PROJECT REPORT

On

REALTIME SENTIMENT ANALYSIS OF TWITTER

Submitted in partial fulfillment for the award of the degree

Of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENIGNEERING

By

Anish Samir Mashankar (Reg. No.: 1031290474)

Meghna Saxena (Reg. No.: 1031210483)

Under the guidance of

Dr. E Poovammal
(Professor, Department of Computer Science and Engineering)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING AND TECHNOLOGY
SRM UNIVERSITY
(Under section 3 of UGC Act, 1956)

SRM Nagar, Kattankulathur- 603203
Kancheepuram District

APRIL 2016

BONAFIDE CERTIFICATE

Certified that this project report titled **"REALTIME SENTIMENT ANALYSIS"** is the bonafide work of ANISH SAMIR MASHANKAR (Reg. No: 1031210474) AND MEGHNA SAXENA (Reg No. 1031210483), who carried out the project under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion or any other candidate.

Signature of the Guide

Dr. E Poovammal

Professor

Department of Computer Science and Engineering

SRM University

Kattankulathur- 603203

Signature of the HOD

Dr. B. AMUTHA

Professor & Head

Department of Computer Science and Engineering

SRM University

Kattankulathur- 603203

DATE:

ABSTRACT

Chirrup is a tool developed to analyze public sentiment about a particular topic by mining user tweets from Twitter. Chirrup runs an Apache Storm topology for data analytics and displays graphical reports on a web page.

The data so mined is persisted in MongoDB as JSON Documents. The tool uses Flask web framework, written in python, for serving web pages that display statistics using interactive charts driven by aggregation queries on MongoDB.

The tool is designed to provide in depth user response and interaction insights about the queried topic. Chirrup, using an Apache Storm topology can be run on distributed systems and is thus capable of handling explosion of data input.

While sharing many of the same goals as other Twitter Analysis tools, Chirrup is driven by the motivation to become an effective tool for research, and market studies especially by students and independent researchers since it's easily deployable and open source.

We describe the motivation for Chirrup and the design and implementation of its analytics backend and visualization dashboard.

ACKNOWLEDGEMENT

We take this opportunity to offer our sincere and humble thanks to our Director Dr. C. MUTHAMIZHCHELVAN for providing us with an excellent atmosphere for doing research. We are also grateful and indebted to our Head of the Department Dr. B. AMUTHA, for her constant encouragement.

We thank our final year project Co-ordinator Dr. M Murali for helping us every step of the way.

We would like to express our deepest gratitude to our project guide Dr. E. POOVAMMAL without whose support and unfailing guidance and assistance this project could not have been possible. All through the work, in spite of her busy schedule, she has extended cheerful and cordial support to us in completing this project.

We also would like to thank all the staff members of the Computer Science and Engineering department for their assistance.

Authors

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Since the dawn of social networking, millions of people are posting their viewpoints on the Internet. Hence such websites have become a huge source of information, which could be used for various purposes such as data profiling, human behavioral analysis as well as market research inter alia. Social media platforms collectively play a big part in the data consumers create and share. Every 60 seconds on Facebook: 510 comments are posted, 293,000 statuses are updated, and 136,000 photos are uploaded [1]. A whopping 77 percent of B2C companies and 43 percent of B2B companies acquired customers from Facebook. [2]

Consider Twitter: It boasts of around 316 million users that tweet an average of 500,000 times per minute. That corresponds to 500 million tweets per day and around 200 billion tweets per year [3] . The statistics are astounding; now imagine the wonders we could do by putting this data to a meaningful and profitable use.

One such way is to analyze a general viewpoint by gauging public sentiment about any topic. This involves the use of natural language processing/text analysis/computational linguistics to decipher and extract information from source. This process is known as sentiment analysis (opinion mining).

This project implements the use of open source, distributed, fault tolerant tools to perform sentiment analysis of data mined from social networking platforms. This is done in a way to facilitate novice user interaction and use as it provides an easy to use user interface.

