Investigating the cause and effect of obesity in Melbourne, Australia with the Cloud using Twitter and AURIN data

COMP90024 - Cluster and Cloud Computing Semester 1 Assignment 2

TEAM 9

Name	Student ID
Duoyi Zhang	956812
Hongtao Ni	938737
Kuldeep Suhag	919397
Naiyun Wu	1008438
Zexian Huang	1012710

Contents

Introduction

For the second assignment of COMP90024-Cluster and Cloud Computing, the assignment is to tell a story in Australia regarding one or more of the 7 deadly sins using data from Twitter and AURIN. The goal is to design and implement a cloud-based solution to collect, process and analyze the data to tell the story. This cloud-based solution must be automated and scalable using various technologies taught in class. The result should be displayed on a publicly accessible website.

Scenarios

Obesity was decided to be the main topic that we are going to investigate. According to the Australian Bureau of Statistics, ⅔ of the Australian population over 18 were overweight or obese in 2017-2018. In the same report, a fourth of the Australian population from 5-17 of age were obese or overweight [1]. With so many people becoming overweight or obese, those who are not are using social media to “fat shame” such people. Even Game of Throne star Sophie Turner came under fire for being fat shamed and she even “considered suicide” [2].

With so many overweight individuals in Australia that could potentially come under the scrutiny of the “fat shaming” movement on social media, it is inevitable that a few becomes depressed, lazy, angry or hateful. In this project, our team will build a system which can collect social media data from twitter and analyze their sentiment and determine whether it belongs in one of three sins: Gluttony, Sloth and Wrath, with the focus being in Melbourne.

We want to first determine whether the sin of gluttony is the primary reason for obesity in Melbourne. We will use sentiment analysis and determine how many tweets are related to the sin of Gluttony and whether this correlates to the “*number of overweight or obese people*” data in AURIN.

Then we can use the same AURIN data for obesity and compare it to the number of tweets related to sloth to check for correlation. Thereafter, we can use the obesity AURIN data and compare it with the AURIN sedentary data.

Finally, we can also use the AURIN data for obesity and compare it to the tweets related to wrath as well as assault AURIN data.

To summarize, we are investigating the cause of and effect of obesity in Greater Melbourne District and will compare.

- Obesity AURIN data vs Gluttony Tweets
- Obesity AURIN data vs Sloth Tweets
- Obesity AURIN data vs Wrath Tweets

Hypothesis

- Gluttony is the primary cause of obesity, areas with higher concentration of tweets related to Gluttony will have more people who are obese or overweight.
- Areas with higher obesity will have higher tweets related to sloth and more sedentary counts.
- Areas with higher obesity will have higher tweets related to wrath and more assault counts.

Requirements

To be able to do the analysis, we need a system which can collect twitter data. The system must also be able to process, analyze and display both the twitter data and AURIN data.

Functional Requirement:

- **Harvester** script with sentiment analysis capabilities to collect relevant Twitter data
- **Database** to store the collected Twitter/AURIN data by the harvester.
- **Application server**
    - Get data from the database
    - Process the data
    - Send the data to the web frontend
- **Web server**
    - Get the data from application server
    - Generate HTML for users to connect to via the browser to view the processed data

Non-Functional Requirements:

- Easy to deploy/setup
  - Database, application and web server's setups must be automated for fast deployment
- Scalability
  - Must use a cluster for database so more database nodes can easily be added for faster data collection if needed
  - Since we are dealing with big data, the system must be able to scale to handle the 4 v's of big data: Velocity, Volume, Variety, Veracity
- Failure tolerance
  - Database data must be replicated so one node failure does not result in the loss of data
  - When one harvester fails, system should not be affected besides the loss in speed in the collection of data
  - When one node of the web service fails, other methods must still be available to connect to the web service
- Portability
  - Our software must be contained and easily ported to any system. The dependencies of our software should not affect deployment.

System Design and Architecture

Basic System Overview

There are 4 main components of this system. First is the twitter harvester which gets the data from Twitter using Twitter API. Then comes the database which the harvester writes to continuously. Our application server periodically pulls the newest data from the database and saves in a cache. Finally, the web server reads from the cache and displays the data to the user when it receives a HTTP request.

