CHAPTER 1

INTRODUCTION

1.1 CONCEPT

Twitter is a "micro-blogging" social networking website that has a large and rapidly growing user base. Those who use Twitter can write short 140 characters long or less updates called 'tweets'. 'Tweets' are seen by those who 'follow' the person who 'tweeted'. Due to the growing popularity of the website, Twitter can provide a rich bank of data in the form of harvested "tweets".

Twitter by its very nature, allows people to convey their opinions and thoughts openly about whatever topic, discussion point or product that they are interested in sharing their opinions about. Therefore Twitter is a good medium to search for potentially interesting trends regarding prominent topics in the news or popular culture.

Sentiment analysis (or opinion mining) refers to the use of natural language processing, text analysis and computational linguistics to identify and extract subjective information in source material. The value of Twitter in recent years has increased as businesses, political groups and curious Internet users alike have started to assess the public's general sentiment for their products and services from twitter posts. Sentiment analysis provides a means of tracking opinions and attitudes on the web and determines if they are positively or negatively received by the public.

The purpose of Text mining is to process unstructured (textual) information and to extract meaningful numeric indices from the text, allowing the application of various data mining algorithms to explain the textual dataset.Text analysis involves information retrieval, lexical analysis to study word frequency distributions, pattern recognition, tagging/annotation, information extraction, data mining techniques including link and association analysis, visualization, and predictive analytics. The overarching goal is, essentially, to turn text into data for analysis, via application of natural language processing (NLP) and analytical methods.Text mining methods and software is also being researched and developed by major

firms, including IBM and Microsoft, to further automate the mining and analysis processes, and by different firms working in the area of search and indexing in general as a way to improve their results.

The classification model which this project will develop will determine whether the tweet status updates (which cannot exceed 140 characters) reflects positive opinion or negative opinion on the behalf of the person who tweeted. This project will use a hybrid of knowledge based sentiment analysis methodologies which have been more traditionally used, and those of machine learning methodologies which used a more intuitive approach to sentiment. The results of these two methodologies will be used to perform a thorough analysis of the dataset.

1.2 AIM OF THE PROECT

In order to conduct any kind of analysis on twitter the construction of a suitable dataset of tweets needs to be built. Twitter API is an app which extracts tweets from twitter and loads them into a dataset. The Twitter API is called the twitter4j API. It provides access to lakhs of tweets to users who register themselves as developers on the twitter website.

To ensure privacy and access only to authenticated users, twitter generates credentials in the form of four keys which are used to perform any connection with the twitter server which is called the twitter hose. These tweets enter the analysis tool i.e Apache Storm by means of a spout which is like a tap. This spout provides a single source of tweets for analysis as required. Later, this data is passed through a series of bolts which are used to perform analysis in a step by step manner.

This analysis involves the process of filtering and removing useless information which will not affect the sentiment of any topic. Some examples of these filtered words are *the*, *an*, *according* etc. The absence of these words does not affect the intended meaning in any way. These filtered words are removed by the process of parsing the tweets. This is done using a separate bolt.

The parsed tweets are then sent to another bolt which is used to calculate the ranking of the sent tweet in the form of number of re-tweets. This helps determine the popularity of the

tweet and helps us understand whether a topic is trending or not. This is also done using a separate bolt called the Count Bolt.

The output of this bolt is given to another bolt which is responsible for displaying the top trending hashtags using the webpage run using a python micro-server called flask. This webpage is locally hosted and enables a naïve person to visualize the trending topics in a real time scenario.

The separate module is designed to help us understand the sentiment related to a particular trending topic as per user discretion using the Naïve Bayes Classifier. This classifier classifies tweets into the categories of positive, negative and neutral thereby helping the user understand the sentiment of the people on twitter about that topic.

CHAPTER 2

LITERATURE SURVEY

2.1 EXISTING SYSTEM

Sentiment analysis has been handled as a Natural Language Processing task at many levels of granularity. Starting from being a document level classification task, it has been handled at the sentence level and more recently at the phrase level. The level of granularity has varied from one paper to another. Each of the levels have their own advantages and disadvantages. However, the level which closely mirrors the real time result is considered t be the most relevant.

Micro blog data like Twitter, on which users post real time reactions to and opinions about “everything”, poses newer and different challenges. Some of the early and recent results on sentiment analysis of Twitter data are by Goetal. Goetal. (2009) use distant learning to acquire sentiment data. They use tweets ending in positive emoticons like “:)” “:-)” as positive and negative emoticons like “:(” “:-(” as negative.

They build models using Naive Bayes, MaxEnt and Support Vector Machines (SVM), and they report SVM outperforms other classifiers. In terms of feature space, they try a Unigram, Bigram model in conjunction with parts-of-speech (POS) features. They note that the unigram model outperforms all other models. Specifically, bigrams and POS features do not help.

Pak and Paroubek (2010) collect data following a similar distant learning paradigm. They perform a different classification task though: subjective versus objective. For subjective data they collect the tweets ending with emoticons in the same manner as Go et al. (2009). For objective data they crawl twitter accounts of popular newspapers like “New York Times”, “Washington Posts” etc. They report that POS and bigrams both help (contrary to results presented by Go et al. (2009)).

Both these approaches, however, are primarily based on n-gram models. Moreover, the data they use for training and testing is collected by search queries and is therefore biased. This is because search queries are designed by humans, most of who are prone to preconceived notions and also suffer from errors like hindsight's bias. These models which involve human decision making are therefore considered to be polluted by most eminent people in this field of study.

2.2 PROPOSED SYSTEM

The existing systems suffer from errors because most of them are Human dependent in their execution and analysis. This project is a step against that. This project merely aims to analyze data and spot the popular trending hashtags from the tweets independently without any human involvement in any aspect. We are proposing to use Apache Storm to implement the functionality. We have chosen Apache Storm since Storm is considered to be the best tool to analyze real time streaming tweets. The other alternatives like Hadoop are not considered suitable for the purpose of streaming analytics.

The tweets are proposed to be sent to the Apache Storm topology consisting of Spouts and Bolts using the Twitter Garden Hose. The Twitter garden Hose in this case is Twittter4j API. The tweets enter the topology using the credentials issued by twitter for the developers. Later, the tweets are parsed by the proposed bolt and only the useful information is made available for further analysis.

The filtered tweets are sent to another bolt which basically counts the number of occurrences of the said word. It then sends the word and its count to another bolt. This bolt ranks the words in terms of the count to determine the trending topics. These trending topics are stored separately and they get updated dynamically as the users tweet in the real time environment.

The trending words are then sent to the last bolt which issues a call to the Reddis library. The reddis library is used to display data using servers. In this proposed system, we are using the Flask Library to host a micro-server and a html file is written to display the trending word count using the D3.js library in the form of a word cloud.

The last and final module is related to the sentiment analysis as per the user requirements. The module is developed to connect to twitter to find information relevant to a particular trending topic. All these tweets are mined using the Beautiful Soup library and they are analyzed using the Naïve Bayes classifier. The naïve bayes classifier uses the probabilistic methods to determine the posterior probabilities. These probabilities are converted to percentages to represent the public opinion about a trending topic. The opinion is generally classified into positive, negative and neutral as per the word usage. The naïve bayes classifier is used because of the higher accuracy rates in determining the public perception as per the testing results.

CHAPTER-3
ANALYSIS AND DESIGN

3.1 PROPOSED SYSTEM

The existing systems suffer from many biases as discussed earlier. The proposed system is illustrated below. The proposed system is a real time analytic system so it offers many additional advantages. The proposed system does not require any database to store the data since analytics to understand current opinion does not require any historical prerogative.

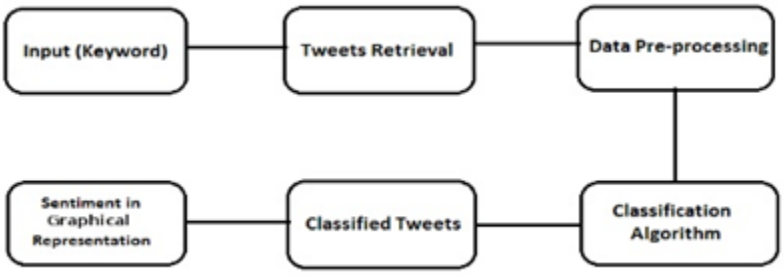

Fig1 Block Diagram

The block diagram clearly displays the various stages of the proposed system. The input is in the form of the twitter garden hose i.e Twitter4j API. The retrieved tweets are processed and filtered to remove unwanted words. This filtering is done using the concept of parsing. The parsed tweets are then counted and the top trending hashtags are displayed using a web application. The web application is hosted on a Flask micro-server and it is rendered using the Reddis library.

The visualization is accomplished using the HTML, CSS and a java script library called D3. The visualization is displayed using a beautiful word cloud which renders the size as per the number of tweets across the world.

3.2 SCOPE OF THE PROJECT

This project is an extremely useful tool which can be used to visualize current trending topics as well as retrieve information about them. This project also helps the user get the opinion of the people about a particular topic to understand their opinion. The main advantage of this project lies in its ability to be flexible in terms of the number of people surveyed. There is no limit on the number of people and opinions to be considered, since it's a real time application. This project also displays the results to naïve users in an easy to understand and interpret format. It can hence be considered a standalone system for all categories of users.

3.3 GIT and GITHUB

Git is a distributed revision control and source code management system with an emphasis on speed. It was initially designed and developed by Linus Torvalds for Linux kernel development. It is a free software distributed under the terms of the GNU General Public License version 2.Version Control System (VCS) is a software that helps software developers to work together and maintain a complete history of their work.

Git allows developers to work simultaneously without overwriting each other's changes. This is accomplished using the concepts of branching where there are master and slave branches. It also maintains a history of every version and hence we can always go back to a previous version if required. This ensures that the developer doesn't have to worry about maintaining and saving multiple versions of the same project locally. There are two types of version control systems namely Centralized version control systems (CVCS) and Distributed/Decentralized version control system (DVCS).

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of

the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the server goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

Git offers many advantages because of its open source platform.Git is released under GPL's open source license. It is available freely over the internet. You can use Git to manage property projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements. It is surprisingly very fast and extremely small. It is extremely small because of the local nature of the operations. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation. The core part of Git is written in C, which avoids runtime overheads associated with other high-level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

Git offers an implicit backup so the chances of losing data are negligible because of the availability of multiple versions of the project. Data present in the client machine is the exact copy of the data present on the server. So, in case of a crash, the client data can be safely ported to the server. It is extremely secure because of the usage of cryptographic hash function called secure hash function or SHA1. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that, it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git. Git is useful because it can be run on machines with very simple hardware requirements. This is not the case of CVCS where the central server requires extremely powerful server to enable collaboration by the team. This is a performance bottleneck when larger teams are

involved. In case of DVCS, developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.Git also allows easier branch management which isn't the case with CVCS where this effort is time consuming and inefficient.

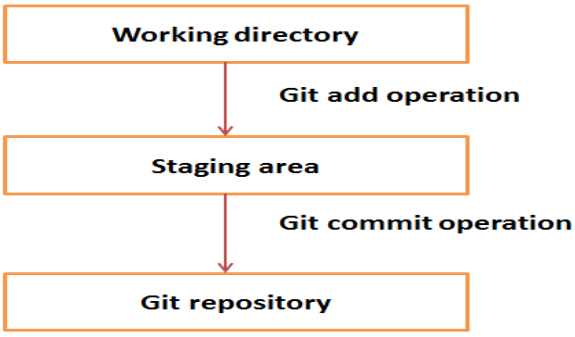

Fig2 Git Operations

The general workflow involved in Git begins with the cloning of the Git Repository as a working copy. We then modify or edit the files as per the requirements of the developer. This process may also involve collaboration with other developers. We may also involve in code reviews before committing. This is followed by committing, where the changes create a new commit locally. This commit must be changed on the central server as well. This is accomplished by means of pushing the changes to the repository. These last two steps are repeated as and when the developer wants to incorporate changes and mistakes. This is represented in the illustration below.

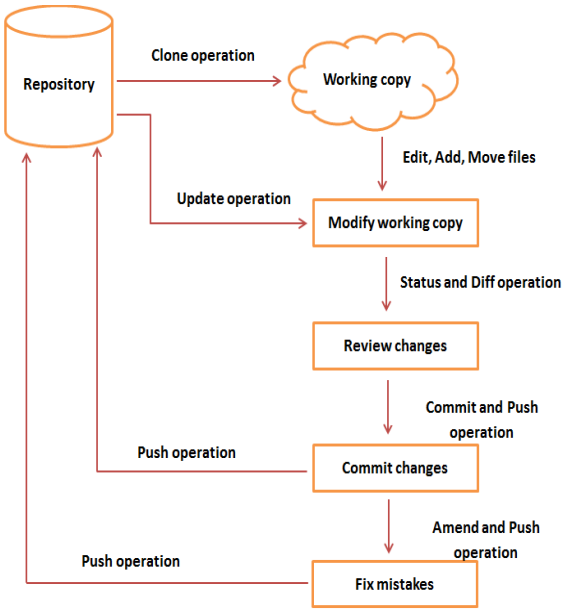

Fig3 Github Workflow

A Git repository contains the history of a collection of files starting from a certain directory. The process of copying an existing Git repository via the Git tooling is called cloning. After cloning a repository the user has the complete repository with its history on his local machine. Of course, Git also supports the creation of new repositories. The developer can also delete a repository. But the deletion of a repository will lead to loss of entire history of the project. If you clone a Git repository, by default, Git assumes that you want to work in this repository as a user. Git also supports the creation of repositories targeting the usage on a server. The bare repositories are supposed to be used on a server for sharing changes coming from different developers. Such repositories do not allow the user to modify locally files and to create new versions for the repository based on these modifications. But non-bare repositories target the user. They allow you to create new changes through modification of files and to create new versions in the repository. This is the default type which is created if you do not specify any parameter during the clone operation.