The naïve approach to such an application is to setup scripts that mine data from social networks, wrangle the data, perform sentiment analysis and display results on a webserver. The problem with such an approach is that it does not guarantee exactly once processing, fault tolerance and consistency.

1.2 Objective

The project uses Apache Storm[4] for performing the data mining, wrangling and analysis. The sentiment analysis is done by using the AFINN[5] algorithm. The output from the Apache Storm topology is then fed into the stream processor which is powered by Apache Kafka[6]. A listener to the Kafka topic feeds the incoming stream into the MongoDB[7] for persistent storage and data serving. The frontend user interface utilizes the Flask[8] micro-web app framework written in Python.

The fault tolerance is made possible by the robustness of these tools (Apache Storm, Apache Kafka, and MongoDB) and the reason for their choice is explained in later section.

1.3 Structure of the Report

We begin with reviewing the literature we went through while sketching the project outline. Here we discuss the motivation behind our project and existing frameworks. We also discuss the algorithm used for sentiment analysis in detail and the wordlist on which it is based.

Then we put forward the system requirements for the deployment of Chirrup and explain the installation procedure step by step. The next chapter looks at the various tools used and the reasons why they were chosen for the project.

Application overview talks about the architecture and the various decisions taken. It gives an overall idea of the backend and frontend structure.

In the later chapters, we discuss about data wrangling, analysis and persistence. Finally we explain the frontend components and the design of the UI.

We conclude by describing performance of the tool setup for a single node. Performance characteristics such as data size explosion, time to retrieve and process data etc. are discussed.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

To analyze text sentiments, there are various different tools available on the internet as well as downloadable applications. However, not all of them had a small processor and memory footprint. Some minimalist algorithms such as Naive Bayes were much successful than the complex ones such as Wordnets and Decision trees. However, dictionary methods are considered best choices as they have fast computational times and hence don't fall much behind than the incoming stream. The choice of dictionary and caching methods depend on how accurate the trained data inside them is. By doing extensive research on various such tools, we selected AFINN list to be the model of sentiment classification.

Here we discuss why we chose AFINN as the wordlist to score the data mined from twitter.

2.2 AFINN Approach: Wordlist For Sentiment Analysis

The reason we chose AFINN is because we had to deal with social networking data which includes elements like modern slangs, abbreviations and emoticons and this word list takes into account all the aforementioned elements.

When the author measured the performance of application of word lists with a correlation coefficient. He found that the list and ANEW were quite ahead of the word lists in General Inquirer and OpinionFinder.

He also tried SentiStrength Web service sentiment analyzer on the 1,000 Mislove tweets. This is not just a simple word list but is a program that has, e.g., handling of emoticons, negations and spelling. This Web service showed to be the best: slightly ahead of this list and ANEW.

The version termed AFINN-96 distributed on the Internet1 has 1468 different words, including a few phrases. The newest version has 2477 unique words.

In some cases he even used Twitter to determine in which contexts the word appeared. He also used the Microsoft Web n-gram similarity Web service ("Clustering words based on context similarity") to discover relevant words. Words such as "surprise"—with high arousal but with variable sentiment—were not included in the word list.

Even though General Inquirer and OpinionFinder have the largest word lists, they do not perform as good as SentiStrength, AFINN and ANEW for sentiment strength detection in microblog posting. The two former lists both score words on polarity rather than strength and it could explain the difference in performance. The list comes slightly ahead of ANEW in Twitter sentiment analysis, thus we decided to work with AFINN.

CHAPTER 3

SYSTEM REQUIREMENTS

3.1 Introduction

The objective of developing the system was to make it runnable on all specifications of computer. However, due to the limits of technology, we have certain minimal requirements for the tool to run. These requirements however are only recommended and not to be taken very strictly. We have tested the system completely with all integrations. For the technology stack we are going to start from the lowest level: the system, then we will move onto the data engineering section and finally discuss how to make the web application runnable. This chapter will also guide in the installation and setting up of the entire environment.