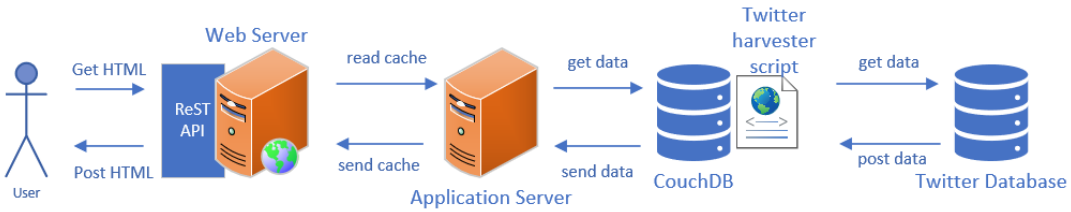


Figure 1 Simple System Architecture of the System

Detailed System Architecture

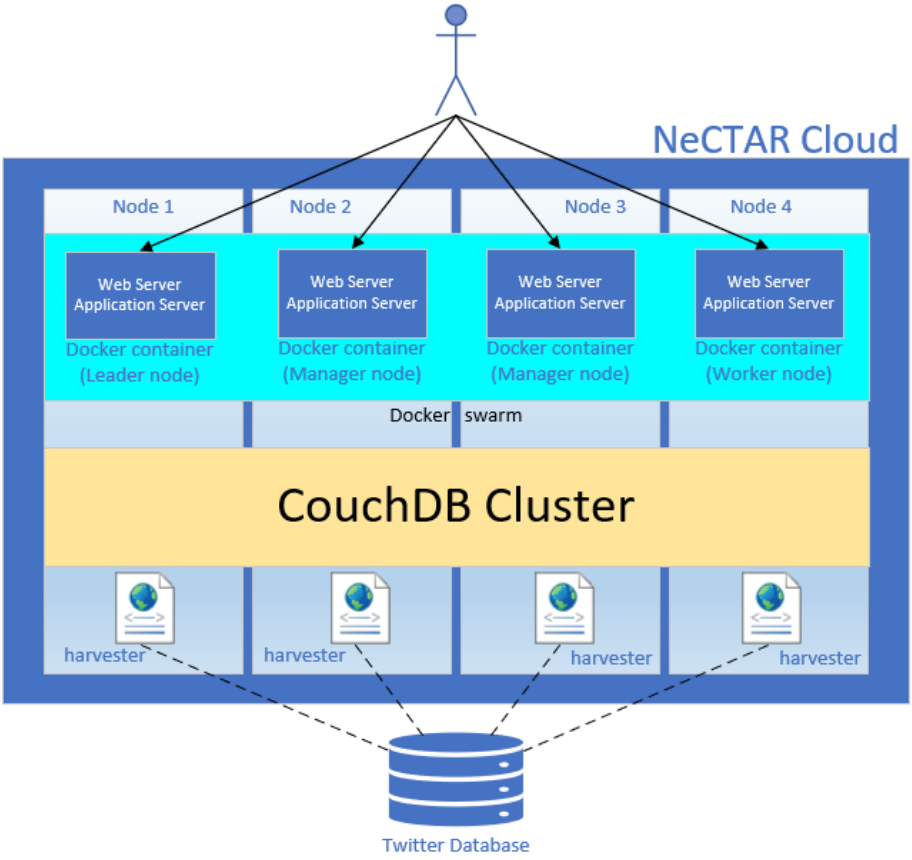


Figure 2 Detailed System Architecture

A more detailed system architecture is shown in Figure 2. We are building our system on top of the NeCTAR cloud based on OpenStack which provides Infrastructure As A Service (IAAS) allowing us to configure as low level as the operating system while it provides the storage, network, servers as well as virtualization. 4 instances were allocated to us for this project and we utilize all four instances to provide redundancy to our system which improves failure tolerance. Each instance has a docker container which runs the web+app server and the four instances forms a docker swarm. This is explained further in the ***Web and App Server Containerization with Docker*** section. The server inside the docker container pulls the data from the CouchDB cluster which gets its data from Twitter via a harvester script. In the next few sections. We will first describe each layer in Figure 2 in detail. Along the way, we will describe how we achieve the non-functional requirements.

Pros and Cons of using NeCTAR

Pros

- NeCTAR is excellent platform for cloud systems efficiently and effectively. OpenStack API is able which allows the deployment of instances easy with ANSIBLE.
- If more resources are required for an application, one can request for it and get it if the request is granted by the admin.
- The speed of launching instances is quite fast as compared to other clouds.
- It is available to research community, so, commercial usage is limited. But, it can handle different loads, and accessibility, and flawlessly help in running multiple instances.
- The security group provides the functionality for filtering network traffic. Two security rules are applied in default to allow outbound access from users' instance to the internet.

Cons

- Ubuntu 18.04 or later version has removed CouchDB from its default software repository. That means we cannot directly use apt-get command to install CouchDB. To solve this problem, we must add both CouchDB key and repository into ubuntu system [3].
- Another drawback of using nectar cloud is it requires internet to access my resource. If the loss of internet happens, users are not able to manager their data.
- In term of privacy, users are facing the problem of indirect access control, which means a third party like the ISPs and telecom company control user's internet access, and they possess the authority to entry user's sensitive documents and files without authentication.
- One concern we have encountered is the possible service outages. When certain problems occur at the data cloud center, our servers and resources may be lost in a heartbeat.

Data Collector and Processor

Crawler for tweets

The data collection is designed by referring Twitter API which provides both standard Search API and Stream API for harvesting tweets in past 7 days. However, standard Search API keeps a 15 minutes access time limit and Stream API is also restricted to one connection each time with one developer access tokens which limits our efforts in getting enough data. Hence, we created a hybrid crawler leveraging both search API and stream API for fast harvesting without touching the access limit.
Firstly, we created a geo-location filter box by getting marginal coordinates from Klokantech. We use eight squares Figure 3 to crop out the area we are interested in so, there might be some mis-crops at the edge as city areas are not squares. Since our studying granularity is on city level instead of suburbs, a few mis-crops on the edge are statistically tolerable. Secondly, some tweets filtered from stream API may not contain precise coordination as point longitude/latitude but only contain the city name and a bounding box. In our study, we will not leverage the precise coordinates of each tweets and only focus on the area of the city they came from.

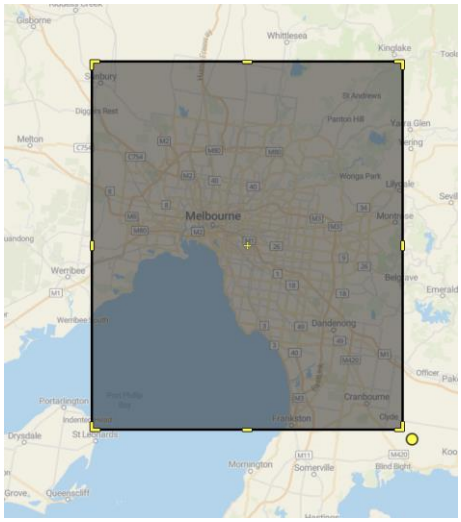


Figure 3 Bounding box for geo-filtering

We also implemented a suspend/wake mechanism for search API to cease querying after hitting rate limit and restart after a given time. In the twitter harvester, we also implemented an embedded sentiment analyzer, topic parser and time point partitioning and use a combination of the three to determine which sin a tweet belongs to.

Sentiment Analyzer

For getting the data for the sins like wrath where we must do sentiment analysis. We get this data by analyzing tweets and classify them into three classes, positive, negative and neutral. We implemented a machine learning method by leveraging TextBlob [4]. TextBlob can score a

sequence of texture symbols including English alphabets and some emojis ranging from -1 to 1 where -1 is negative and 1 refer to positive. We also implemented a preprocessor to our system and tried different combination of pre-process methods which includes remove-stop-words, lower alphabets, lemmatization and many more. We also created a text clean function which fixes all the tweet lingo for example “thats” is transformed to that is. To get the tweets for different scenarios for example assault, overweight, tweets may contain some special topics that we are interested in. We implemented a basic pattern match topic tagger. We constructed different extendable topic glossaries and make sure that they don't overlap with each other. We created topics related to our scenarios including sedentary, overweight and assault. For tweets that exclude our defined topics will be tagged with bad tweet topic.