A local non-bare Git repository is typically called local repository.A local repository provides at least one collection of files which originate from a certain version of the

repository. This collection of files is called the working tree. It corresponds to a checkout of one version of the repository with potential changes done by the user.

The user can change the files in the working tree by modifying existing files and by creating and removing files. A file in the working tree of a Git repository can have different states. The untracked state is the state of the file when it is not tracked by Git. This means that the file is yet to be staged or committed. The tracked stage is the stage when the file has been committed at least once but the file is yet to be bought to the staging area. The staged files are the files in the staged area which form the latest commit or the newest commit which will be displayed to the user when checking the commit history. The dirty stage is when the file has been modified but not staged. This is when the user has edited and saved changes but has not committed them or staged them.

After doing changes in the working tree, the user can add these changes to the Git repository or revert these changes. The modification of the working tree must be followed by either staging or committing to ensure that these changes persist locally. The staging can be completed using the git add command while the staging can be accomplished using the git commit command. This process is depicted in the following graphic below.

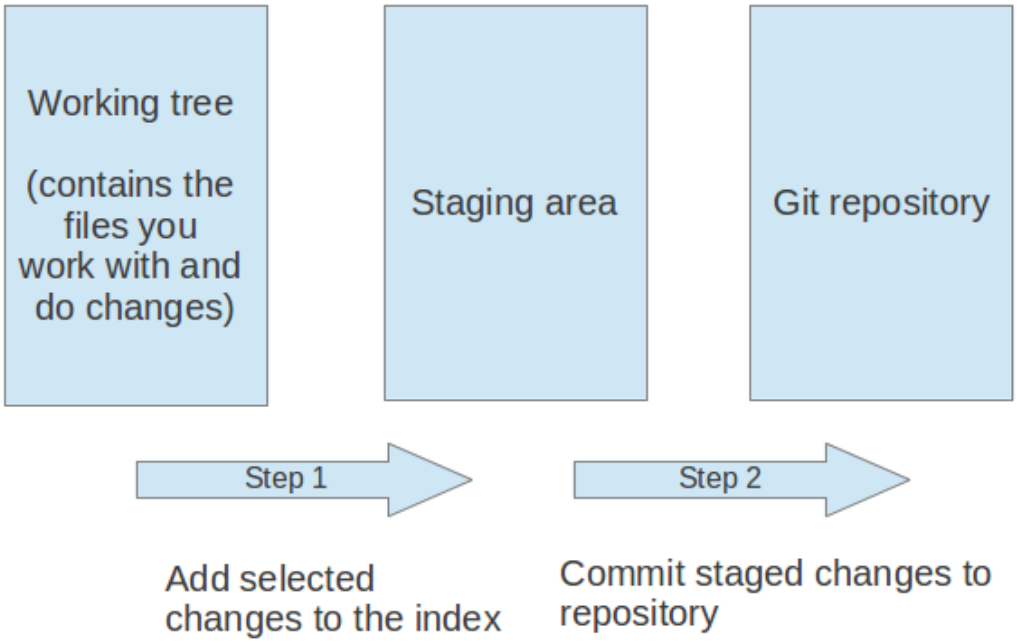

Fig4w Git Commit Operation

The `git add` command stores a snapshot of the specified files in the staging area. It allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes. Some tools and Git user prefer the usage of the index instead of staging area. Both terms mean the same thing.

After adding the selected files to the staging area, you can commit these files to add them permanently to the Git repository. Committing creates a new persistent snapshot (called commit or commit object) of the staging area in the Git repository. A commit object, like all objects in Git, is immutable.

The staging area keeps track of the snapshots of the files until the staged changes are committed. For committing the staged changes we use the `git commit` command. If you commit changes to your Git repository, you create a new commit object in the Git repository.

3.4 VAGRANT

Vagrant is a very important tool which helps manage the workflow while coding and testing the applications. Most developers suffer from the problems of difference in test environment and usage environment. This difference can cause applications to crash because of differences between the environment of use and development. Vagrant helps bridge these problems by defining a system for the developer which is similar to the usage environment, so the developer can incorporate changes while developing itself to avoid problems to the user. This ensures that the developer writes production ready code from the start itself and can help minimize bugs.

This process of developing production ready code involves the usage of dependencies. Vagrant will isolate dependencies and their configuration within a single disposable, consistent environment, without sacrificing any of the tools you are used to working with (editors, browsers, debuggers, etc.). Once you or someone else creates a single Vagrantfile, you just need to vagrant up and everything is installed and configured for you to work. Other members of your team create their development environments from the same configuration, so whether you are working on Linux, Mac OS X, or Windows, all your team members are running code in the same environment, against the same dependencies, all configured the same way. This ensures that there are no "works on my machine" bugs. Vagrant also gives you a disposable environment and consistent workflow for developing and testing infrastructure management scripts.

We can quickly test things like shell scripts, Chef Cookbooks, Puppet modules, and more using local virtualization such as VirtualBox or VMware. Then, with the same configuration, you can test these scripts on remote clouds such as AWS or RackSpace with the same workflow. Ditch your custom scripts to recycle EC2 instances, stop juggling SSH prompts to various machines, and start using Vagrant to bring sanity to your life. Once a developer configures Vagrant, you do not need to worry about how to get that app running ever again. No more bothering other developers to help you fix your environment so you can test designs. Just check out the code, vagrant up, and start designing.

Vagrant is also considered to be superior when compared to Docker because of lack of support for Docker from windows and BSD or Linux. Vagrant usage can be started simply by downloading Virtual Box and Vagrant from the respective websites. After the download, we can setup the machine requirements of our choice using two simple commands shown below.

```
$ vagrant init hashicorp/precise64
$ vagrant up
```

After running the above two commands, you will have a fully running virtual machine in VirtualBox running Ubuntu 12.04 LTS 64-bit. You can SSH into this machine with vagrant ssh, and when you are done playing around, you can terminate the virtual machine with vagrant destroy. The first step in configuring any Vagrant project is to create a Vagrantfile. The purpose of the Vagrantfile is twofold namely marking the root directory of your project. Many of the configuration options in Vagrant are relative to this root directory. Secondly, it is used to describe the kind of machine and resources you need to run your project, as well as what software to install and how you want to access it. Vagrant has a built-in command for initializing a directory for usage with Vagrant: vagrant init. This is illustrated below using an example directory called the vagrant_getting_started.

```
$ mkdir vagrant_getting_started
$ cd vagrant_getting_started
$ vagrant init
```

Vagrant also ensures that instead of building a virtual machine from scratch, which would be a slow and tedious process, Vagrant uses a base image to quickly clone a virtual machine. These base images are known as "boxes" in Vagrant, and specifying the box to use for your Vagrant environment is always the first step after creating a new Vagrantfile. Boxes are added to Vagrant with vagrant box add. This stores the box under a specific name so that multiple Vagrant environments can re-use it.

```
$ vagrant box add hashicorp/precise64
```

This will download the box named "hashicorp/precise64" from HashiCorp's Atlas box catalog, a place where you can find and host boxes. While it is easiest to download boxes from HashiCorp's Atlas you can also add boxes from a local file, custom URL, etc.

Boxes are globally stored for the current user. Each project uses a box as an initial image to clone from, and never modifies the actual base image. This means that if you have two projects both using the hashicorp/precise64 box we just added, adding files in one guest machine will have no effect on the other machine. In the above command, you will notice that boxes are namespaced. Boxes are broken down into two parts - the username and the box name - separated by a slash. In the example above, the username is "hashicorp", and the box is "precise64". You can also specify boxes via URLs or local file paths.

Now that the box has been added to Vagrant, we need to configure our project to use it as a base. Open the Vagrant file and change the contents to the following shown in the illustration below.

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise64"
end
```

The "hashicorp/precise64" in this case must match the name you used to add the box above. This is how Vagrant knows what box to use. If the box was not added before, Vagrant will automatically download and add the box when it is run. These steps will set up a vagrant machine on the system, which can be run using the vagrant up command as shown in the illustration below.

```
$ vagrant up
```

In less than a minute, this command will finish and you will have a virtual machine running Ubuntu. You will not actually *see* anything though, since Vagrant runs the virtual machine without a UI. To prove that it is running, you can SSH into the machine as shown in the illustration below:

```
$ vagrant ssh
```

This command will drop you into a full-fledged SSH session. Go ahead and interact with the machine and do whatever you want. Although it may be tempting, be careful about rm -rf /, since Vagrant shares a directory at /vagrant with the directory on the host containing your Vagrant file, and this can delete all those files.

The user can continue with the session till their development requirements are not fulfilled. Later, they can log out of the vagrant SSH connection using the logout command as illustrated below.