3.2 System

The machine configurations that were active in the development of this tool are as follows:

- Operating System: Ubuntu Linux 15.10
- Processor: Intel Core i5 – 6^{th} Generation
- RAM: 8GB
- Softwares installed:
    - Python 2.7.10
    - Java 1.8
    - Maven 3.2

However, we recommend all servers to have the following minimum configurations:

- Any processor with speeds at minimum of 2Ghz
- 512MB of available RAM
- Any Unix based operating system
- The following softwares:
    - Python 2.7+
    - Java 1.8+
    - Maven 3+

3.3 Application Stack

The backbone of our data engineering and analysis is done using various tools:

- Apache Zookeeper [9]: The cluster management system used by Apache Storm
- Apache Storm: The backbone of data analytics in the tool
- MongoDB: The persistant data store
- The following Python modules are required to run the web application and perform analysis on Apache Storm:
    - flask: The webserver
    - afinn: The afinn analysis library in Python. This library is the implementation of our sentiment analysis algorithm
    - pymongo: This module provides an API for connecting to MongoDB using the python programming language.

3.4 Setup Instructions For Chirrup

The following setup instruction are valid for any Ubuntu based Linux distribution.

Step 1: Install zookeeper, java, and maven

```
$ sudo apt-get install zookeeper openjdk-8-jdk maven
```

Step 2: Download the Apache Storm binary from their website, extract the tar, and add the bin folder to the `$PATH` variable.

```
$ wget http://mirror.fibergrid.in/apache/storm/apache-storm-1.0.0/apache-storm-1.0.0.tar.gz
```

```
$ tar –xvzf apache-storm-1.0.0.tar.gz
```

```
$ echo ‘$PATH:apache-storm-1.0.0/bin’ >> ~/.bashrc
```

Step 3: Install MongoDB. You can always get it from the official Ubuntu repository

```
$ sudo apt-get install mongodb
```

Step 4: Install the required python packages

```
$ sudo apt-get install python-pip #in case there is no pip
```

```
$ sudo pip install flask afinn pymongo
```

3.5 Running Chirrup

Step 1: Clone the project from github

```
$ git clone https://github.com/anishmashankar/chirrup
```

Step 2: Build the storm project

```
$ cd chirrup
```

```
$ cd storm-project
```

```
$ mvn package
```

The build success message should look something like this:

```
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------
[INFO] Total time: 25.087 s
[INFO] Finished at: 2016-04-27T20:37:00+05:30
[INFO] Final Memory: 27M/756M
[INFO] ------------------------------------------------------
```

Now you can see a new folder created inside the storm-project called “target”. This folder contains the uber-jar, which is going to have all the required classes and executables to run the storm topology.

Step 3: Run the storm topology:

```
$ storm jar target/storm-*-jar-with-dependencies.jar org.rtsa.storm.TheTopology
```

Now you can see a list of logs building up, data being mined from twitter and the console logs all the tweets that it has successfully processed.

```
...... truncated output ......

13944 [cluster-ClusterId{value='5720d778eeb32721708dbeec',
description='null'}-localhost:27017] INFO  o.m.d.cluster -
Monitor thread successfully connected to server with
description ServerDescription{address=localhost:27017,
type=STANDALONE, state=CONNECTED, ok=true,
version=ServerVersion{versionList=[2, 6, 10]},
minWireVersion=0, maxWireVersion=2, maxDocumentSize=16777216,
roundTripTimeNanos=752531}

13948 [Thread-10-output-bolt] INFO  o.m.d.connection - Opened
connection [connectionId{localValue:6, serverValue:212}] to
localhost:27017

13950 [Thread-10-output-bolt] INFO  o.r.s.OutputBolt - {
"tweet-id": 725342503499493377, "tweet-text":
@DrunkVinodMehta come on grow up sir... Bcoz Italian court's
verdict is just out... #SwamyRattlesSonia @Swamy39,
"sentiment": 0.0, "country": "Not Available", "hashtags":
["SwamyRattlesSonia"] }

........truncated output ......
```

You can start seeing results on the web application now. To run the web application, follow these steps:

```
$ cd ${CHIRRUP_HOME}/frontend

$ python front.py

* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 726-441-636
```

The web application is served at default port of 5000 on the address of the server. If running locally, the address to access the web application is http://localhost:5000 as shown in Figure 1.