Time Point Partition

Time point of tweet stand for the point that one typical tweet was sent online is extracted and assigned with timestamp defined by us. We partition 24 hours into five time slot named such as, 12:00am to 03:59am (midnight), 04:00 am to 07:59am (early morning), 08:00am to 12:59pm(morning), 01:00pm to 07:59pm(afternoon), 08:00 to 11:59pm(evening)

JSON Reader

We have also collected the data from Instagram, and the data is collected from university server (Figure 4) using curl. Since we are not using any API, we have made a JSON reader which reads the Instagram data and use the same sentiment analyzer and topic parsing is used for twitter tweets. Storing the data of twitter and Instagram is in same format so that we don't face any difficulty while doing the analysis.

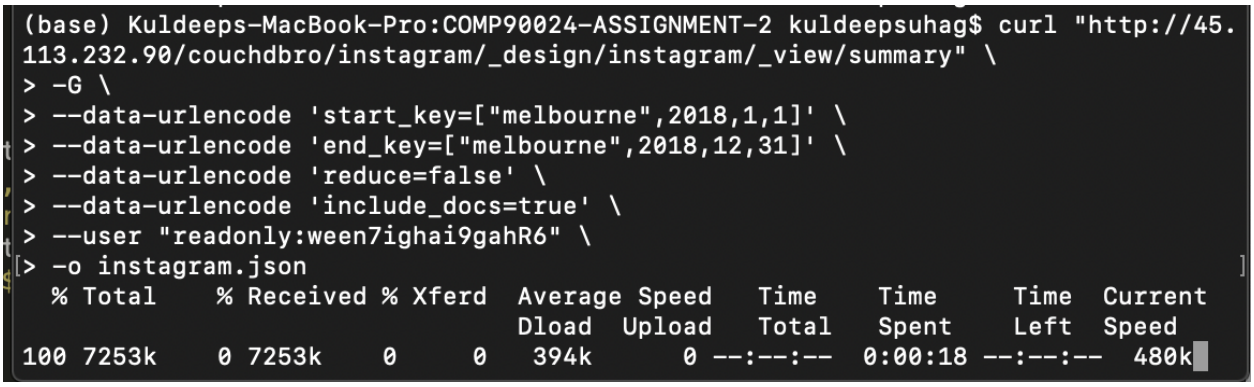


Figure 4 Collecting Instagram data from University Server

CouchDB

CouchDB is one of a new breed of database management system called NoSQL. Specifically, CouchDB is a document -oriented database where data is stored in JSON format and each document field is stored in key-value pair, map or list. Some of the key features of CouchDB are:

HTTP-based RESTful APIs

CouchDB is easily accessed via port 5984 once installed. We were able to access the database and perform CRUD operations on data using simple HTTP requests. In this project, we load the JSON through the curl -X put command and access the view from curl -X get command.

Futon

CouchDB provides its users with a web-based administration console called Futon. It gives us a clear visualization of data and design views. Futon presents the JSON document in a more readable format, so humans can easily understand the underlying structure. Another important advantage of futon is it allows us to create temporary view in the database which helps us to test and debug our Map Reduce function without wasting space for storing it.

Document Oriented storage

CouchDB uses schema free JSON documents for storing data, this means the documents are stored like real-world document, which are self-contained without any data model structure. This makes the designing more scalable and flexible as we can store the document directly rather than dividing and inserting into separate table.

MapReduce

MapReduce in CouchDB is used to create views on the database. In Map (Figure 5) function we managed to filter and sort each document in database and in Reduce (Figure 5) function we can summarize that.

```
function (doc) {
  if (doc.coordinates != null && doc.place.name == "Melbourne" && doc.topic == "gluttony" )
  {emit(doc._id, {place: doc.place.name, coordinates: doc.coordinates, label: doc.sentiment.label, topic: doc.topic, rev:
}
```

Figure 5 CouchDB map function

This technique is powerful and efficient as it processes large amount of data in less time. Although it takes considerable time at first to build B-tree for the view, but once views are created the query time is less. Another advantage of having B-tree structure is that it is scalable, when new data is added it will perform the required calculation on the new data rather than building the entire tree unless the MapReduce functions remain same. However, we have not included the polygon information in our raw data, so we cannot use reduce function to generate the tweets count in different polygons. More details will be discussed in the Problem section.

Duplication prevention

In our system, three data processors in each slave work in parallel and save processed tweets into uniform CouchDB running on database instance. We leverage the automatic document duplication prevention mechanism in CouchDB to help us ignore harvesting redundant tweets. Each tweet was given unique id by twitter, and each document in CouchDB is given a unique id.

Therefore, we use tweet ID as document ID and if there is duplication except from database, we will discard the tweet.

CouchDB Structure

Both CouchDB and twitter API expects the data to be in JSON format which is the standard format in web data interchanges.

5984	CouchDB
5986	CouchDB (node-local API)
4369	Erlang main port
9100-9200	Ports open for communication between nodes

Table 1 Ports used for communication in cluster configurations

To verify the connection between two instances we have used erlang language.

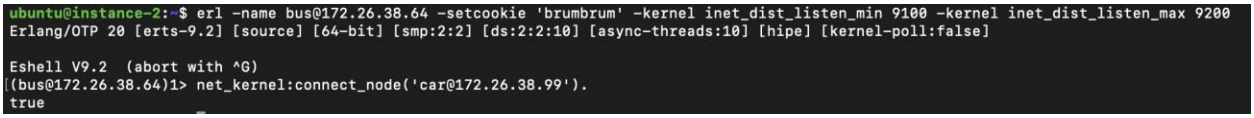


Figure 6 The erlang connection is made using the ports range 9100-9200

CouchDB's _id and _rev tags are top level tags provided by CouchDB. In our case, _id has been manually adjusted buy the harvester to mirror that of the tweet id, specifically id_str. This technique is used to ease the handling of duplicate tweets in the database. It offloads any duplicate checking directly to CouchDB, if the _id that exist is attempted to be the CouchDB will return reflect error.

Additionally, our twitter harvester provides sentiment, and timestamp tags to each document. The sentiment tag can have three possible values, either it can be between (-1,0] or (0) or between [0,1). This value either relates to positive (>0), neutral (=0) or negative(<0).