```
vagrant@precise64:~$ logout
Connection to 127.0.0.1 closed.
```

We can also run the vagrant destroy command but this will prevent any resource access to the virtual machine environment till the vagrant session is restarted again.

3.5 APACHE STORM

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. Storm is simple, can be used with any programming language, and is a lot of fun to use! Storm has many use cases: realtime analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Storm is fast: a benchmark clocked it at over a million tuples processed per second per node. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Storm integrates with the queueing and database technologies you already use. A Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed**.** In a short time, Apache Storm became a standard for distributed real-time processing system that

allows you to process large amount of data, similar to Hadoop. Apache Storm is written in Java and Clojure. It is continuing to be a leader in real-time analytics.

Storm has Master/Slave architecture with Zoo Keeper based coordination. The master node is called as nimbus and slaves are supervisors. Storm topology runs until shutdown by the user or an unexpected unrecoverable failure. If nimbus / supervisor dies, restarting makes it continue from where it stopped, hence nothing gets affected. Storm is therefore very useful and some of its advantages are listed below. It is open source, robust, and user friendly. It could be utilized in small companies as well as large corporations. It is fault tolerant, flexible, reliable, and supports any programming language. It allows real-time stream processing and is unbelievably fast because it has enormous power of processing the data. Storm can keep up the performance even under increasing load by adding resources linearly. It is highly scalable. It performs data refresh and end-to-end delivery response in seconds or minutes depends upon the problem. It has very low latency with operational intelligence. It also provides guaranteed data processing even if any of the connected nodes in the cluster die or messages are lost.

Spouts and bolts are connected together and they form a topology. Real-time application logic is specified inside Storm topology. In simple words, a topology is a directed graph where vertices are computation and edges are stream of data. A simple topology starts with spouts. Spout emits the data to one or more bolts. Bolt represents a node in the topology having the smallest processing logic and the output of a bolt can be emitted into another bolt as input.Storm keeps the topology always running, until you kill the topology. Apache Storm's main job is to run the topology and will run any number of topology at a given time.spouts and bolts. They are the smallest logical unit of the topology and a topology is built using a single spout and an array of bolts. They should be executed properly in a particular order for the topology to run successfully. The execution of each and every spout and bolt by Storm is called as "Tasks". In simple words, a task is either the execution of a spout or a bolt. At a given time, each spout and bolt can have multiple instances running in multiple separate threads. A topology runs in a distributed manner, on multiple worker nodes. Storm spreads the tasks evenly on all the worker nodes. The worker node's role is to listen for jobs and start or stop the processes whenever a new job arrives.

Stream of data flows from spouts to bolts or from one bolt to another bolt. Stream grouping controls how the tuples are routed in the topology and helps us to understand the tuples flow in the topology. There are four in-built groupings available in Storm. They are Shuffle grouping, Fields grouping, Global grouping and All grouping. In shuffle grouping, an equal number of tuples is distributed randomly across all of the workers executing the bolts. The following diagram depicts the structure.

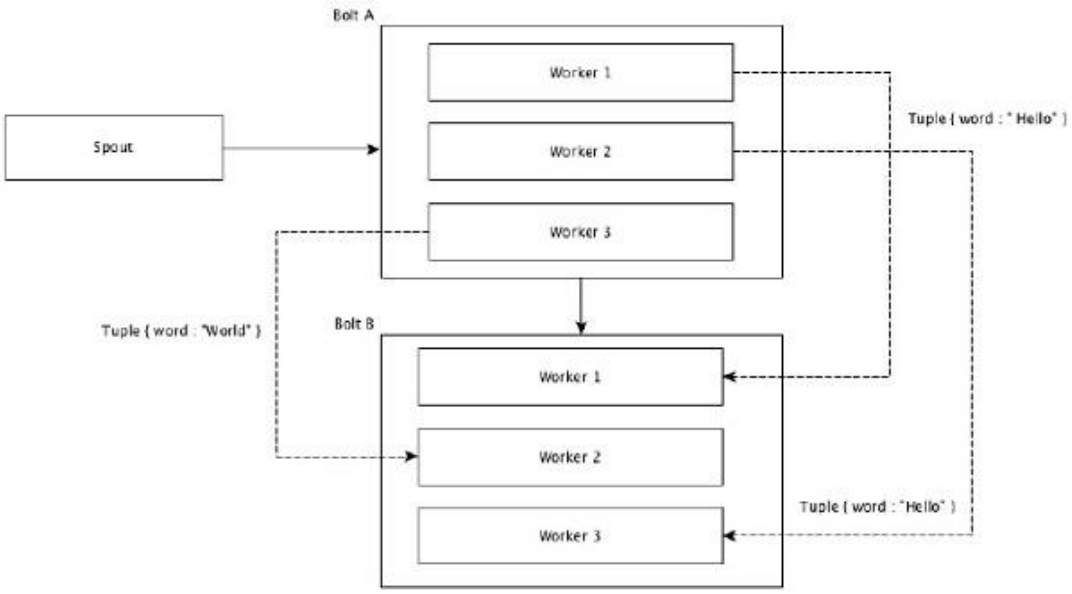

Fig 5 Shuffle Grouping

In Fields grouping, the fields with same values in tuples are grouped together and the remaining tuples kept outside. Then, the tuples with the same field values are sent forward to the same worker executing the bolts. For example, if the stream is grouped by the field "word", then the tuples with the same string, "Hello" will move to the same worker. The following diagram shows how Field Grouping works.

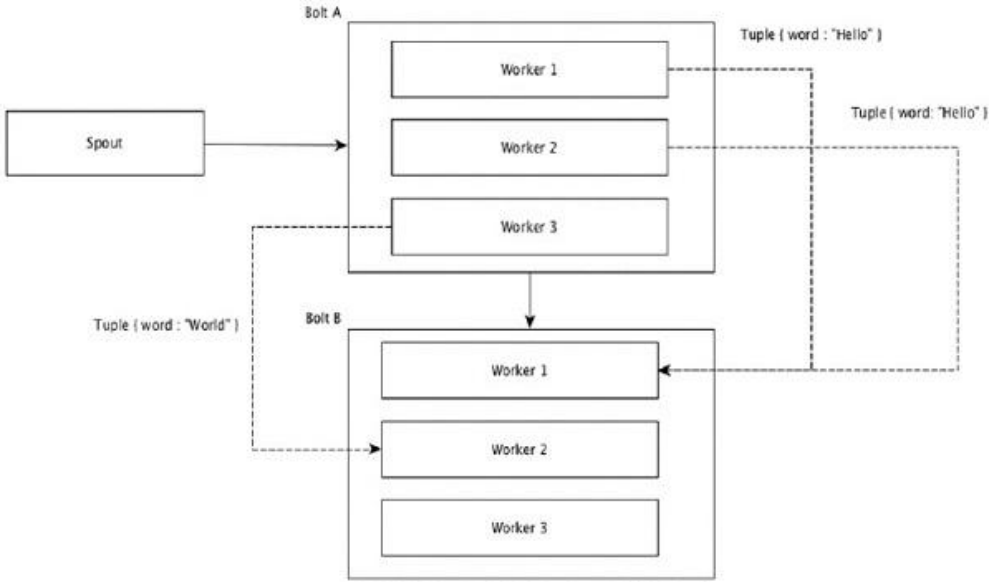

Fig 6 Fields Grouping

In Global grouping, all the streams can be grouped and forward to one bolt. This grouping sends tuples generated by all instances of the source to a single target instance (specifically, pick the worker with lowest ID).

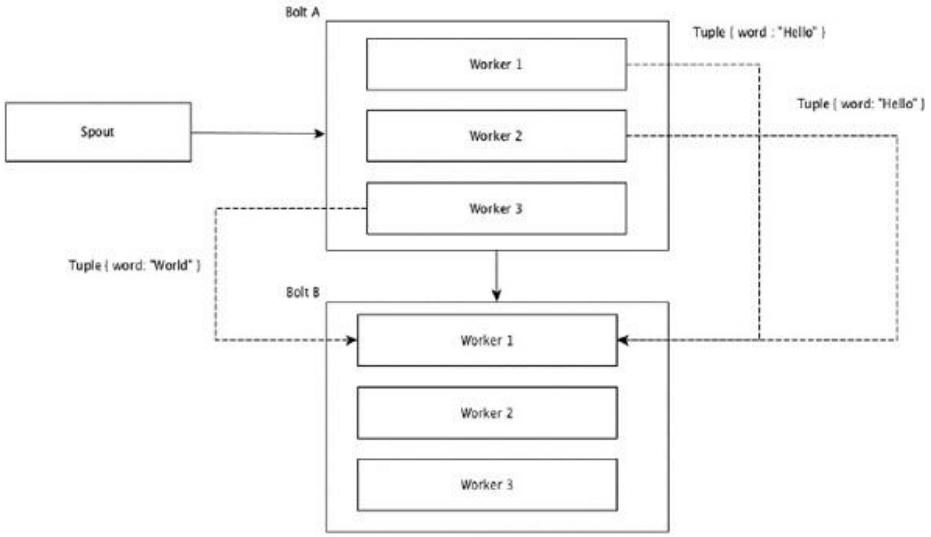

Fig 7 Global Grouping

All Grouping sends a single copy of each tuple to all instances of the receiving bolt. This kind of grouping is used to send signals to bolts. All grouping is useful for join operations.

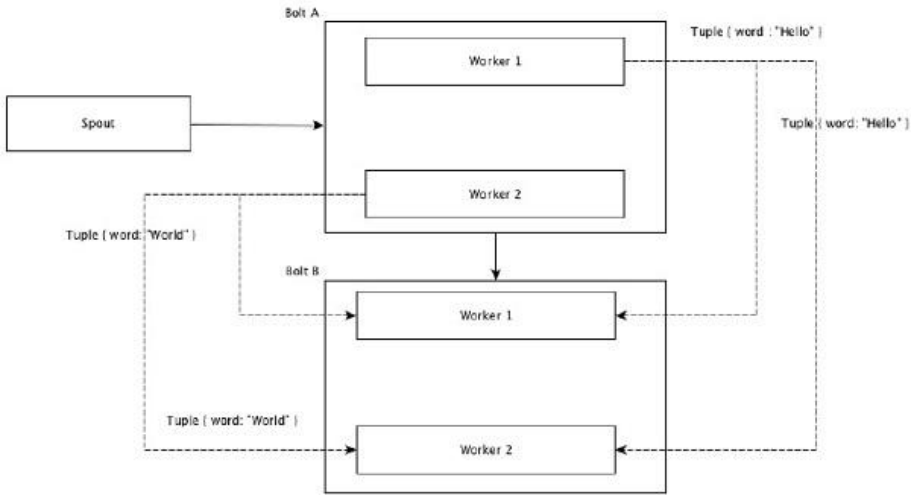

Fig 8 All Grouping

Apache Storm has two type of nodes, Nimbus (master node) and Supervisor (worker node). Nimbus is the central component of Apache Storm. The main job of Nimbus is to run the Storm topology. Nimbus analyzes the topology and gathers the task to be executed. Then, it will distribute the task to an available supervisor. A supervisor will have one or more worker process. Supervisor will delegate the tasks to worker processes. Worker process will spawn as many executors as needed and run the task. Apache Storm uses an internal distributed messaging system for the communication between nimbus and supervisors. Initially, the nimbus will wait for the "Storm Topology" to be submitted to it.Once a topology is submitted, it will process the topology and gather all the tasks that are to be carried out and the order in which the task is to be executed. Then, the nimbus will evenly distribute the tasks to all the available supervisors. At a particular time interval, all supervisors will send heartbeats to the nimbus to inform that they are still alive. When a supervisor dies and doesn't send a heartbeat to the nimbus, then the nimbus assigns the tasks to another supervisor. When the nimbus itself dies, supervisors will work on the already assigned task without any issue. Once all the tasks are completed, the supervisor will wait for

a new task to come in. In the meantime, the dead nimbus will be restarted automatically by service monitoring tools. The restarted nimbus will continue from where it stopped. Similarly, the dead supervisor can also be restarted automatically. Since both the nimbus and the supervisor can be restarted automatically and both will continue as before, Storm is guaranteed to process all the tasks at least once. Once all the topologies are processed, the nimbus waits for a new topology to arrive and similarly the supervisor waits for new tasks. There are two modes of operation in Storm namely Local and Production. Local Mode is used for development, testing, and debugging because it is the easiest way to see all the topology components working together. In this mode, we can adjust parameters that enable us to see how our topology runs in different Storm configuration environments. In this mode, storm topologies run on the local machine in a single JVM. In the production mode, we submit our topology to the working storm cluster, which is composed of many processes, usually running on different machines. Here, a working cluster will run indefinitely until it is shut down.

Apache Storm processes real-time data and the input normally comes from a message queuing system. An external distributed messaging system will provide the input necessary for the realtime computation. Spout will read the data from the messaging system and convert it into tuples and input into the Apache Storm. The interesting fact is that Apache Storm uses its own distributed messaging system internally for the communication between its nimbus and supervisor. Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging systems. A distributed messaging system provides the benefits of reliability, scalability, and persistence. Most of the messaging patterns follow the publish-subscribe model (simply Pub-Sub) where the senders of the messages are called publishers and those who want to receive the messages are called subscribers. Once the message has been published by the sender, the subscribers can receive the selected message with the help of a filtering option. Usually we have two types of filtering, one is topic-based filtering and another one is content-based filtering. Note that the pub-sub model can communicate only via messages. It is a very loosely coupled architecture; even the senders don't know who their subscribers are. Many of the message patterns enable with message broker to exchange publish messages for timely access by many subscribers. A real-life example is Dish TV, which publishes different

channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available. Storm extensively uses Thrift Protocol for its internal communication and data definition. Storm topology is simply Thrift Structs. Storm Nimbus that runs the topology in Apache Storm is a Thrift service.

We need Java, Zookeeper and Storm to be installed to start processing the data. Once we have installed all these software's, we must clone the storm starter project using Github and Git on the local machine. We use Maven a build tool to manage all the dependencies associated with the project. Maven is generally used to create JARs and Storm generally uses these JAR files to run topologies. We can run a topology by using the Storm jar command shown in the illustration below.