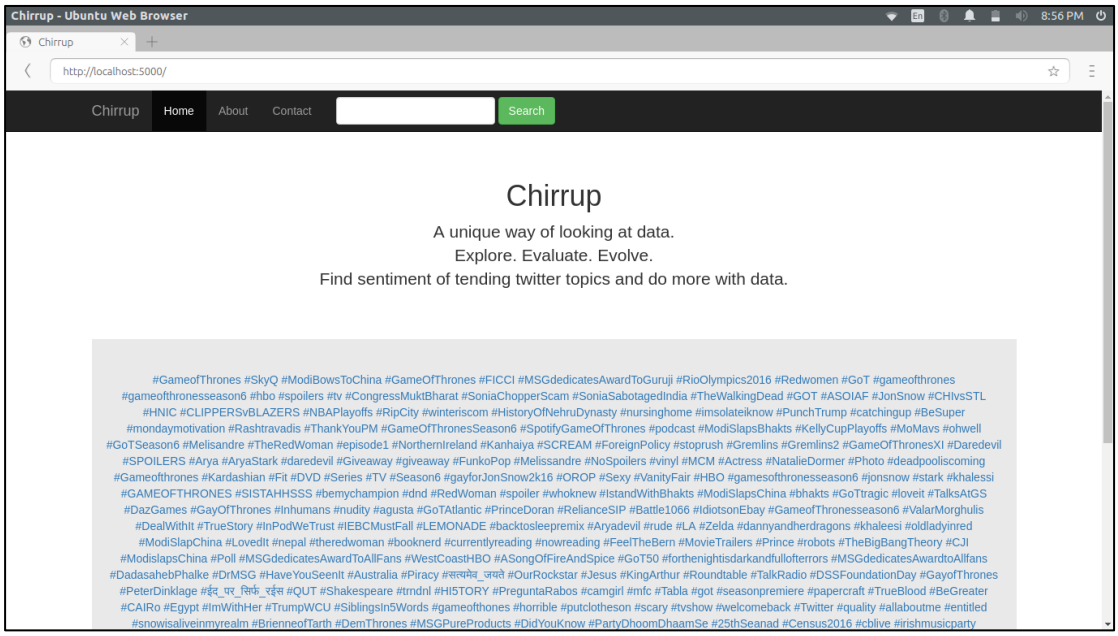

Figure 1 Chirrup live on http://localhost:5000

CHAPTER 4

TOOLS USED

All projects have supporting tools and frameworks on which they depend upon. Similarly, we had to select a set of tools to implement this application. After thorough research, we decided on the particular set of tools for our project. We discus the various tools, their features, usage in our project and the respective reason why they were chosen.

4.1 Apache Storm

Apache Storm is the backbone of analytics in our project. In this section, we introduce the basic terminology, and the reason for choosing the tool over its alternatives.

4.1.1 Overview

Apache Storm is an open source distributed real time stream processing computation system. Storm makes it easy to reliably process unbounded streams of data and offers various methods of dealing with architectural faults. The tool integrates easily with existing systems and is a great choice for implementing new stream processing systems.

4.1.2 Terminology

4.1.2.1 Topology

A topology contains the logic for the real time application. It consists of interconnected bolts and spouts that exchange data using tuples.

4.1.2.2 Spout

A spout is a source of stream. A spout will read data from an external source such as a Message stream or a streaming API and forward the data in the form of tuples to the adjacent bolt.

4.1.2.3 Bolt

A bolt is a computational unit of a topology. All the transformations on data are done inside a bolt. The number of bolts in the cluster also denotes the level of granularity of computation involved in the process.