The timestamp captures the tweets for a time. The time is divided into five different period: morning, afternoon, evening, midnight, early morning.

```
{
  "_id": "00211b22af75bba1f9d7fa3273000039",
  "_rev": "1-f629d9cd7267d146372654817613d153",
  "id_str": "1126753339704614913",
  "coordinates": null,
  "timestamp": "early morning",
  "place": {
    "place_type": "city",
    "name": "Sydney",
    "bounding_box": {
      "coordinates": [
        [
          [
            150.520929,
            -34.118347
          ],
          [
            150.520929,
            -33.578141
          ],
          [
            151.343021,
            -33.578141
          ],
          [
            151.343021,
            -34.118347
          ]
        ]
      ],
      "type": "Polygon"
    },
    "full_name": "Sydney, New South Wales",
    "country_code": "AU",
    "country": "Australia"
  },
  "user": {
    "id": 2827813633,
    "id_str": "2827813633",
    "name": "Emily DP",
    "description": "🤓"
  },
  "lang": "und",
  "text": "😍😍😍 https://t.co/CR6gwwTJ7n",
  "sentiment": {
    "polarity": "0.0",
    "subjectiviy": "0.0",
    "label": "Neutral"
  },
  "topic": "bad_tweet",
  "hashtag": []
}
```


Figure 7 document structure

To verify the cluster setup and show all the nodes, simply type in the following command:

```
ubuntu@instance-1:~$ curl admin:admin@localhost:5984/_membership
{"all_nodes":["couchdb@172.26.37.220","couchdb@172.26.38.64","couchdb@172.26.38.68
","couchdb@172.26.38.99"],"cluster_nodes":["couchdb@172.26.37.220","couchdb@172.26
.38.64","couchdb@172.26.38.68","couchdb@172.26.38.99"]}
```

Figure 8 cluster setup

CouchDB Sharding

The CouchDB cluster provides the functionality of sharding, which synchronously divides our data into shards and distributes each copy to every node of the cluster. The sharding ensures our database possesses the ability to against the disconnection and loss of cluster node. In our

cluster setting, the number of node n = 3, and each node is assigned 8 shards, which gives us 24 shard replicas in total for a single database.

```
ubuntu@instance-1:~$ sudo curl -s 172.26.38.99:5984/tweets_view | jq .
{
  "db_name": "tweets_view",
  "purge_seq": "0-g1AAAAFseJzLYWBg4MhgTmEQTc4vTc5ISXIwNDfSMzLTMzbXMzIyy
  "update_seq": "613922-g1AAAAGEeJzLYWBg4MhgTmEQSc4vTc5ISXIwNDfSMzLTM7b
  "sizes": {
    "file": 704341697,
    "external": 724926297,
    "active": 546055537
  },
  "other": {
    "data_size": 724926297
  },
  "doc_del_count": 207,
  "doc_count": 613715,
  "disk_size": 704341697,
  "disk_format_version": 7,
  "data_size": 546055537,
  "compact_running": false,
  "cluster": {
    "q": 8,
    "n": 3,
    "w": 2,
    "r": 2
  },
  "instance_start_time": "0"
}
```

Figure 9 Sharding configuration of our system

The following figure displays how those 24 shard replicas are located on the cluster with single database endpoint:

```
ubuntu@instance-1:~$ sudo curl -s 172.26.38.99:5984/tweets_view/_shards | jq .
{
  "shards": {
    "00000000-1fffffff": [
      "couchdb@172.26.38.64",
      "couchdb@172.26.38.68",
      "couchdb@172.26.38.99"
    ],
    "20000000-3fffffff": [
      "couchdb@172.26.37.220",
      "couchdb@172.26.38.68",
      "couchdb@172.26.38.99"
    ],
    "40000000-5fffffff": [
      "couchdb@172.26.37.220",
      "couchdb@172.26.38.64",
      "couchdb@172.26.38.99"
    ],
    "60000000-7fffffff": [
      "couchdb@172.26.37.220",
      "couchdb@172.26.38.64",
      "couchdb@172.26.38.68"
    ],
    "80000000-9fffffff": [
      "couchdb@172.26.38.64",
      "couchdb@172.26.38.68",
      "couchdb@172.26.38.99"
    ],
    "a0000000-bfffffff": [
      "couchdb@172.26.37.220",
      "couchdb@172.26.38.68",
      "couchdb@172.26.38.99"
    ],
    "c0000000-dfffffff": [
      "couchdb@172.26.37.220",
      "couchdb@172.26.38.64",
      "couchdb@172.26.38.99"
    ],
    "e0000000-ffffffff": [
      "couchdb@172.26.37.220",
      "couchdb@172.26.38.64",
      "couchdb@172.26.38.68"
    ]
  }
```

Figure 10 Shards

Web and Application Server

We choose express.js as our web server, and npm to management packages. Packages we used are:

- Body-parser: tool to parse json files.
- Express: set up the main skeleton of our server.
- Jade: front-end render engine.

- Node-CouchDB: tool to read and write data from CouchDB.
- Point-in-polygon: find out whether coordinates are inside polygon or not.

Other common package names are listed in the *package.json* file and details of each package can be found in the *package-lock.json* file. Instead of connecting to CouchDB, web server will only response to client users. It will directly reads local caches and send it back to the frontend users. The whole working directory is shown as below:

```
├── app.js              // handle all restful requests and responses
├── bin                 // act as the main entrance of the program
├── ipAddress.txt       // list which holds host ip inside the container
├── node_modules        // contain all packages
├── package-lock.json   // details of packages
├── package.json        // list of packages for npm to manage
├── public              // store resources which can be accessed by users
├── routes              // handling all routing requests.
└── views               // store all html files
```

For application server, it works under the express skeleton, but it will not use any express components. Instead, it will process the data from CouchDB and store results in caches as json files. Since node.js has some limitations on asynchronized callbacks in single files, we have seperated the main function into 10 parts under public/JavaScript directory:

```
├── all.js              // allocate tweets into polygons of Melbourne
├── arson.js            // count arson tweets in each polygon.
├── assault.js          // count assault tweets in each polygon.
├── createView.js       // find coordinates of sins
├── envyCount.js        // count envy tweets in each polygon
├── gluttonyCount.js    // count gluttony tweets in each polygon
├── run.sh              // script to run all functions
├── scenario.js         // another function to allocate tweets into polygons of Melbourne
├── sentiment.js        // record coordinates of sentiment results.
├── slothCount.js       // count sloth tweets in each polygon
└── wrathCount.js       // count wrath tweets in each polygon
```

Run.sh includes scripts to run all the functions, and it will be executed every 10 minutes by crontab. The reason for using this mechanism will be clarified in the Problems section.