```
storm jar all-my-code.jar org.apache.storm.MyTopology arg1 arg2
```

3.6 MAVEN

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. Maven provides developers a complete build lifecycle framework. Development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle. In case of multiple development teams environment, Maven can set-up the way to work as per standards in a very short time. As most of the project setups are simple and reusable, Maven makes life of developer easy while creating reports, checks, build and testing automation setups. To summarize, Maven simplifies and standardizes the project build process. It handles compilation, distribution, documentation, team collaboration and other tasks seamlessly. Maven increases reusability and takes care of most of build related tasks. In order to build the project, Maven provides developers options to mention life-cycle goals and project dependencies (that rely on Maven plugin capabilities and on its default conventions). Much of the project management and build related tasks are maintained by

Maven plugins. Developers can build any given Maven project without need to understand how the individual plugins work.

POM stands for Project Object Model. It is fundamental Unit of Work in Maven. It is an XML file. It always resides in the base directory of the project as pom.xml.The POM contains information about the project and various configuration detail used by Maven to build the project(s).POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal. Maven pom.xml is also not required to be written manually. Maven provides numerous archetype plugins to create projects which in order create the project structure and pom.xml. Maven also helps the developer develop tests to run and when we package the code to for a JAR, these tests will be run and we will get a build success only if these tests pass. We can complete that by running the command mvn package, which will result in build success when the tests run successfully. Any storm topology is run using maven since it is a very reliable build tool which can be used across developer paradigms without any issues.

A large software application generally consists of multiple modules and it is common scenario where multiple teams are working on different modules of same application. For example consider a team is working on the front end of the application as app-ui project (app-ui.jar:1.0) and they are using data-service project (data-service.jar:1.0).Now it may happen that team working on data-service is undergoing bug fixing or enhancements at rapid pace and they are releasing the library to remote repository almost every other day.Now if data-service team uploads a new version every other day then many problems will arise. The data-service team should tell app-ui team every time when they have released an updated code. The app-ui team required to update their pom.xml regularly to get the updated version. To handle such kind of situation, SNAPSHOT concept comes into play. SNAPSHOT is a special version that indicates a current development copy. Unlike regular versions, Maven checks for a new SNAPSHOT version in a remote repository for every build. Now data-service team will release SNAPSHOT of its updated code everytime to repository say data-service:1.0-SNAPSHOT is replacing an older SNAPSHOT jar. In case of Version, if Maven once downloaded the mentioned versions say data-service: 1.0, it will never try to download

a newer 1.0 available in repository. To download the updated code, data-service version is be upgraded to 1.1.In case of SNAPSHOT, Maven will automatically fetch the latest SNAPSHOT (data-service: 1.0-SNAPSHOT) everytime app-ui team build their project. This SNAPSHOT functionality is extremely useful for collaboration between multiple teams and reduces the complexity of coding and communication between multiple teams while encouraging the concept of code reusability.

3.7 REDIS

Redis is an open source, BSD licensed, advanced key-value store. It is often referred to as a data structure server, since the keys can contain strings, hashes, lists, sets and sorted sets. Redis is written in C. It helps us create and deploy a highly scalable and performance-oriented system. It is an apt solution for building high performance, scalable web applications. Redis is very fast and can perform about 110000 SETs per second, about 81000 GETs per second. It natively supports most of the datatypes that developers already know such as list, set, sorted set, and hashes. This makes it easy to solve a variety of problems as we know which problem can be handled better by which data type. All Redis operations are atomic, which ensures that if two clients concurrently access, Redis server will receive the updated value. Redis is a multi-utility tool and can be used in a number of use cases such as caching, messaging-queues (Redis natively supports Publish/Subscribe), any short-lived data in your application, such as web application sessions, web page hit counts, etc. The support for publish-subscribe model ensures the application for this project.

Redis is an in-memory database but persistent on disk database, hence it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than the memory. Another advantage of in-memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk. Thus, Redis can do a lot with little internal complexity. Redis supports 5 types of data types namely String, Hash, List, Sets and Sorted Sets. Redis string is a sequence of bytes. Strings in Redis are binary safe, meaning they have a known length not determined by any special terminating characters. Thus, you can store

anything up to 512 megabytes in one string. A Redis hash is a collection of key value pairs. Redis Hashes are maps between string fields and string values. Hence, they are used to represent objects. Redis Lists are simply lists of strings, sorted by insertion order. You can add elements to a Redis List on the head or on the tail. Redis Sets are an unordered collection of strings. In Redis, you can add, remove, and test for the existence of members in O(1) time complexity. Redis Sorted Sets are similar to Redis Sets, non-repeating collections of Strings. The difference is, every member of a Sorted Set is associated with a score, that is used in order to take the sorted set ordered, from the smallest to the greatest score. While members are unique, the scores may be repeated.

The usage of Redis server or database requires the presence of the Redis client which can be set up using the command redis-cli.

```
$redis-cli
redis 127.0.0.1:6379>
redis 127.0.0.1:6379> PING
PONG
```

This command will create a client which will use an IP address and a port of the computer for the purpose of communication. There is also an option to communicate safely with the server using the passwords. The client then has to use the AUTH command followed by the password to log into a session with the server. We can check this later using the PING command to check whether the client is connected to the server or not.

```
redis 127.0.0.1:6379> AUTH "password"
OK
redis 127.0.0.1:6379> PING
PONG
```

This project involves the usage of the in-memory database redis to store the tweets temporary for rendering on the local server set up using the Flask library. The redisdatabse will only contain the top trending hashtags which will be displayed using the word cloud setup in the d3.js library.

3.8 FLASK

Flask is a web application framework written in Python. Armin Ronacher, who leads an international group of Python enthusiasts named Pocco, develops it. Flask is based on Werkzeug, WSGI toolkit and Jinja2 template engine.Web Application Framework or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.Web Server Gateway Interface (WSGI) has been adopted as a standard for Python web application development. WSGI is a specification for a universal interface between the web server and the web applications. Werkzeug is a WSGI toolkit, which implements requests, response objects, and other utility functions. This enables building a web framework on top of it. The Flask framework uses Werkzeug as one of its bases. Jinja2 is a popular templating engine for Python. A web templating system combines a template with a certain data source to render dynamic web pages.

Flask is often referred to as a micro framework. It aims to keep the core of an application simple yet extensible. Flask does not have built-in abstraction layer for database handling, nor does it have form a validation support. Instead, Flask supports the extensions to add such functionality to the application. We test our flask micro-server by running the following code shown in the illustration below.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
   return 'Hello World'

if __name__ == '__main__':
   app.run()
```

The code written is a simple hello world program which is displayed when the user visits the homepage of the websites. In the same way we can determine the website structure by defining multiple paths and the user can accordingly enter the URL to visit the website. In the above example, '/' URL is bound with hello_world() function. Hence, when the home

page of web server is opened in browser, the output of this function will be rendered. Finally the run() method of Flask class runs the application on the local development server. The run() method can be called with four optional parameters namely host, port, debug and options. The absence of these values ensures that the default values are given. The default values are 127.0.0.1, 5000, false and forwarded to underlying Werkzeug server for the host, port, debug and options respectively.

The execution of this web application is also very simple. We simply run the command python app_name.py and then go to the host URL to observer the output. To stop the execution of the web application we simply type CTRL + C in the command prompt, this command will help us exit the execution of the python script. Flask also provides special support for debugging an application. A Flask application is started by calling the run() method. However, while the application is under development, it should be restarted manually for each change in the code. To avoid this inconvenience, enable debug support. The server will then reload itself if the code changes. It will also provide a useful debugger to track the errors if any, in the application. The Debug mode is enabled by setting the debug property of the application object to True before running or passing the debug parameter to the run() method. We can observe this in the code snippet shown below in the illustration.

```
app.debug = True
app.run()
app.run(debug = True)
```

Now when we execute the web application using the python command, we can observer changes dynamically as and when we perform changes in the python code and this proves to be very useful for developers who need not run the application again and again to observe even minor changes. The route() function is used to bind the URL to specific functionality. For example in the code snippet shown below in the illustration, when the user visits the http://localhost:5000/hello URL, the output of the hello_world() function will be rendered in the browser.

```
@app.route('/hello')
def hello_world():
   return 'hello world'
```

3.9 BEAUTIFUL SOUP

Beautiful Soup is a python library which is useful for web scrapping. Web scraping (web harvesting or web data extraction) is a computer software technique of extracting information from websites.HTML parsing is easy in Python, especially with help of the BeautifulSoup library. The usage of Beautiful Soup library requires the presence of Beautiful Soup module and Requests module. These modules can be downloaded simply using the PIP functionality of python. To download these modules, we simply run the commands displayed in the illustration below.

```
$ pip install requests

$ pip install beautifulsoup4
```

The PIP installer will ensure the availability of these modules for all scripts for future use. Beautiful Soup provides a few simple methods and Pythonic idioms for navigating, searching, and modifying a parse tree: a toolkit for dissecting a document and extracting what you need. It doesn't take much code to write an application. Beautiful Soup automatically converts incoming documents to Unicode and outgoing documents to UTF-8. You don't have to think aboutencodings, unless the document doesn't specify an encoding and Beautiful Soup can't auto detect one. Then you just have to specify the original encoding. Beautiful Soup sits on top of popular Python parsers like lxml and html5lib, allowing you to try out different parsing strategies or trade speed for flexibility. We can test the working of Beautiful Soup by executing a simple python script as shown in the illustration below.

```
from bs4 import BeautifulSoup

import requests

url = raw_input("Enter a website to extract the URL's from: ")

r  = requests.get("http://" +url)

data = r.text

soup = BeautifulSoup(data)

for link in soup.find_all('a'):
   print(link.get('href'))
```

This code will display all the websites mentioned in a particular URL. This is a very simple and elegant way to parse a webpage and it also abstracts most of the methods from the developer and this helps the developer focus on the core tasks of developing the application. The sample output when we run this code with the input as "www.pythonforbeginners.com" is shown below in the illustration.

```
Enter a website to extract the URL's from: www.pythonforbeginners.com
http://www.pythonforbeginners.com
http://www.pythonforbeginners.com/python-overview-start-here/
http://www.pythonforbeginners.com/dictionary/
http://www.pythonforbeginners.com/python-functions-cheat-sheet/
http://www.pythonforbeginners.com/lists/python-lists-cheat-sheet/
http://www.pythonforbeginners.com/loops/
http://www.pythonforbeginners.com/python-modules/
http://www.pythonforbeginners.com/strings/
http://www.pythonforbeginners.com/sitemap/
http://www.pythonforbeginners.com/feed/
http://www.pythonforbeginners.com
```

All the websites present on the website are displayed on separate lines as shown. f you pass in a regular expression object, Beautiful Soup will filter against that regular expression using its match() method. This library is very important since it provides a very simple way for the scrapping for twitter data for sentiment analysis.

3.10 D3.js

D3.js is a library which provides access to CSS and JavaScript enabled visualizations and animations. This library is used frequently by data analysts and data presenters throughout the world. It is considered to be the gold standard for data visualization. D3 uses CSS-style selectors to identify elements on which to operate, so it's important to understand how to use them. The HTML head always includes the CSS declarations. This is followed

here as well. The script tags are used to include the java script associated with a website. The D3 library is used in a very similar way.

D3 is at its best when rendering visuals as Scalable Vector Graphics. SVG is a text-based image format. Meaning, you can specify what an SVG image should look like by writing simple markup code, sort of like HTML tags. In fact, SVG code can be included directly within any HTML document. Web browsers have supported the SVG format for years (except for Internet Explorer), but it never quite caught on, until now. The visualizations made using D3 also have an inherent advantage in terms of responsiveness. The responsiveness of the visualization makes it extremely useful and the developer's work is simplified.

The setup of D3 is extremely simple. We download D3 into the folder where the visualization is being developed. We create a separate folder in the project folder for D3 and then we download the library into that folder from the man website. We can then use all the functions of D3 by including script tag shown in an illustration below.

```
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

The output of this webpage is observed using the local host or the XAMP server as usual. D3 uses a Document Object Model i.e. DOM Model. We first set up the DOM using the commands between the script tags shown earlier. One example of that is shown in the code snippet shown below.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
    <script type="text/javascript" src="../../d3/d3.v3.js"></script>
    <style type="text/css"></style>
  </head>
  <body>
    <script type="text/javascript">d3.select("body").append("p").text("New
    paragraph!");</script>
    <p>New paragraph!</p>
  </body>
</html>
```

Now in the DOM, there is a new paragraph element that was generated on-the-fly! This technique is used again and again to generate code on the fly for each and every element.

CHAPTER-4

SYSTEM DESIGN

4.1 SYSTEM REQUIREMENTS

System specification documents most predominantly contain information on basic website requirements which include:

Performance levels

Reliability

Quality

Interfaces

Security and Privacy

Constraints and Limitations

Functional Capabilities

System specifications are:

1. Pentium 4 with minimum 1.x GHz processor or equivalent processor

2. Minimum 1 GB RAM (2 GB RAM recommended)

3. Hard disk with minimum 1 GB free space

4. NIC (network interface card) connected to network pentium III

5. RAM : 32 Bit Computer

6. Hard disk : 500 GB

7. Monitor : SVGA color monitor

8. Keyboard : 105 standard mouse

9. Twitter Account and Twitter Registered App

10. Python 2.x Installed

11. Java Installed and Class PATH is set

12. Apache Storm

13. Maven Installed

14. Redis Installed along with Beautiful Soup

15. Vagrant Installed along with Virtual Box

4.2 MODULE DESCRIPTION

This project consists of two modules name the Streaming Module and the Sentiment Module. The modules have been designed with upmost focus on automation and reduction of the user's work so as to ensure that even naïve users can use the project in real time applications to gauge the opinion of the public. The modules work independently and hence are not prone to errors. This independence also helps when twitter suffers from problems. The modules each have applications hosted on Twitter that enable access to tweets as and when made available.

The Sentiment module is written primarily using Python. It uses the concepts of Natural Language Processing Toolkit i.e. NLTK and involves the use of Bayes Classifier. It uses the Beautiful Soup Library along with Tweepy modules to retrieve tweets from Twitter using the credentials granted for the application. We use the tweets and filter them first to remove the delimiters and other words which do not have any effect what so ever on the sentiment relating to a topic. The filtered tweets are sent to the Bayes classifier. The classifier will classify the tweets into positive or negative or neutral depending upon the words used in the tweets. This classifier is considered to be one of the most accurately predictive classifier. It is the simplest algorithm which uses the concepts of frequency distribution to calculate

initial probabilities. These initial probabilities are then used to calculate posterior probability. This posterior probability represents the sentiment of the people tweeting on a particular topic. These sentiments in terms of percent of positive or negative or neutral tweets are displayed in the python IDLE window along with some of the most popular tweets.

