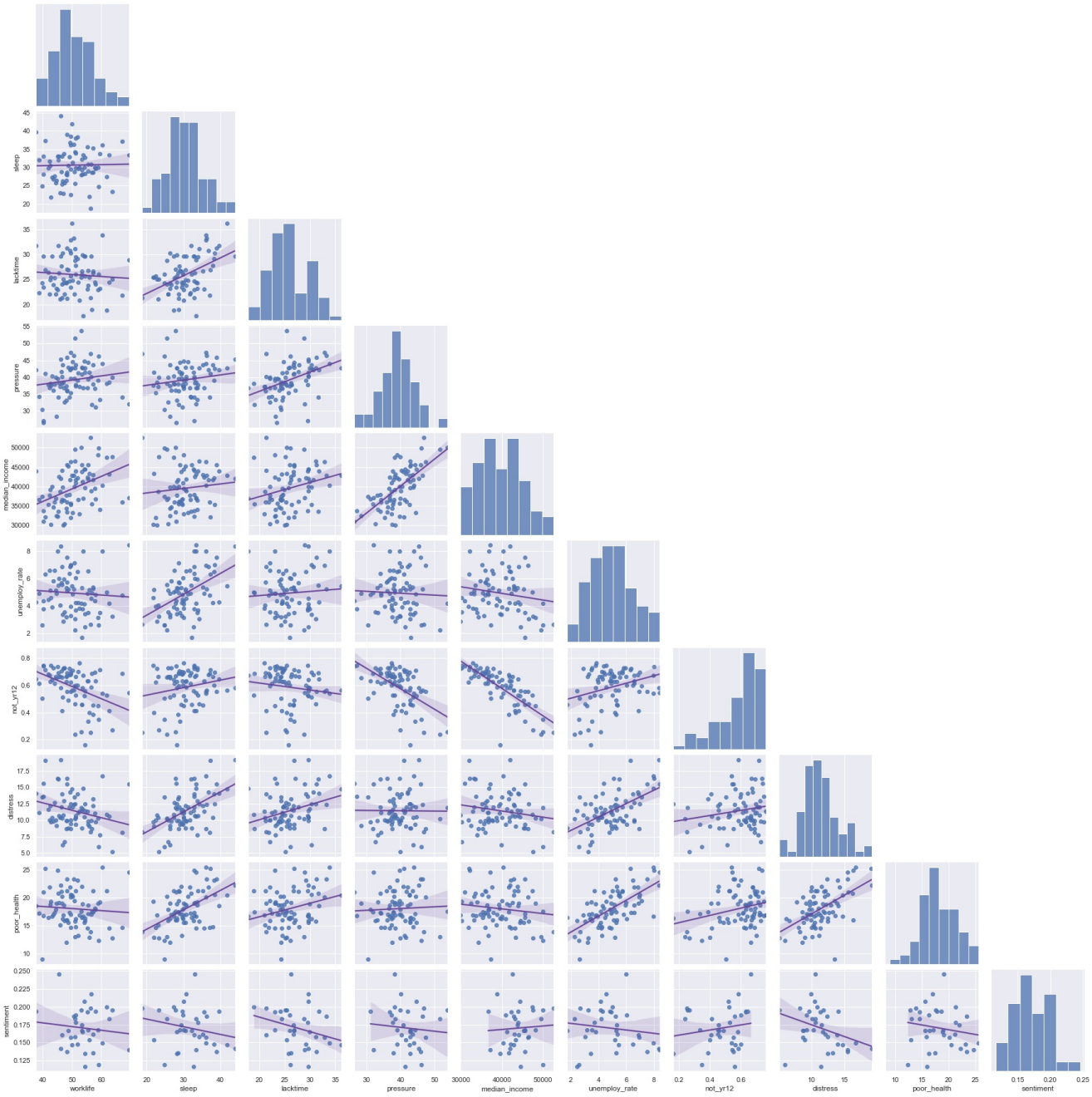


Figure 11: *Plotting pairs of features against one another*

10.5 Backend APIs

Below is a detailed list of all the implemented back-end APIs:

(i) `GET /api/dashboard/stateCount`

Description: This API returns the number of tweets originating from each state

Response: application/json

Payload:


```
[
  {
    "state": "QLD",
    "value": 639083
  },
  {
    "state": "NSW",
    "value": 1906722
  }
]
```


Data Dictionary:

Attribute	Type	Description
State	*String*	Australian state
Value	*Integer*	Total tweet count

(ii) `GET /api/dashboard/locationCounts`

Description: This API returns the number of tweets aggregated over latitude and longitude, rounded to 3 digits in CouchDB and MapReduce.