Web and App Server Containerization with Docker

Docker is a useful tool to package an application in its own container which is what we have done. Docker, like a VM, have its own environment it runs in but is more lightweight in comparison since it still utilizes the host's OS where as a VM will spin up a new OS on top of the host's OS.

Using docker has several other benefits. Firstly, any dependencies of our web and app server can be contained inside a docker image. Any machine with a suitable docker version can simply pull the image and initialize the server even if the image and host have different OS. This makes

our application highly portable which is one of the requirements under the Non-Functional Requirements: sectionNon-Functional Requirements:. Secondly, we initialize a docker swarm across all four instances and start our web+app server as a docker service. What this means is any manager node in the docker swarm can start the web service contained in the docker image all four instances automatically increasing failure tolerance, yet another non-functional requirement. With the swarm initialized, any time we have updated the web+app server image on docker hub repository, any manager node can perform a rolling update. The swarm also enables the web+app server to be easily scalable. Any new server instances we create on NeCTAR can simply install docker and join the swarm. Once this new instance has joined, any manager can issue a scale up command with the number of services being one more than what it was previously. The docker swarm will automatically allocate the service to the new instance.

Docker File

Docker file creates an application image from an existing image. In our docker file, the steps we take are

1. Pull the latest Ubuntu image from docker hub
2. Update and upgrade on the Ubuntu image
3. Install npm
4. Copy over the necessary web+app server files over
5. Copy over the script to start the server
6. Run the script as the ENTRYPOINT with ‘daemon off’ to ensure it runs in the background

After building the image, this image can be pushed to docker hub. This image can be used to start services in a swarm or used to perform a rolling update to the existing services.

In essence, docker provides many benefits for our application, however, everytime we update our server code, we need to perform several steps to accomplish this.
Pulling from git for the newest version of the server, building the docker file, tag the server to point to our docker hub repo, push the file, update the swarm. This can get very tedious to do after a while. This is where we used ANSIBLE for automation.

Automation with ANSIBLE

Ansible is an automation tool for Unix based systems which is used in configuration management for the deployment, maintenance and scaling of applications. It uses the yml code structure to declare the state. While deploying and maintaining our application across 4 instances might be easy, a real cloud application should be easily scalable and may include many more instances. With Ansible, all configurations can be automated. The following table shows the structure of the Ansible files we wrote.

```
├── host_vars
│   ├── nectar.yaml                 // Defined variables for Nectar
│   └── setSwarm.yml                // Defined variables for swarm
├── hosts                           // Hosts that playbook accesses
├── roles                           // Includes all roles
│   ├── buildImage                  // Build docker service image
│   ├── createService               // Create service for swarm leader
│   ├── copyFromGit                 // Copy files from github
│   ├── initFirstManager            // Create swarm leader
│   ├── joinAsManager               // Join swarm as a manager
│   ├── joinSwarm                   // Join swarm as a worker
│   ├── openstack-common            // Install dependencies to run openstack
│   ├── openstack-images            // Grab images from openstack API
│   ├── openstack-instance          // Create instances for Nectar
│   ├── openstack-security-group    // Create security groups
│   ├── openstack-volume            // Create volumes
│   ├── setDockerEnv                // Setup docker configuration
│   ├── setEnvironment              // Setup environment for instances
│   └── updateService               // Update service in the future
├── openrc.sh                       // Interact with openstack API
├── runIns.sh                       //Launch instances
├── installEnv.sh                   // Install software
├── runDocker.sh                    // Deploy swarm
├── configure_cluster.sh            // Set configurations for CouchDB
├── install_CouchDB.sh              // Install CouchDB
├── nectar.yaml                     // The playbook for setup instances
├── setEnv.yaml                     // The playbook for setup environment
├── setSwarm.yml                    // The playbook for setup swarm
└── updateSer.yml                   // The playbook for update service
```

Above playbooks can achieve three functionalities: launching instances, installing software and deploying swarm.

The inventory file

The inventory file named ‘hosts’ contains groups of IP addresses that we are going to connect. It contains four functional groups which we are going to use: a ‘databases’ group containing all the four instances’ IP addresses; a ‘leader’ group containing the swarm leaders’ IP address; a ‘manager’ group with swarm managers’ IP addresses and a ‘worker’ group with swarm worker’s IP address.

Playbooks

1. Launching instances

The ‘nectar.yaml’ file is used to launch instances on NeCTAR.
ANSIBLE allows us to deploy cloud services through our local machines by interacting with OpenStack API. To achieve this, a file needs to be downloaded from the dashboard called ‘openrc.sh’ and this file needs to be in the same directory while executing the playbook.
Initially, variables are defined under ‘host_var’ file repository. We specified the detailed information for each variable used in roles in ‘nectar.yaml’. For example, we defined a security group called HTTP for opening port 80 for HTTP requests.
Then the playbook will gather facts of the local machine. It can specify what system we are using and execute the command accordingly. For example, if we are using MAC, it will skip commands for Linux such as ‘sudo apt-get’ in the script.
Finally, this script will run the roles. There are five roles in the playbook. The first one is called ‘openstack-command.yaml’. This one is used to create dependencies with OpenStack API. For example, it installs OpenStack API on remote server for interacting with OpenStack. The second role is called ‘openstack-image’. This role is used to list all images for OpenStack. The last three roles are ‘openstack-volume’, ‘openstack-security-group’ and ‘openstack-instance’. Those three roles create volumes, security groups finally starts the instances according to the detail information we provided in variable files.

2. installing software

The ‘setEnv.yaml’ file is used to install software. Since we have already created instances, we can get access to those instances through their IP addresses. ‘setEnvironment’ is a role that installs all the software and their dependencies. For example, it installs pip, CouchDB and Docker among others. Finally, this playbook deploys the CouchDB cluster.

3. deploying swarm

The ‘setSwarm’ file is used to deploy swarm for the project.
In this swarm, hosts are classified as leader, manager and worker. In this playbook, we are assuming we have added the ssh key of the master instance. This allows us to pull or clone from git to get the latest version of the server code. Then, with the “ServerDocerFile” located in the git repository, Ansible builds the file as an image, logs in to our Docker hub repo and do a push. Then we initialize a Swarm’s first manager who is also the leader by calling the role ‘initFirstManager’. Tokens for manager and workers are stored. Then managers and workers can join the swarm by using ‘joinSwarm’ and ‘joinAsManager’. Finally, the Ansible script creates the service through leader with the image that we just pushed earlier and the services will start on all instances.