The second module is the Streaming module. This module is developed using Java and Clojure. It consists of spouts and bolts, which are the processing units in Apache Storm. These units are built using Maven, which is a built tool. Maven converts the code into a Java Archive Resource file or a JAR file. This JAR file is then run by Storm which has the Nimbus always running using Supervisors. The spout keeps collecting tweets till the user kills the process in Storm using Ctrl + C or the vagrant destroy command. The tweets are processed using a series of bolts, each of which serve a particular functionality. The first bolt is used to filter the tweets and remove delimiters and words which do not contribute to a tweet. These delimiters and words are mainly used because of Grammar rules and do not reflect perceptions of people. The second bolt converts the tweets into a key value pair. This is done to understand the trending hashtag. This trending hashtag becomes the key and the value is the count of people tweeting about that particular topic. This stage is called countable bolt. The next bolt is used to rank the tweets in terms of the value from the key value pair. It forms the descending order of tweets and helps us access the top N tweets. Here the N is specified by the user before running the program. This bolt is called the Ranking Bolt. The output of the Ranking Bolt is connected to the last bolt called the Report Bolt. This bolt is responsible for calling and accessing the in memory database Redis. Redis stores the values temporarily and these values get refreshed as and when new tweets come in. Redis is linked to the python micro server setup using the Flask library. This micro server runs on local host and displays the keys and values from the in memory database Redis. The visualization is made using the D3.js library using SVG graphics. The HTML document is defined in the beginning and the D3 library is accessed using the script tags. The visualization is made using a word cloud which changes and grows in size. We also use rotational animations to make the user realize the concept of Streaming data. The size of the trending topics displayed is proportional to their popularity. The most popular trending topics have the largest size and vice-versa.

CHAPTER 5

IMPLEMENTATION

5.1 SENTIMENT ANALYSIS MODULE

```
import re

import numpy as np

import tweepy

from tweepy import OAuthHandler

from textblob import TextBlob


class TwitterClient(object):
    '''
    Generic Twitter Class for sentiment analysis.
    '''
    def __init__(self):
        '''
        Class constructor or initialization method.
        '''
        # keys and tokens from the Twitter Dev Console
        consumer_key = 'sESjkOJeISTasX0npnInCVEvD'
        consumer_secret = 'ZngxknIobQxKGjC1bISocKsQnHPurhDn6njsr4I2EzDEnxPyS3'
        access_token = '849506572468449284-htq5H96RqmVmpHrPGuH4xOoyXsDbGKw'
```

```
        access_token_secret = 'yuwj2OeyZ23YE6iHeJfZtwF3ssZXajt3a7kmbZm7O5jJj'
        # attempt authentication
        try:
            # create OAuthHandler object
            self.auth = OAuthHandler(consumer_key, consumer_secret)
            # set access token and secret
            self.auth.set_access_token(access_token, access_token_secret)
            # create tweepy API object to fetch tweets
            self.api = tweepy.API(self.auth)
        except:
            print("Error: Authentication Failed")

    def clean_tweet(self, tweet):
        '''
        Utility function to clean tweet text by removing links, special characters
        using simple regex statements.
        '''
        return ' '.join(re.sub("(@[A-Za-z0-9]+)|([^0-9A-Za-z \t])|(\w+:\/\/\S+)", " ",
tweet).split())
```

```
def get_tweet_sentiment(self, tweet):
    '''
    Utility function to classify sentiment of passed tweet
    using textblob's sentiment method
    '''
    # create TextBlob object of passed tweet text
    analysis = TextBlob(self.clean_tweet(tweet))
    # set sentiment
    if analysis.sentiment.polarity > 0:
        return 'positive'
    elif analysis.sentiment.polarity == 0:
        return 'neutral'
    else:
        return 'negative'


def get_tweets(self, query, count = 10):
    '''
    Main function to fetch tweets and parse them.
    '''
    # empty list to store parsed tweets
    tweets = []
```

```
try:
    # call twitter api to fetch tweets
    fetched_tweets = self.api.search(q = query, count = count)

    # parsing tweets one by one
    for tweet in fetched_tweets:
        # empty dictionary to store required params of a tweet
        parsed_tweet = {}

        # saving text of tweet
        parsed_tweet['text'] = tweet.text
        # saving sentiment of tweet
        parsed_tweet['sentiment'] = self.get_tweet_sentiment(tweet.text)

        # appending parsed tweet to tweets list
        if tweet.retweet_count > 0:
            # if tweet has retweets, ensure that it is appended only once
            if parsed_tweet not in tweets:
                tweets.append(parsed_tweet)
        else:
```

```
                tweets.append(parsed_tweet)

            # return parsed tweets
            return tweets

        except tweepy.TweepError as e:
            # print error (if any)
            print("Error : " + str(e))

def main():
    # creating object of TwitterClient Class
    api = TwitterClient()
    # calling function to get tweets
    tweets = api.get_tweets(query = 'SRH', count = 10)

    # picking positive tweets from tweets
    ptweets = [tweet for tweet in tweets if tweet['sentiment'] == 'positive']
    # percentage of positive tweets
    print("Positive tweets percentage: {} %".format(100*len(ptweets)/len(tweets)))
    # picking negative tweets from tweets
    ntweets = [tweet for tweet in tweets if tweet['sentiment'] == 'negative']
```

```
    # percentage of negative tweets
    print("Negative tweets percentage: {} %".format(100*len(ntweets)/len(tweets)))
    xt = 100*len(ptweets)/len(tweets)
    yt = 100*len(ntweets)/len(tweets)
    zt = xt + yt
    pt = 100 - zt
    rt = int(pt)
    print ("neutral tweets")
    print(rt)
     # printing first 5 positive tweets
    print("\n\nPositive tweets:")
    for tweet in ptweets[:10]:
        print(tweet['text'])


    # printing first 5 negative tweets
    print("\n\nNegative tweets:")
    for tweet in ntweets[:10]:
        print(tweet['text'])
if __name__ == "__main__":
    # calling main function
    main()
```

5.2 TWEET TOPLOGY

```
package udacity.storm;

import backtype.storm.Config;

import backtype.storm.LocalCluster;

import backtype.storm.StormSubmitter;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.testing.TestWordSpout;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.topology.TopologyBuilder;

import backtype.storm.topology.base.BaseRichSpout;

import backtype.storm.topology.base.BaseRichBolt;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Tuple;

import backtype.storm.tuple.Values;

import backtype.storm.utils.Utils;

import udacity.storm.spout.RandomSentenceSpout;


class TopNTweetTopology

{
```

```
public static void main(String[] args) throws Exception
{
 //Variable TOP_N number of words
 int TOP_N = 10;
 // create the topology
 TopologyBuilder builder = new TopologyBuilder();
 /*
  * In order to create the spout, you need to get twitter credentials
  * If you need to use Twitter firehose/Tweet stream for your idea,
  * create a set of credentials by following the instructions at
  *
  * https://dev.twitter.com/discussions/631
  *
  */
 // now create the tweet spout with the credentials
 TweetSpout tweetSpout = new TweetSpout(
 "MedH9l8TcIViaNbi0TrQxxOlp",
 "JJNzHmyRPeqF1b3brSWylOYpaS3orxY7EeFl05jHfnjFcxoL0L",
 "1351146829-WU82A0YbG9uoNNNSPHD43AaBDwP2q6Mb9uX3hwI",
 "jPrrno88OMiZjEwCf8XOFaFUEFgCukb93z4szNuypBvq8"
```

```
);

// attach the tweet spout to the topology - parallelism of 1

builder.setSpout("tweet-spout", tweetSpout, 1);

// attach the Random Sentence Spout to the topology - parallelism of 1

//builder.setSpout("random-sentence-spout", new RandomSentenceSpout(), 1);

// attach the parse tweet bolt using shuffle grouping

builder.setBolt("parse-tweet-bolt", new ParseTweetBolt(), 10).shuffleGrouping("tweet-spout");

//builder.setBolt("parse-tweet-bolt", new ParseTweetBolt(), 10).shuffleGrouping("random-sentence-spout");

// attach the count bolt using fields grouping - parallelism of 15

builder.setBolt("count-bolt", new CountBolt(), 15).fieldsGrouping("parse-tweet-bolt", new Fields("tweet-word"));

// attach rolling count bolt using fields grouping - parallelism of 5

// TEST

//builder.setBolt("rolling-count-bolt", new RollingCountBolt(30, 10), 1).fieldsGrouping("parse-tweet-bolt", new Fields("tweet-word"));

//from incubator-storm/.../storm/starter/RollingTopWords.java

//builder.setBolt("intermediate-ranker", new IntermediateRankingsBolt(TOP_N), 4).fieldsGrouping("rolling-count-bolt", new Fields("obj"));

builder.setBolt("intermediate-ranker", new IntermediateRankingsBolt(TOP_N), 4).fieldsGrouping("count-bolt", new Fields("word"));
```

```
    builder.setBolt("total-ranker", new
TotalRankingsBolt(TOP_N)).globalGrouping("intermediate-ranker");
    // attach the report bolt using global grouping - parallelism of 1
    builder.setBolt("report-bolt", new ReportBolt(), 1).globalGrouping("total-ranker");
    // create the default config object
    Config conf = new Config();
    // set the config in debugging mode
    conf.setDebug(true);
    if (args != null && args.length > 0) {
      // run it in a live cluster
      // set the number of workers for running all spout and bolt tasks
      conf.setNumWorkers(3);
      // create the topology and submit with config
      StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    } else {
      // run it in a simulated local cluster
      // set the number of threads to run - similar to setting number of workers in live cluster
      conf.setMaxTaskParallelism(3);
      // create the local cluster instance
      LocalCluster cluster = new LocalCluster();
      // submit the topology to the local cluster
```

```
        cluster.submitTopology("tweet-word-count", conf, builder.createTopology());
        // let the topology run for 300 seconds. note topologies never terminate!
        Utils.sleep(300000);
        // now kill the topology
        cluster.killTopology("tweet-word-count");
        // we are done, so shutdown the local cluster
        cluster.shutdown();
    }
  }
}
```

5.3 TWEET SPOUT

```
package udacity.storm;
import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.testing.TestWordSpout;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.TopologyBuilder;
```

```
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import twitter4j.conf.ConfigurationBuilder;
import twitter4j.TwitterStream;
import twitter4j.TwitterStreamFactory;
import twitter4j.Status;
import twitter4j.StatusDeletionNotice;
import twitter4j.StatusListener;
import twitter4j.StallWarning;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.LinkedBlockingQueue;
/**
 * A spout that uses Twitter streaming API for continuously
 * getting tweets
 */
public class TweetSpout extends BaseRichSpout
```

```
{
 // Twitter API authentication credentials
 String custkey, custsecret;
 String accesstoken, accesssecret;
 // To output tuples from spout to the next stage bolt
 SpoutOutputCollector collector;
 // Twitter4j - twitter stream to get tweets
 TwitterStream twitterStream;
 // Shared queue for getting buffering tweets received
 LinkedBlockingQueue<String> queue = null;
 // Class for listening on the tweet stream - for twitter4j
 private class TweetListener implements StatusListener {
  // Implement the callback function when a tweet arrives
  @Override
  public void onStatus(Status status)
  {
   // add the tweet into the queue buffer
   queue.offer(status.getText());
  }
  @Override
  public void onDeletionNotice(StatusDeletionNotice sdn)
```

```
  {

  }

  @Override

  public void onTrackLimitationNotice(int i)

  {

  }

  @Override

  public void onScrubGeo(long l, long l1)

  {

  }

  @Override

  public void onStallWarning(StallWarning warning)

  {

  }

  @Override

  public void onException(Exception e)

  {

    e.printStackTrace();

  }

};

/**
```

```
 * Constructor for tweet spout that accepts the credentials
 */
public TweetSpout(
    String              key,
    String              secret,
    String              token,
    String              tokensecret)
{
  custkey = key;
  custsecret = secret;
  accesstoken = token;
  accesssecret = tokensecret;
}
@Override
public void open(
    Map                 map,
    TopologyContext     topologyContext,
    SpoutOutputCollector    spoutOutputCollector)
{
  // create the buffer to block tweets
  queue = new LinkedBlockingQueue<String>(1000);
```

```
 // save the output collector for emitting tuples
 collector = spoutOutputCollector;
 // build the config with credentials for twitter 4j
 ConfigurationBuilder config =
    new ConfigurationBuilder()
          .setOAuthConsumerKey(custkey)
          .setOAuthConsumerSecret(custsecret)
          .setOAuthAccessToken(accesstoken)
          .setOAuthAccessTokenSecret(accesssecret);
 // create the twitter stream factory with the config
 TwitterStreamFactory fact =
    new TwitterStreamFactory(config.build());
 // get an instance of twitter stream
 twitterStream = fact.getInstance();
 // provide the handler for twitter stream
 twitterStream.addListener(new TweetListener());

 // start the sampling of tweets
 twitterStream.sample();
}
@Override
```

```
public void nextTuple()
{
  // try to pick a tweet from the buffer
  String ret = queue.poll();
  // if no tweet is available, wait for 50 ms and return
  if (ret==null)
  {
    Utils.sleep(50);
    return;
  }
  // now emit the tweet to next stage bolt
  collector.emit(new Values(ret));
}
@Override
public void close()
{
  // shutdown the stream - when we are going to exit
  twitterStream.shutdown();
}

/**
```

```
 * Component specific configuration
 */
@Override
public Map<String, Object> getComponentConfiguration()
{
  // create the component config
  Config ret = new Config();
  // set the parallelism for this spout to be 1
  ret.setMaxTaskParallelism(1);
  return ret;
}
@Override
public void declareOutputFields(
    OutputFieldsDeclarer outputFieldsDeclarer)
{
  // tell storm the schema of the output tuple for this spout
  // tuple consists of a single column called 'tweet'
  outputFieldsDeclarer.declare(new Fields("tweet"));
 }
}
```