4.1.3 Reason

There were several alternatives to Apache Storm that offered near about the same level of reliability. Some of these tools include: Apache Spark, Apache Flink and Apache Samza. During the incubation of this project, Apache Flink was still under nightly development and we needed something more robust and stable to power our data. Apache Samza did not offer a very good integration with the Web API. In fact, it totally depended on Apache Kafka project for Data input and Apache Hadoop as a data store. The toughest competitor was Apache Spark. Spark is being as actively developed as Storm. It provided the same level of robustness and performance as Apache Storm. It also had well developed Machine Learning libraries, which are not present in the distribution of Apache Storm. The reason that Apache Spark was beaten in this battle was at the level of modularity Storm offered. Also, running map reduce jobs on continuous stream inputs would be computationally very costly. Hence, we chose Apache Storm as it finally met all the requirements we had thought of during the planning phase of the project.

4.2 Apache Kafka

Apache Kafka is a fast, scalable, distributed and durable message communication system. Kafka takes care of storing messages and making them available in the correct order. The developers of the project call it a "distributed commit log".

4.2.1 Overview

Apache Kafka is the message broker in our application stack. It sends forms a link between the Apache Storm Output stream and the MongoDB. The Storm topology produces a message for every output and Kafka then conveys these messages to be written to the MongoDB collection.

4.2.2 Terminology

4.2.2.1 Broker

A broker is a Message passer that is responsible for accepting messages from a producer and sending it to a consumer.

4.2.2.2 Producer

An entity that sends messages to Kafka broker is called a producer.

4.2.2.3 Consumer

A consumer is an entity that receives messages from a Kafka broker and performs processing on them.

4.2.3 Reason

There are several message brokers available, which are open source and actively developed. Some of them include ActiveMQ, RabbitMQ and Kestrel. RabbitMQ and Kestrel do not offer the level of fault tolerance as ActiveMQ and Kafka do. So, it was down to the two players ActiveMQ and Apache Kafka. ActiveMQ has been a big player in the industry for several years. It provides high replication, and continuous producer connectivity. However, ActiveMQ did not have the feature for consumer tracking and uses files to maintain state information about consumers and messages. Kafka does a wonderful job on all places ActiveMQ cannot and even more. Kafka being a robust system by itself, maintains state information in another industry standard key-value store called Zookeeper. Zookeeper is a distributed cluster management system that offers reliable KV stores and worker statistics. Kafka also enables consumer tracking so that if consumers go down and come up later, they can start resuming from the point where they left off. No messages would be lost in the restart process. So, without doubt Kafka truly offers a non-blocking message queue system. It is obvious that Twitter is going to produce higher volume of data for a unit time than the writes to MongoDB. Hence, choice of a non blocking queue was must for this application.

4.3 MONGODB

MongoDB is an open source, cross platform document database. Its is classified as NoSQL database and prefers JSON-like documents which have dynamic schema over traditional relational database structure.

4.3.1 Overview

MongoDB serves as the persistent storage for the sentiment data mined from twitter. It is the analyzed and final dataset used for querying by the web application.

4.3.2 Terminology

4.3.2.1 BSON

BSON is a binary representation of JSON documents, though it contains more data types than JSON.

4.3.2.2 Document

A record in MongoDB, which is a data structure, composed of field and value pairs. The Document stores records as BSON documents. The values of fields may include other documents, arrays, and arrays of documents.

4.3.2.3 Collection

Documents are stored in collections. These collections are the same as tables in relational databases.

4.3.2.3 Linking

Corresponds to joins in the relational database system.

4.3.2.4 Database

The collections are stored in Database. The name of the database in this project is **Chirrup**.

4.3.2.5 Sharding

MongoDB uses horizontal scaling. The user chooses a shard key, which determines how the data in a collection will be distributed. The data is split into ranges (based on the shard key) and distributed across multiple shards. (A shard is a master with one or

more slaves.). Alternatively, the shard key can be hashed to map to a shard – enabling an even data distribution.

4.3.3 Reason

MongoDB is a NoSQL database structure and hence makes high availability in an unreliable environment easier since a real-time system grows exponentially. It offers features like ad hoc queries, replication and sharding. It is best suited for data mining as it is optimized for heavy inserts. It makes working over a distributed system not only possible but easier as load balancing is nicely optimized. It can be used as a file system also, taking advantage of load balancing and data replication features over multiple machines for storing files (Grid File System).

4.4 Flask

Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine. It is BSD licensed.