4. Updating service

The ‘update.sh’ is used to update service for each docker node. This one is preserved for the future. There are three steps to update service. The first and second steps are same as the create service task. It needs to build image locally and push to docker hub. After that it will update services.

Advantages of ansible

There are three reasons why we choose Ansible.

1. Automation

Manually setup cloud system might be complex and hard to maintain. In our system, we need to deploy multiple configurations. For example, we need to manage the configuration for both the cloud instance and the docker container. Manually managing those configurations on every instance might cause human errors since there are many details to consider.

We can automatically complete these tasks by using Ansible. In this case, all the instances can be built with the same recipe, which guarantees the consistency of the created instances.

2. Efficiency

Ansible enables us to deploy the cloud environment efficiently. If we want to update the server, we can simply run the Ansible scripts. Rather than check the state of each server and upgrade them to the latest if it is not, we can run the same script on all the servers. In that way, Ansible can provide efficiency for us.

3. Agentless

Ansible is an agentless tool and no additional software is required. We can use SSH to execute commands and connect to a remote server. This agentless architecture enables us to manage servers easily.

Problems with Ansible

Although ansible has many advantages, we found some problem while using it.

1. Bad indentation

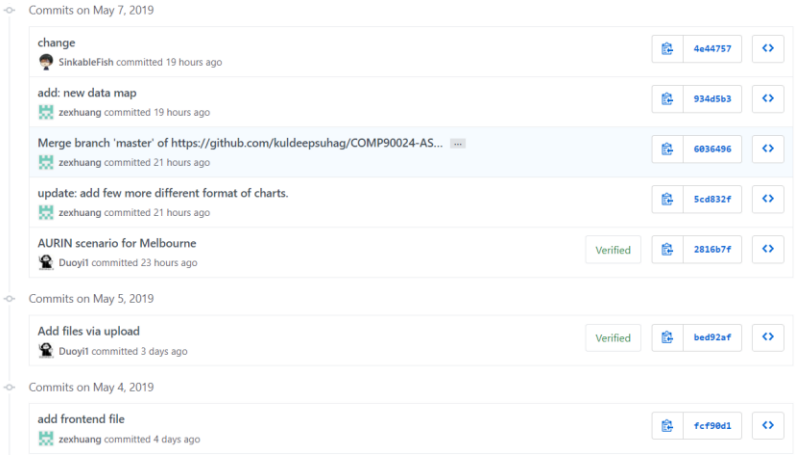
YAML files have a relatively strict requirement on indentation comparing with other scripting languages making it hard to debug at times.

2. OS restriction

Ansible is not user-friendly for Windows users. Users have to install Linux on the system and use it.

Version Control: GIT

Our team used GIT as our version control system and GitHub to store it remotely. It allows us to keep previous versions of our code in case a change breaks something, at which point we can revert to an earlier version. The other benefit is it allows us to develop the system in parallel while also allowing us to view the others' progress.



Problems

In this section, we will outline some of the difficulties we faced during this project.

- Network unavailable in instances
  After setting up instances, we found there are no Internet for whole four instances. The reason for this is that proxy is required to visit Internet under university availability zone. But another problem has been raised due to the university proxy.

- Fail to clone GitHub repositories through ssh link
  Since our GitHub repository is a private one, once someone wants to clone the it, username and password will be required for authentication. But it is dangerous to write username and password in ansible files. Instead we tried to use ssh link to clone the git repository. However, the ssh link of GitHub is unreachable due to the proxy we added before. This problem confused us for a long time. We tried many ways to solve this problem including changing proxy, rebooting network manager etc. At last, we found another student who has the same problem, and added GitHub configuration following the guidance. The problem is solved in some aspects. Now, after adding ssh key of instance in the GitHub, we can clone private GitHub repository through ssh link automatically by using ansible.

- Fail to connect database outside docker containers
  At first, we created our CouchDB's and run our servers on the instances directly. Everything worked fine at that time. However, when we set up docker containers and put our servers into the containers. The application server failed to connect the CouchDB outside the container. To solve this problem, we first decided to add *--network=host* parameter to bind container network with host while creating the container, but later we found 127.0.0.1 inside the container did not point to the host ip address and whole swarm could not work normally. At last, we chose to use a file to store the host ip address. Before launching the server, docker will first detect the host ip outside the

container and store it in ipAddress.txt inside the container. Then server will connect to host database according to the ip list in ipAddress.txt.

- Asynchronized functions timeout
  We did not find out which polygon does the tweet belong to in our raw data. At first, we thought the polygons in Melbourne are squares just like the first assignment. But, later we found a more specific polygon definition in AURIN and the shapes of polygon are various. There is no way to allocate tweets in CouchDB views. So, we write other functions to calculate different scenarios in different polygons. However, these functions are asynchronized and some of them might execute for a long time (we chose 30 polygons in Melbourne and each one has hundreds of coordinates in borders. Over 400k tweets need to be compared with all these polygon data). These long-time functions will activate the timeout exception since node.js has some limits on multiple asynchronized functions in a single program.

Figure 11 timeout for node.js codes

As a result, we cannot merge these processes into the web server part. To solve this problem, we separated these functions into independent ones and wrote a script to run all of them. The script will be executed every ten minutes and web server can work normally even these functions all crash down. Also, these functions will generate results and store them in the cache folder, which can accelerate page loading speed.

Deployment guide

- See: https://github.com/kuldeepsuhag/COMP90024-ASSIGNMENT-2 readme.

Result and Discussion - Data

We collected data related to assault, sedentary and obesity from AURIN.
Assault data [5] reflects assault and related offence count in different LGA areas.
Since there is a dramatic different of population between different LGA areas, the number of assaults is varied accordingly. For example, Melbourne has the largest number of people in Victoria and there are more assault accounts in Melbourne. For that reason, we need to normalize data for Assault as well as the Sedentary and Obesity data. The way we normalize it is divided the assault counts by the number of people in that area.
Sedentary data is also normalized and it [6] reflects the percentage of people who sit for 7 hours or more per day.

Obesity statistic [5] means adult who were overweight or obese in each LGA area. This figure is calculated by TUA_PHIDU based on Age-standardized rate.