5.4 PARSE TWEET BOLT

```
package udacity.storm;

import backtype.storm.Config;

import backtype.storm.LocalCluster;

import backtype.storm.StormSubmitter;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.testing.TestWordSpout;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.topology.TopologyBuilder;

import backtype.storm.topology.base.BaseRichSpout;

import backtype.storm.topology.base.BaseRichBolt;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Tuple;

import backtype.storm.tuple.Values;

import backtype.storm.utils.Utils;

import java.util.Map;

import java.util.Arrays;

/**

 * A bolt that parses the tweet into words
```

```
 */
public class ParseTweetBolt extends BaseRichBolt
{
  // To output tuples from this bolt to the count bolt
  OutputCollector collector;
  private String[] skipWords = {"rt", "to", "me","la","on","that","que",
    "followers","watch","know","not","have","like","I'm","new","good","do",
    "more","es","te","followers","Followers","las","you","and","de","my","is",
    "en","una","in","for","this","go","en","all","no","don't","up","are",
    "http","http:","https","https:","http://","https://","with","just","your",
    "para","want","your","you're","really","video","it's","when","they","their","much",
    "would","what","them","todo","FOLLOW","retweet","RETWEET","even","right","like",
    "bien","Like","will","Will","pero","Pero","can't","were","Can't","Were","TWITTER",
    "make","take","This","from","about","como","esta","follows","followed"};
  @Override
  public void prepare(
      Map                     map,
      TopologyContext         topologyContext,
      OutputCollector         outputCollector)
  {
    // save the output collector for emitting tuples
```

```
  collector = outputCollector;
}
@Override
public void execute(Tuple tuple)
{
  // get the 1st column 'tweet' from tuple
  String tweet = tuple.getString(0);
  // provide the delimiters for splitting the tweet
  String delims = "[ .,?!]+";
  // now split the tweet into tokens
  String[] tokens = tweet.split(delims);
  // for each token/word, emit it
  for (String token: tokens) {
    //emit only words greater than length 3 and not stopword list
    if(token.length() > 3 && !Arrays.asList(skipWords).contains(token)){
      if(token.startsWith("#")){
        collector.emit(new Values(token));
      }
    }
  }
}
```

```
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer)
  {
    // tell storm the schema of the output tuple for this spout
    // tuple consists of a single column called 'tweet-word'
    declarer.declare(new Fields("tweet-word"));
  }
}
```

5.5 COUNT BOLT

```
package udacity.storm;
import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.testing.TestWordSpout;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.topology.base.BaseRichBolt;
```

```
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;
import java.util.HashMap;
import java.util.Map;
/**
 * A bolt that counts the words that it receives
 */
public class CountBolt extends BaseRichBolt
{
  // To output tuples from this bolt to the next stage bolts, if any
  private OutputCollector collector;
  // Map to store the count of the words
  private Map<String, Long> countMap;
  @Override
  public void prepare(
      Map                     map,
      TopologyContext         topologyContext,
      OutputCollector         outputCollector)
  {
```

```
  // save the collector for emitting tuples
  collector = outputCollector;
  // create and initialize the map
  countMap = new HashMap<String, Long>();
}
@Override
public void execute(Tuple tuple)
{
  // get the word from the 1st column of incoming tuple
  String word = tuple.getString(0);
  // check if the word is present in the map
  if (countMap.get(word) == null) {
    // not present, add the word with a count of 1
    countMap.put(word, 1L);
  } else {
    // already there, hence get the count
    Long val = countMap.get(word);
    // increment the count and save it to the map
    countMap.put(word, ++val);
  }
  // emit the word and count
```

```
    collector.emit(new Values(word, countMap.get(word)));
  }
  @Override
  public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer)
  {
    // tell storm the schema of the output tuple for this spout
    // tuple consists of a two columns called 'word' and 'count'
    // declare the first column 'word', second column 'count'
    outputFieldsDeclarer.declare(new Fields("word","count"));
  }
}
```

5.6 RANKING BOLT

```
package udacity.storm;
import backtype.storm.tuple.Tuple;
import org.apache.log4j.Logger;
//import storm.starter.tools.Rankings;
import udacity.storm.tools.Rankings;
/**
 * This bolt merges incoming {@link Rankings}.
 * <p/>
```

```
* It can be used to merge intermediate rankings generated by {@link IntermediateRankingsBolt} into a final,
* consolidated ranking. To do so, configure this bolt with a globalGrouping on {@link IntermediateRankingsBolt}.
*/
public final class TotalRankingsBolt extends AbstractRankerBolt {
private static final long serialVersionUID = -8447525895532302198L;
private static final Logger LOG = Logger.getLogger(TotalRankingsBolt.class);
public TotalRankingsBolt() {
super();
}
public TotalRankingsBolt(int topN) {
super(topN);
}
public TotalRankingsBolt(int topN, int emitFrequencyInSeconds) {
super(topN, emitFrequencyInSeconds);
}
@Override
void updateRankingsWithTuple(Tuple tuple) {
Rankings rankingsToBeMerged = (Rankings) tuple.getValue(0);
super.getRankings().updateWith(rankingsToBeMerged);
super.getRankings().pruneZeroCounts();
```

```
  }
  @Override
  Logger getLogger() {
    return LOG;
  }
}
```

5.7 INTERMEDIATE RANKING BOLT

```
package udacity.storm;
import backtype.storm.tuple.Tuple;
import org.apache.log4j.Logger;
//import storm.starter.tools.Rankable;
//import storm.starter.tools.RankableObjectWithFields;
import udacity.storm.tools.Rankable;
import udacity.storm.tools.RankableObjectWithFields;
/**
 * This bolt ranks incoming objects by their count.
 * <p/>
 * It assumes the input tuples to adhere to the following format: (object, object_count, additionalField1,
 */
public final class IntermediateRankingsBolt extends AbstractRankerBolt {
```

```
  private static final long serialVersionUID = -1369800530256637409L;
  private static final Logger LOG = Logger.getLogger(IntermediateRankingsBolt.class);
  public IntermediateRankingsBolt() {
    super();
  }
  public IntermediateRankingsBolt(int topN) {
    super(topN);
  }
  public IntermediateRankingsBolt(int topN, int emitFrequencyInSeconds) {
    super(topN, emitFrequencyInSeconds);
  }
  void updateRankingsWithTuple(Tuple tuple) {
    Rankable rankable = RankableObjectWithFields.from(tuple);
    super.getRankings().updateWith(rankable);
  }
  @Override
  Logger getLogger() {
    return LOG;
  }
}
```

5.8 REPORT BOLT

```
package udacity.storm;

import backtype.storm.Config;

import backtype.storm.LocalCluster;

import backtype.storm.StormSubmitter;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.testing.TestWordSpout;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.topology.TopologyBuilder;

import backtype.storm.topology.base.BaseRichSpout;

import backtype.storm.topology.base.BaseRichBolt;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Tuple;

import backtype.storm.tuple.Values;

import backtype.storm.utils.Utils;

import java.util.Map;

import com.lambdaworks.redis.RedisClient;

import com.lambdaworks.redis.RedisConnection;

import udacity.storm.tools.*;
```

```
import udacity.storm.tools.Rankings;
import com.google.common.collect.ImmutableList;
import com.google.common.collect.Lists;
/** * A bolt that prints the word and count to redis */
public class ReportBolt extends BaseRichBolt
{
  // place holder to keep the connection to redis
  transient RedisConnection<String,String> redis;
  @Override
  public void prepare(
      Map                 map,
      TopologyContext     topologyContext,
      OutputCollector     outputCollector)
  {
    // instantiate a redis connection
    RedisClient client = new RedisClient("localhost",6379);
    // initiate the actual connection
    redis = client.connect();
  }
  @Override
  public void execute(Tuple tuple)
```

```
  {
    Rankings rankableList = (Rankings) tuple.getValue(0);
    for (Rankable r: rankableList.getRankings()){
      String word = r.getObject().toString();
      Long count = r.getCount();
      redis.publish("WordCountTopology", word + "|" + Long.toString(count));
    }
    // access the first column 'word'
    //String word = tuple.getStringByField("word");
    // access the second column 'count'
    //String word = rankedWords.toString();
    //Integer count = tuple.getIntegerByField("count");
    //Long count = new Long(100);
    // publish the word count to redis using word as the key
    //redis.publish("WordCountTopology", word + ":" + Long.toString(count));
  }
  public void declareOutputFields(OutputFieldsDeclarer declarer)
  {
    // nothing to add - since it is the final bolt
  }
}
```

CHAPTER 6

SYSTEM TESTING

System Testing is a process of executing a program with the explicit intention of finding errors, which cause program failure. There are two general strategies for testing software. They are Code Testing and Specification testing.

6.1 CODE TESTING

This strategy examines the logic of a program and has been carried out to identify three levels of correctness of programs. Possible correctness is first achieved by giving arbitrary inputs. Then the inputs are carefully selected to obtain predicted output. This gives the probable correctness. All potentially problematic areas are checked in this way for the software to achieve probable correctness. Absolute correctness can be demonstrated by a test involving every possible combination of inputs. However, this cannot be performed with the software but to the existence of the various possible combinations of the inputs and due to time restrictions.

6.2 SPECIFICATION TESTING

The specifications are examined which states what the program should do and how it should perform under various conditions. Then test cases are developed for each condition or combinations of conditions and submitted for processing. By examining the results, it is determined whether the program performs according to its specified requirements.

6.3 LEVELS OF TESTING

The two levels of Testing are Unit Testing and System Testing. Each of these have their own inherent advantages and all the production ready code is written in such a way that the code undergoes both System Testing and Unit Testing.

6.4 UNIT TESTING

Unit testing is done for the programs making up the systems. It is focused to find out module errors and enables to detect errors in coding and logic that are contained in the module. Unit testing is performed from bottom-up, starting with the smallest and lowest levels modules and proceeding one.

6.5 SYSTEM TESTING

At a time System Testing finds out the discrepancies between the system and its original objective, current specifications and systems documentation.
The training session consists of getting the users used to software by asking them to perform data entry in our presence and look into the problems if encountered .
Testing can be done in two ways.

1. Sample Tests
2. Real Tests

6.6 SAMPLE TESTS

The software was tested with sample data that we randomly selected. I tested all functions with such random data and I was successful in getting accurate results. It was at this time I got to know certain intricacies of the system that I had overlooked. Without much delay however, I got over the problems and managed to perfect the software at least to the extent possible.

6.7 REAL TEST

For the real test, I have planned to do in due course. I initialized the software and creation of entities through the weather module, conversational entries through the Small Talk module and generated reports with the estimations. The various information retrieval

functions as per user need are also implemented that elucidates the emotional appeal and other information like displaying the headlines and also providing brief information about any subject.