Flask is a "micro" framework because it leaves the decision to choose tools or libraries on the developer. The developer does not have to use a particular tool or library. Flask has no database abstraction layer, form validation or any other such component for which other third party libraries exist. Instead, Flask supports extensions that can add features to the application as if they were implemented in Flask itself. There are extensions existing for database integration, object-relational mappers, form validation, open authentication technologies and several other framework related tools.

4.4.1 Overview

Flask serves as the web framework for the application. It integrates all the miscellaneous backend tools with the HTML5 frontend.

4.4.2 Thread-Locals in Flask

Flask uses thread-local objects internally. This is done to avoid passing objects around from functions within a request in order to stay thread safe. This requires a valid request context for dependency injection or when attempting to reuse code, which uses a value pegged to the request. These thread-locals are mentioned in the code and documentation.

4.4.3 Reason

Since Flask is a micro framework and does not force the developer to choose a particular tool, it was the best choice for us. We are working with several different tools and libraries like Apache Storm, Apache Kafka and MongoDB so no other platform offered the same versatility.

CHAPTER 5

APPLICATION OVERVIEW

5.1 Introduction

One of the initial design decisions that we had to make was to come up with an application architecture that was optimized. It included devising a data flow model over the application and maintain an optimal granularity of operations.

We discussed over several models and finally came up with an architectural plan that is discussed in the succeeding sections.

5.2 Application Architecture

There are two components of this application, as shown in Figure 2. One is the web portal and other is a Storm topology developed for data analysis. A user inputs a topic from the web, which is communicated to a Storm *spout* using the Kafka broker. A consumer is present at the spout of the Storm topology which filters the incoming Twitter Stream based on the topic input from the web portal.

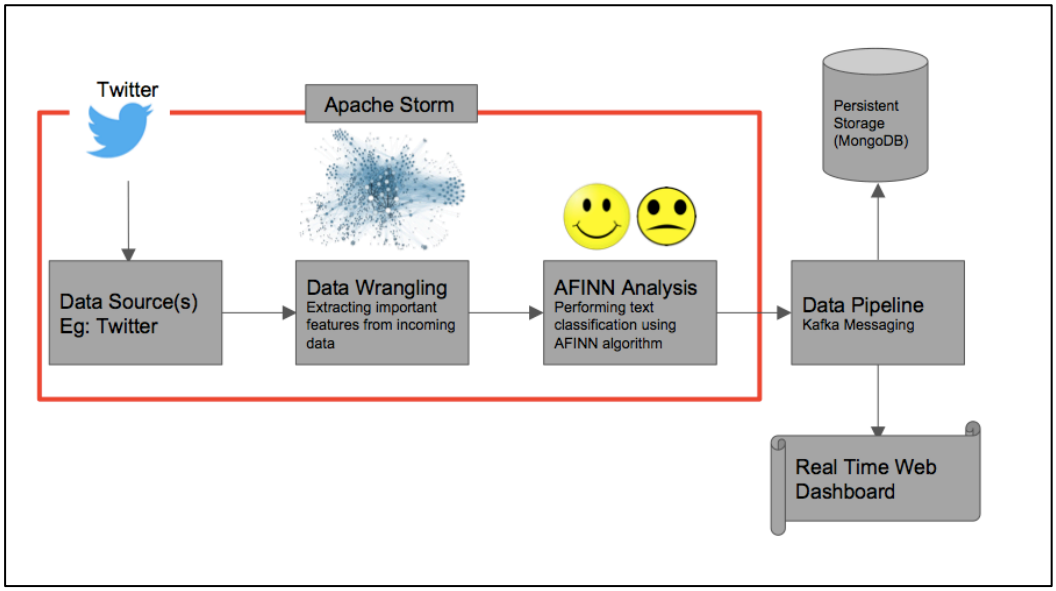

Figure 2 Chirrup Architecture

5.2.1 Backend

The Storm topology as shown in Figure 3, performs the following operations on the data:

1 The Twitter API sends Tweets as JSON Documents. These JSON documents contains excessive amount of useful information (like hashtags, geo-location,) about the Tweet being evaluated. The data from the tweet is sent without modification to the Data Wrangling bolt.