Response: application/json

Payload:


```
[
  {
    "geometry": {
      "coordinates": [
        10.21,
        123.759
      ],
      "type": "Point"
    },
    "properties": {
      "count": 5,
      "place": "Melbourne CBD"
    },
    "type": "Feature"
  }
]
```


Data Dictionary:

Attribute	Type	Description
geometry	*Object*	<table><tr><th>Attribute</th><th>Type</th><th>Description</th></tr><tr><td>**coordinates**</td><td>list of double</td><td>latitude, longitude</td></tr><tr><td>**type**</td><td>String</td><td>describes coordinate type - Point or Polygon</td></tr></table>
property	*Object*	<table><tr><th>Attribute</th><th>Type</th><th>Description</th></tr><tr><td>**count**</td><td>Integer</td><td>Number of tweets recorded</td></tr><tr><td>**place**</td><td>String</td><td>place name where tweets originated from</td></tr></table>
type	*String*	Feature

(iii) `GET /api/dashboard/locationCountsByState\?stateCode=`

Description: This API returns the number of tweets aggregated over latitude and longitude, rounded to 3 digits in CouchDB and MapReduce.

Request param: stateCode to filter the data for a particular stateCode. State Codes are: {VIC, QLD, NT, WA, SA, TAS, NSW}

Response: application/json

Payload:


```
[
  {
    "geometry": {
      "coordinates": [
        10.21,
        123.759
      ],
      "type": "Point"
    },
    "properties": {
      "count": 5,
      "place": "Melbourne CBD"
    },
    "type": "Feature"
  }
]
```


Data Dictionary:

Attribute	Type	Description
geometry	*Object*	<table><tr><th>Attribute</th><th>Type</th><th>Description</th></tr><tr><td>**coordinates**</td><td>*list of doubles*</td><td>latitude/longitude</td></tr><tr><td>**type**</td><td>*String*</td><td>describes coordinate type {Point, Polygon}</td></tr></table>
property	*Object*	<table><tr><th>Attribute</th><th>Type</th><th>Description</th></tr><tr><td>**count**</td><td>*Integer*</td><td>Number of tweets recorded</td></tr><tr><td>**place**</td><td>*String*</td><td>place name where tweets originated from</td></tr></table>
type	*String*	Feature

(iv) `GET /api/dashboard/dailyCount`

Description: This API returns number of tweets per state aggregated over each day

Response: application/json

Payload:


```
[
  {
    "day": "Fri",
    "state": "QLD",
    "value": 639083
  },
  {
    "day": "Mon",
    "state": "NSW",
    "value": 1906722
  }
]
```


Data Dictionary:

Attribute	Type	Description
hour	*Integer*	Hour in 24hr format
state	*String*	Australian State
value	*Integer*	Total tweet count for that hour

(v) `GET /api/dashboard/hourlyCount`

Description: This API returns the number of tweets per state aggregated over each hour in a 24hr format.

Response: application/json

Payload:


```
[
  {
    "hour": 23,
    "state": "QLD",
    "value": 639083
  },
  {
    "hour": 1,
    "state": "NSW",
    "value": 1906722
  }
]
```


Data Dictionary:

Attribute	Type	Description
hour	*Integer*	Hour in 24hr format
state	*String*	Australian State
value	*Integer*	Total tweet count for that hour

10.6 Team Member Roles

Team Member	Tasks Performed
Trina Dey	Twitter Harvester, CouchDB
Soham Mandal	MRC, Ansible
Ankita Dhar	MapReduce
Hilary McMeckan	Analysis, Report
Ezequiel Gallo	Front-end User Interface