As mentioned in earlier section “Data Collector and Process”, our tweets is composed by the following categories:

- Gluttony: this category reflects human dietary issues, and the kind of tweets are generated by the topic parser using related keywords, such as “foodaddict”, “delicious” or “bigmac”.
- Sloth: this category contains tweets that show laziness and sleeping habit of human. A Time Frame Partition algorithm is used to discover tweets that post by uses during the midnight or early morning. Also, keywords like “couch potato”, “lazybones” or “slacker” are applied by the crawler to detect related content in tweets.
- Wrath: This kind of tweets is returned by the sentiment analyzer with “Negative” sentiment and the relevant tokens like “hate”, “rage”, or “mad”.

Similar to the AURIN data, there will be more tweets in Melbourne CBD than other areas due to the population difference. Therefore, we also normalized the Twitter data by dividing the number of Tweets related to a particular sin by the total number of Tweets in an area.
In terms of the number of tweets, we got 2000 tweets related to gluttony, 170k tweets for sloth and 40k for wrath.

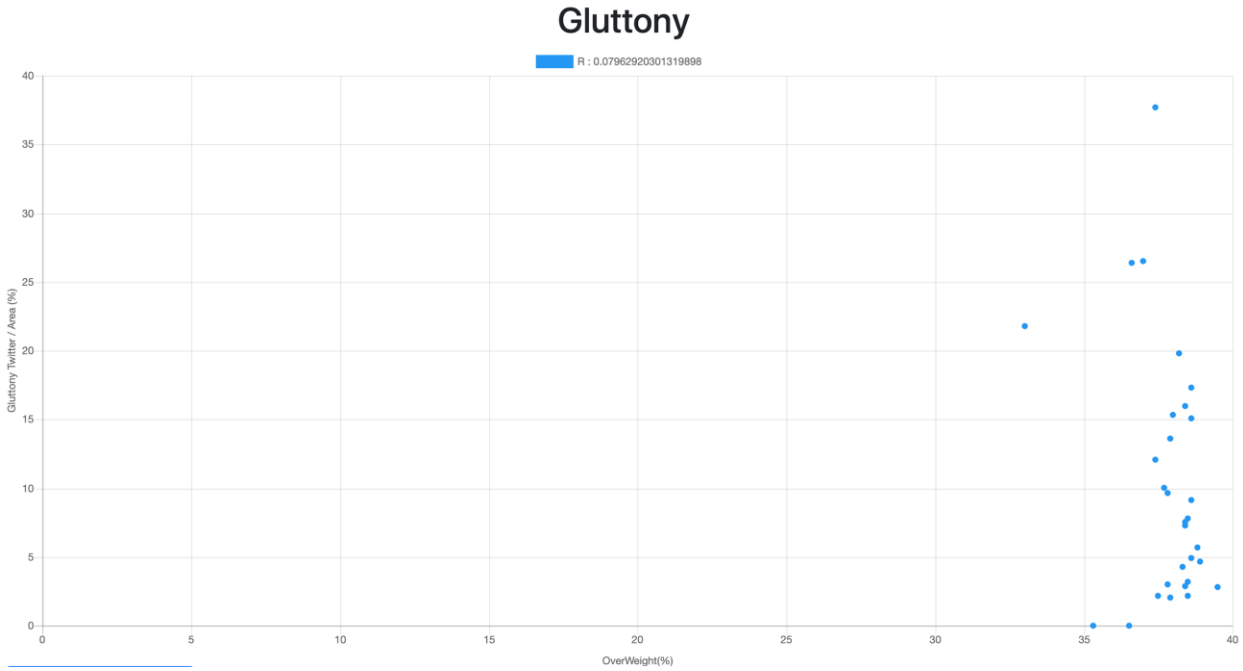


Figure 12 Obesity percentage vs Gluttony tweet percentage

Each point in this graph represents a region in the Greater Melbourne District. The chart above demonstrates no significant correlation between obesity and gluttony tweets. The number of gluttony tweets in areas varies a lot while the obesity % in areas stays constant. Unfortunately, this data is not conclusive, with one of the major reason is that the total amount of gluttony tweets collected is only at around 2000 and is not sufficient.

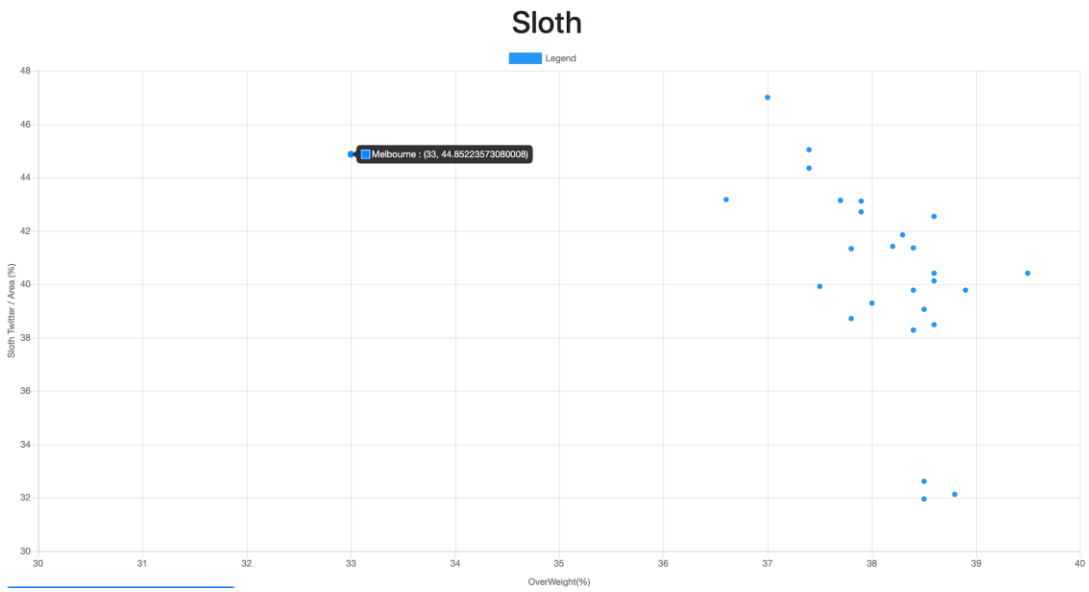


Figure 13 Obesity percentage vs Sloth tweet percentage

This graph explores the relation of sloth tweets on social media and people's actual overweight situation. Unlike the assumption that we make before, our result indicates a slight inverse ratio on these two parameters. Another intriguing interpretation of this graph is people living in area like Melbourne CBD have the lowest overweight percentage among other areas. Our guess is citizens in central area have much more opportunities to access GYM, local parks, and recreation center facilities than people living in suburban areas.