2 The Data Wrangling bolt is responsible for converting the incoming JSON into Tuples. The tuple has following required information about the tweet:

- 2.1 Array of hashtags
- 2.2 The country from which the Tweet came.
- 2.3 The Tweet text
- 2.4 Tweet ID

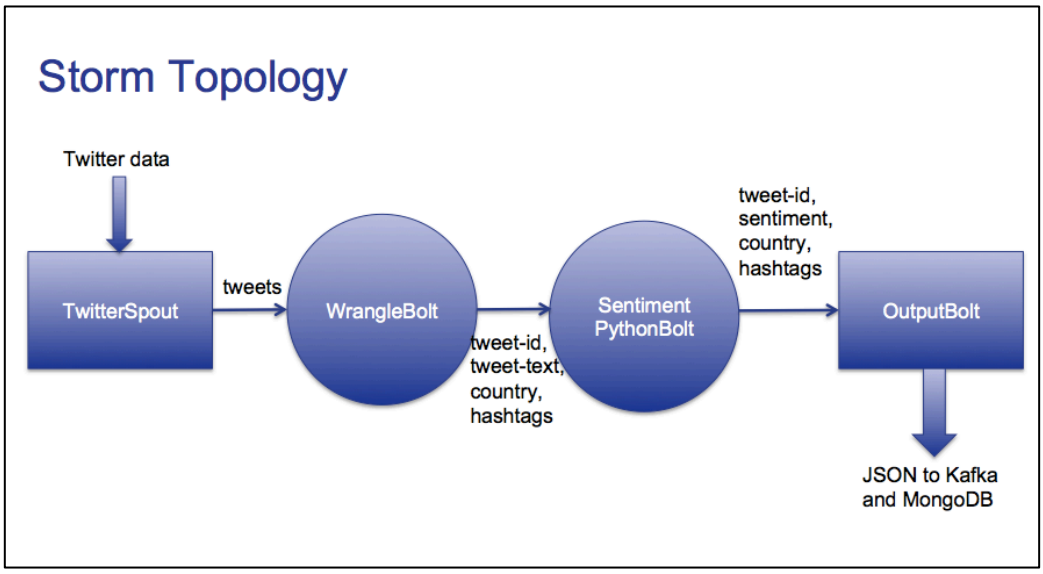

Figure 3 Storm Topology

3 The sentiment bolt extracts Tweet text from the incoming tuple, and applies the AFINN algorithm. The AFINN Python Module was used for this step, as it is an optimized implementation of this algorithm. After the tweet text is scored, the tuple is now transformed to have the following information:

- 3.1 Array of hashtags
- 3.2 Country of the tweet

3.3 Sentiment of the tweet

3.4 Tweet ID

4 This tuple is now forwarded to the Output Bolt

5 The responsibility of the Output Bolt is to write the data to the Kafka broker, which in turn feeds an event stream consumed by the web application and MongoDB.

5.2.2 Frontend

The user-facing component, the front end of this package is a web application. The web application has three major components:

1. The Home Page
    a. The banking page. Offers options to search for hashtags or select one.
2. The per hashtag analysis page
    a. Displays a dashboard of statistics derived from the tweet data.
3. 404 – not found page
    a. Takes care of exceptions such as hashtag not found or spelling error.

Each web page is served as a Flask end point.

CHAPTER 6

DATA MINING AND PREPROCESSING

The first step of our application is to collect data from Twitter. We had to ensure that the stream from Twitter was dealt with utmost care and we don't lose data on the way. To deal with this problem, we used a Java API for Twitter called Twitter4J[10].

Twitter4J is a all compatible with the latest version of Twitter API[11] client for java and also provides in built functions for dealing with OAUTH and gzip compression support. Twitter4J requires no external dependency and has a low JAR footprint on the final executable. In the following sections, we will explain how we have implemented the StatusListener interface and batched stream content inside our Storm topology. Later, we will discuss about how we have wrangled the data for sentiment analysis by removing unwanted content in the tweet such as hyperlinks, punctuations and other noisy data.