CHAPTER 7

SCREENSHOTS

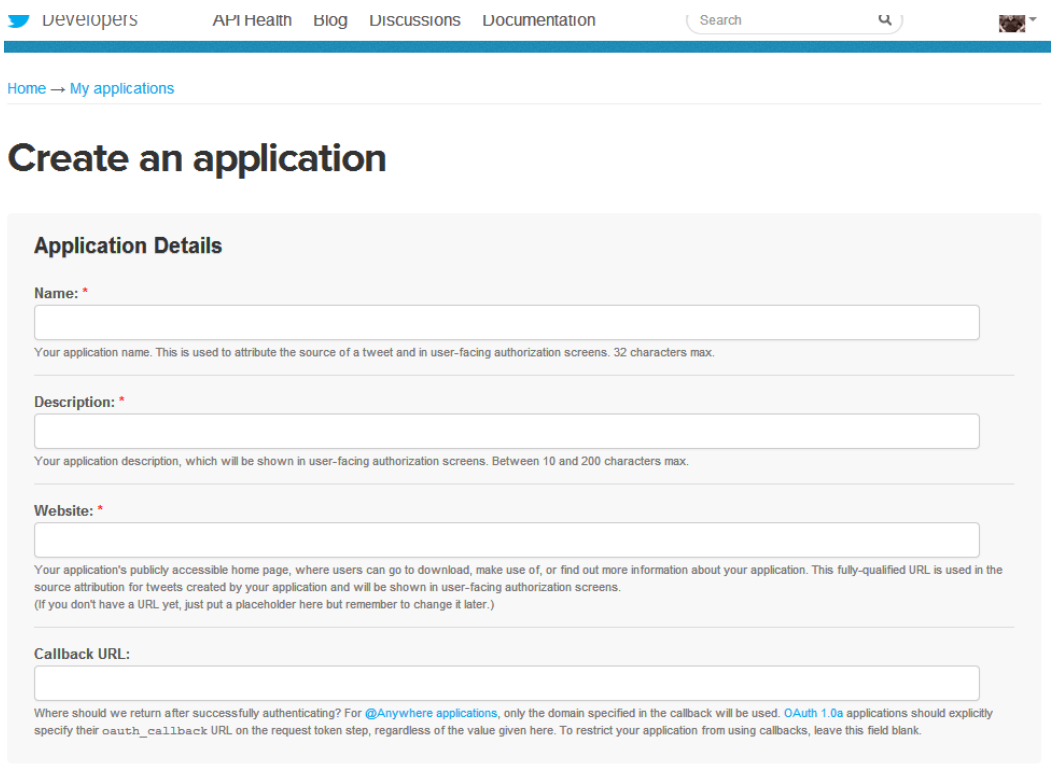

Fig 9 Twitter Application

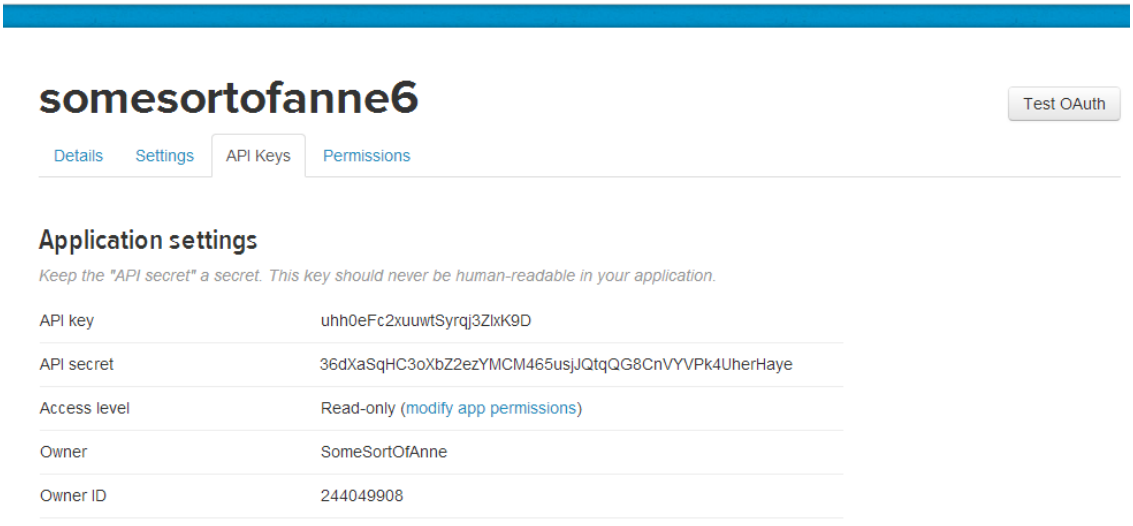

Fig 10 Application Keys

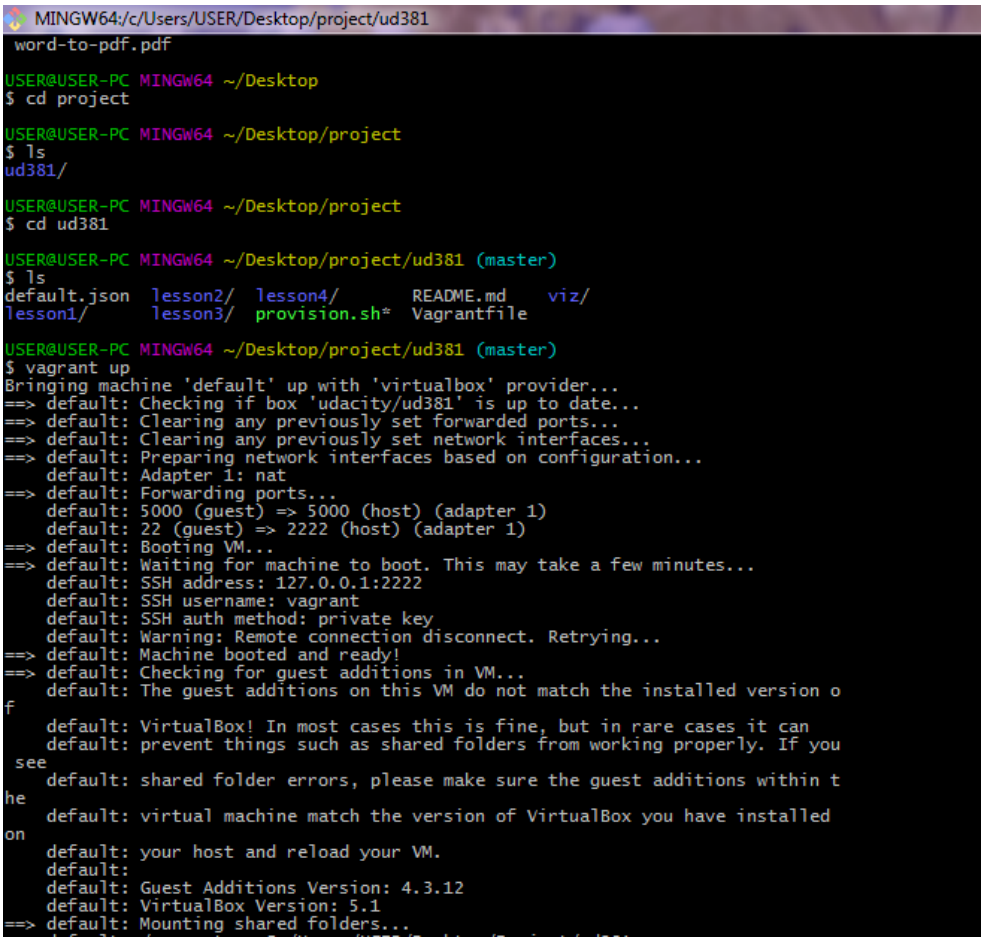

Fig 11 Vagrant Up

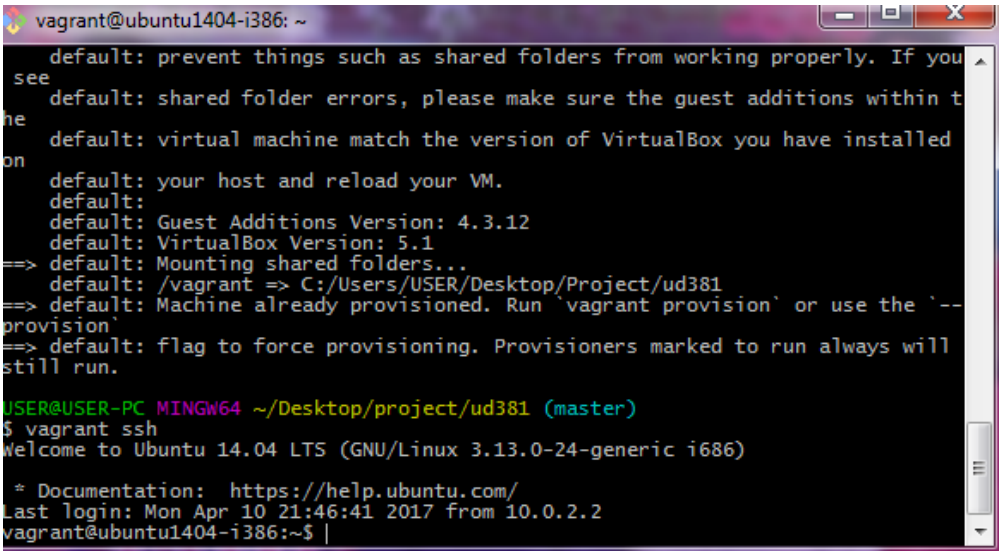

Fig 12 Vagrant SSH

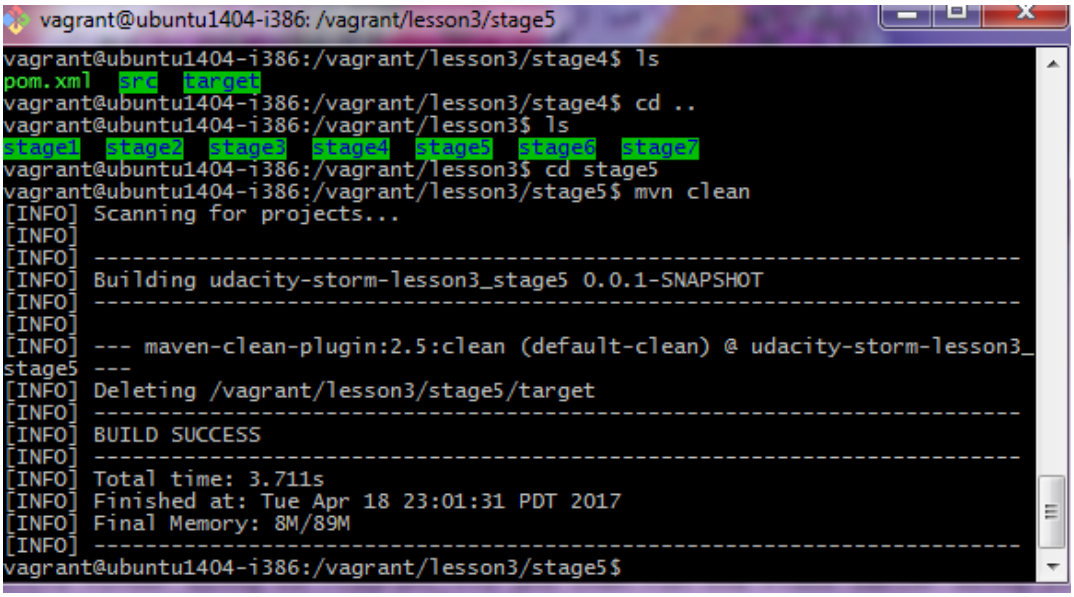

Fig 13 MVN Clean

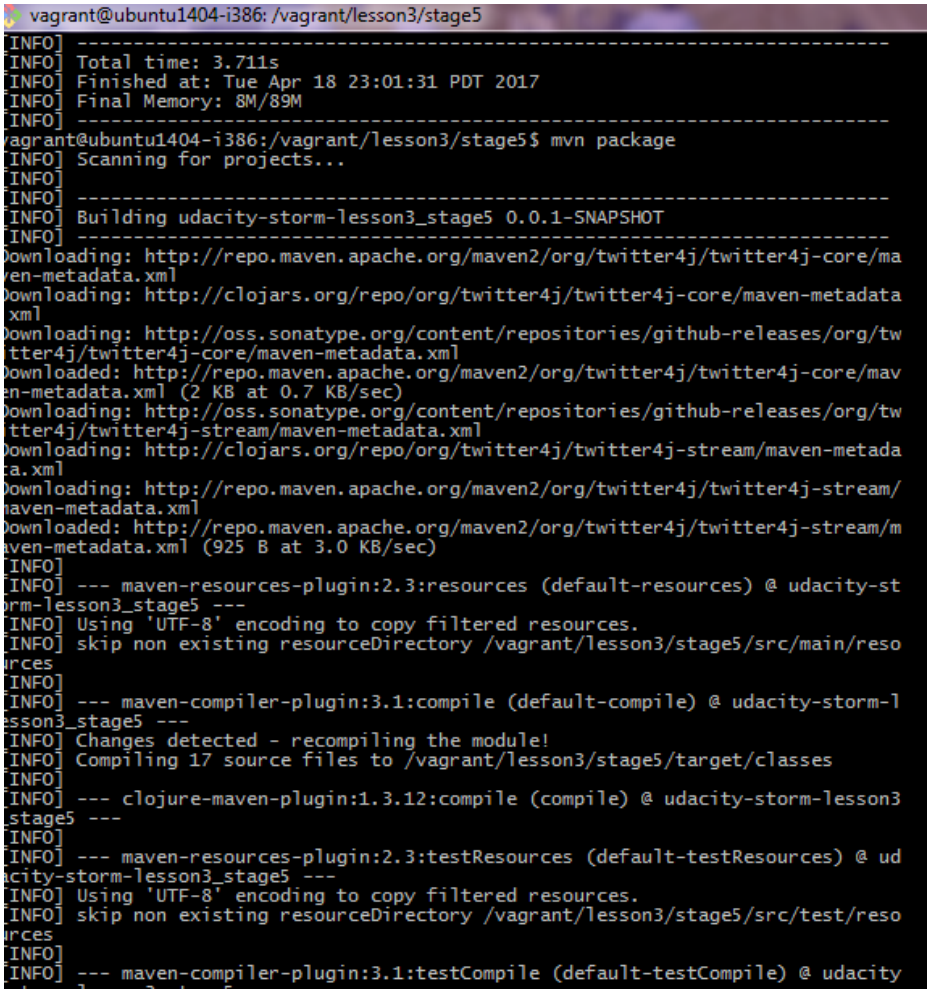

Fig 14 mvn package

TweetSpout.java | *TopNTweetTopology.java*

```
1  package udacity.storm;
2
3  import backtype.storm.Config;
4  import backtype.storm.LocalCluster;
5  import backtype.storm.StormSubmitter;
6  import backtype.storm.spout.SpoutOutputCollector;
7  import backtype.storm.task.OutputCollector;
8  import backtype.storm.task.TopologyContext;
9  import backtype.storm.testing.TestWordSpout;
10 import backtype.storm.topology.OutputFieldsDeclarer;
11 import backtype.storm.topology.TopologyBuilder;
12 import backtype.storm.topology.base.BaseRichSpout;
13 import backtype.storm.topology.base.BaseRichBolt;
14 import backtype.storm.tuple.Fields;
15 import backtype.storm.tuple.Tuple;
16 import backtype.storm.tuple.Values;
17 import backtype.storm.utils.Utils;
18
19 import udacity.storm.spout.RandomSentenceSpout;
20
21 class TopNTweetTopology
22 {
23   public static void main(String[] args) throws Exception
24   {
25     //Variable TOP_N number of words
26     int TOP_N = 10;
```