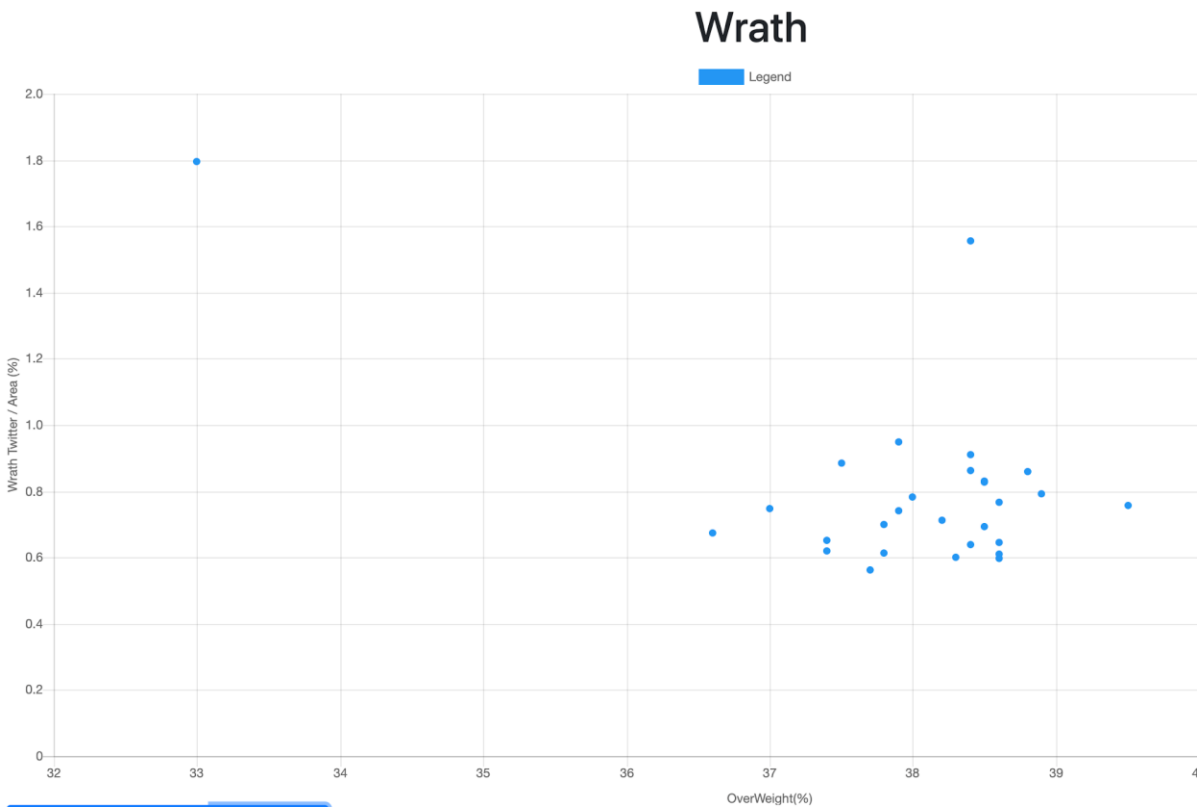


Figure 14 Obesity percentage vs Wrath tweet percentage

We further identifying whether areas with more overweight people posts more tweets related to wrath. The figure shows no clear correlation between these two data sets. The outlier to the top left represents Melbourne CBD and a lot more tweets related to wrath appear in this area.

Conclusion

In conclusion, we build a system to collect social media data related to wrath, sloth and gluttony to compare with obesity to determine whether gluttony is the cause of obesity and whether obesity is the cause of wrath and sloth.

While the amount of data we are currently gathering is still relatively small and cannot be considered to be in the big data realm yet, if the harvester keeps on running over time, the system must be able to scale with it. With technologies such as docker for containerization and portability, ANSIBLE for automation and CouchDB for big data storage capabilities and analytics(MapReduce), our system is able to do exactly that.

Of course, the system is not perfect and there are many improvements that can be made. For example, we can use docker-compose to set up both our web+app server and the database making deployment even easier. Also, despite the amount of automation that we have written for our system with Ansible, there are small parts that still needs manual copy pasting such as when we have to copy the IP addresses. Finally, we are not using a volume to store the

database data and using the storage space inside the instances instead. While we do have duplication across the cluster and deleting one instance might not matter, deleting all instances will result in a loss of data.

As for the hypotheses, we were unable to prove that the cause of obesity is gluttony as we had too little data for this sin. For wrath and gluttony related tweets, we were also unable to determine any correlation just by comparing obesity with the tweet counts.

References

[1] A. B. o. Statistics, »National Health Survey: First Results, 2017-18,« Australian Bureau of Statistics, 2017. [Online]. Available: https://www.abs.gov.au/ausstats/abs@.nsf/Lookup/by%20Subject/4364.0.55.001~2017-18~Main%20Features~Key%20Findings~1. [Senest hentet eller vist den 2019].

[2] J. Feldman, »Sophie Turner Opens Up About Depression And Social Media Scrutiny,« Huffingtonpost, [Online]. Available: https://www.huffingtonpost.com.au/2019/04/17/sophie-turner-opens-up-about-depression-and-social-media-scrutiny_a_23713459/.

[3] Linuxize, »How to Install CouchDB on Ubuntu 18.04,« Linuxize, 2019. [Online]. Available: https://linuxize.com/post/how-to-install-couchdb-on-ubuntu-18-04/.

[4] S. Loria, »TextBlob: Simplified Text Processing,« TextBlob, 2018. [Online]. Available: https://textblob.readthedocs.io/en/dev/.

[5] T. U. Australia, »Adults Health Risk Factor Estimates,« AURIN, Australia, 2014-2015.

[6] G. o. V. -. VicHealth, »Sedentary behaviour (sitting hours per day),« AURIN, Australia, 2014.

Appendix

Github: https://github.com/kuldeepsuhag/COMP90024-ASSIGNMENT-2
YouTube demo: https://www.youtube.com/watch?v=eeytV0PaPqc&feature=youtu.be