Fig 15 TopNTweetTopolgy Code

```
vagrant@ubuntu1404-i386:/vagrant$ cd viz
vagrant@ubuntu1404-i386:/vagrant/viz$ ls
app.py  db  dump.rdb  README.md  rt-provision-32.sh  static  templates
vagrant@ubuntu1404-i386:/vagrant/viz$ python app.py
 * Running on http://0.0.0.0:5000/
```

Fig 16 Flask Server

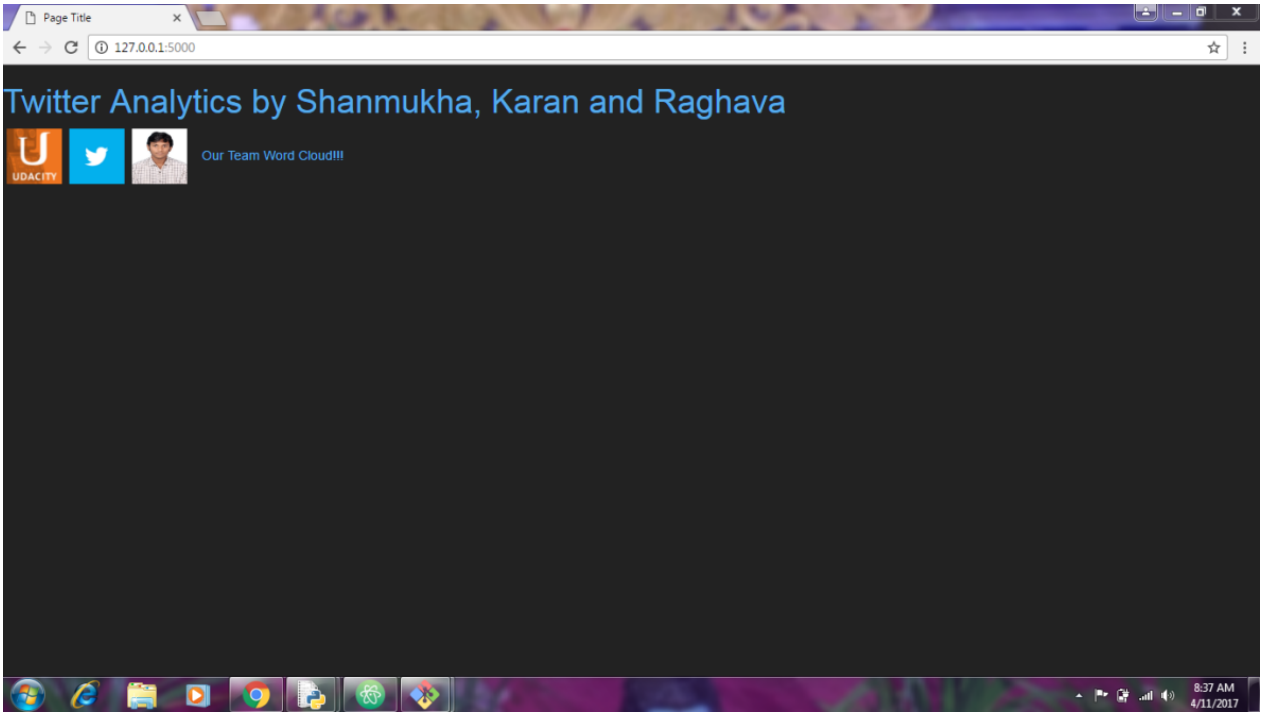

Fig 17 Application before running the topology

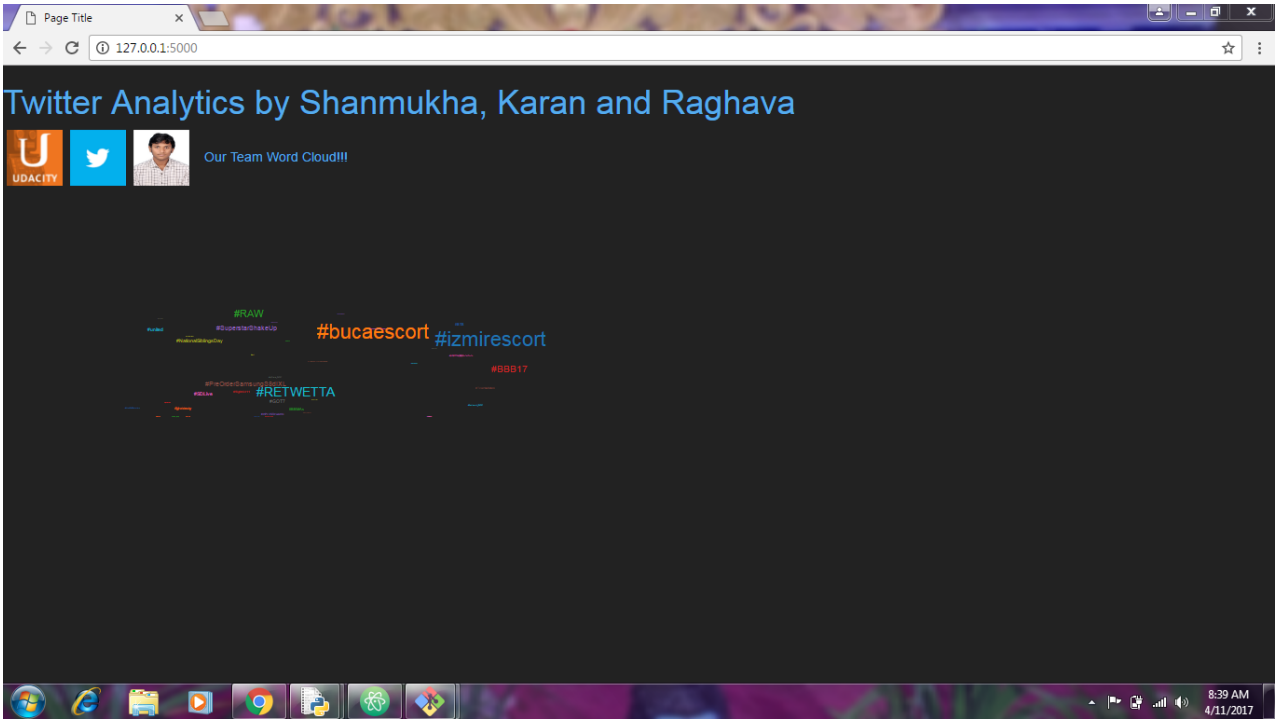

Fig 18 Output Word Cloud

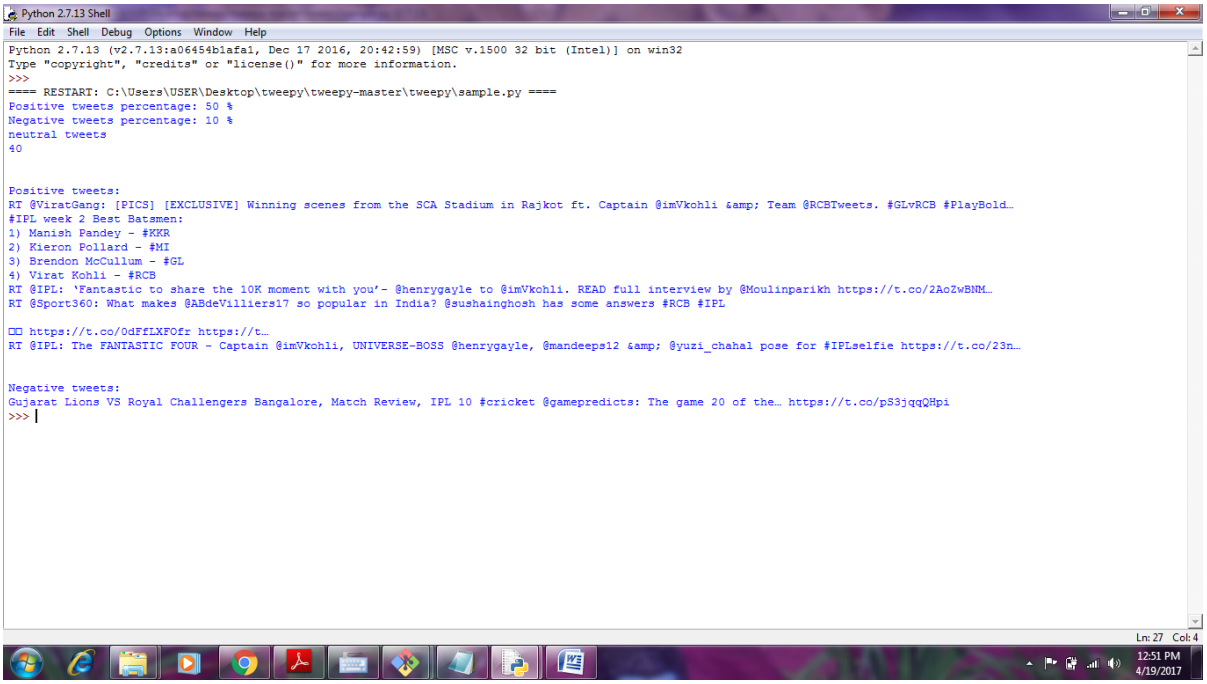

Fig 19 Sentiment Output

CHAPTER 8

CONCLUSIONS AND FUTURE ENHANCEMENTS

8.1 CONCLUSIONS

The main objective of this project is to provide an easy to understand visualization which is understood by the naïve as well as experienced users. It is an attempt to understand the real time environment where the tweets come in at unexpected times. It is an attempt to understand the sentiments of the people relevant to a particular topic and also visualize the most popular tweets.

This tool is extremely useful to people of all regions because it helps them visualize international trending topics as well. It helps eliminate the noise and advertisements while getting the trending topics. It also ensures that only relevant information which is currently trending is displayed and no old trending topics affect the visualization. The sentiment analysis performed is a way to integrate a few concepts of Natural Language Processing Toolkit (NLTK) to understand the opinions. It is performed using parsing of HTML and XML data. The software developed during this project was challenging yet fulfilling to complete. The final output was an energetic boost which represented the efforts of the team for the last couple of months.

8.2 FUTURE ENHANCEMENTS

This project can be further extended to perform analysis on sarcastic tweets as well. The Sarcasm detection is a tough puzzle that is yet to be solved by scientists and programmers. The main problem associated with the detection of sarcasm is the fact that sarcasm depends on the tone of the voice and not the words used in the sentence. Machines are finding it difficult to detect sarcasm because of this very reason. Multiple research teams at top technology companies have been struggling with this problem for a long time now. Despite using multiple neural networks and taking the support of SVMs these companies could not come up with accurate and precise results. The current accuracy of Sarcasm

detection is around 30% and this involves the use of powerful deep learning techniques which are patent protected. So, hopefully this project will be fulfilled by a SNIST student ten to twelve years from now using deep learning techniques.

CHAPTER 9

REFERENCES

The following sources played a crucial role in ensuring the completion of the project. We are extremely thankful to the creators of these resources.

1. Help related to Vagrant https://www.vagrantup.com/
2. Apache Storm http://storm.apache.org/
3. Apache Storm https://www.tutorialspoint.com/apache_storm/index.htm
4. Maven https://www.tutorialspoint.com/maven/maven_overview.htm
5. D3.js https://website.education.wisc.edu/~swu28/d3t/link.html
6. Beautiful Soup http://www.pythonforbeginners.com/python-on-the-web/web-scraping-with-beautifulsoup
7. Twitter https://dev.twitter.com/
8. Redis https://redis.io/
9. Github https://github.com/
10. Flask https://www.tutorialspoint.com/flask/
11. Flask http://flask.pocoo.org/docs/0.12/tutorial/
12. Paper related to Twitter Analysis http://dl.acm.org/citation.cfm?id=2512951