6.1 Mining Twitter Data

As mentioned earlier, we use the Twitter4J's StatusListener interface to read the incoming status stream. For that, we have first created an application on Twitter to get OAUTH credentials so that we can use the API. These credentials are configurable parameters in our final package. A user can enter his Twitter API keys in the credentials variables. An example configuration in the TheTopology.java file is:

```
String custKey = "D90JVJkIzMYBSU4K1zCvP2FXE";
String custSecret =
"r1GWAmHsTtwuFzQFOdGj0Ad7jBq2CynlwRyNiIEvbQpfzOOsTG";
String accessKey = "612666274-
cQQ4ICpXco0pSQU3Zl7IprEuF9jhgaSE2gXhQzkc";
String accessSecret =
"wFytlnD0xolGvOrqGGCm80ZiKcnsNsj3XwnbEbtO1ukz4";
```

The user will just have to input their unique customer key, customer secret token and access key (in place of the values in the code) to set up the Twitter4J API so that it can start streaming tweets.

The Twitter4J library uses these credentials to talk to the API. It subscribes to the Twitter stream in the TwitterSpout.java code using:

```
TwitterStream twitterStream = new TwitterStreamFactory(
                new
ConfigurationBuilder().setJSONStoreEnabled(true).build())
                .getInstance();
twitterStream.addListener(listener);
twitterStream.setOAuthConsumer(consumerKey, consumerSecret);
AccessToken token = new AccessToken(accessToken,
accessTokenSecret);
twitterStream.setOAuthAccessToken(token);
if (keyWords.length == 0) {
     twitterStream.sample();
}
else {
    FilterQuery query = new FilterQuery().track(keyWords);
    twitterStream.filter(query);
}
```

In second line of code, the function twitterstream.addListener takes in a StatusListner object. A StatusListener object is like an event listener. For every status coming in, it executes a function called onStatus(). For the purpose of this project, we have modified that function to only accept tweets that have text and are written in English language. A demonstration of that function is expressed as:

```
public void onStatus(Status status) {
if( status.getText().length() != 0 &&
status.getLang().equals("en")) {
        queue.offer(status);
    }
}
```

A Status objects holds all data that is provided by Twitter on a tweet when we query their API. The functions, getText() returns the actual text inside the tweet and the function getLang() returns a two letter ISO language code. We draw conditional checks on these two functions and add the status to a blocking queue.

In a Storm spout class, the function nextTuple() decides what data to send to the connecting Bolts. The nextTuple() function is implemented by us as:

```
public void nextTuple() {
Status ret = queue.poll();
if (ret == null) {
Utils.sleep(50);
}
else {
_collector.emit(new Values(ret));
}}
```

The code clearly mentions that when the collector receives an acknowledgement for the previous tuple, a status is polled from the blocking queue and then is sent to the collector to be emitted to the adjacent Bolts in the topology.

6.2 Wrangling Data

Data Wrangling is one of the most difficult tasks when it comes to running a Big Data analytics applications is to devise efficient methods to clean data out of the noise. To do this task in an efficient way, we created a Storm Bolt called WrangleBolt.java. This java class deals with the incoming Status object, extracts useful information from the Status, and then creates intermediate data objects to facilitate analysis and then forward it to subsequent Bolt for analysis.

After fetching the data, we have to extract useful text from the data that is free from URLs. The mentioned code snippet removes URL from the tweet text.

```
private String filterOutURLFromTweet(final Status status) {
   final String tweet = status.getText();
   final URLEntity[] urlEntities =    status.getURLEntities();
   int startOfURL;
   int endOfURL;
   String truncatedTweet = "";
        for(final URLEntity urlEntity: urlEntities){
      startOfURL = urlEntity.getStart();
                  endOfURL = urlEntity.getEnd();
                        truncatedTweet += tweet.substring(0,
      startOfURL) + tweet.substring(endOfURL);
        }
    return truncatedTweet;
 }
```

The function filterOutURLFromTweet takes a Status object as parameter and locates the start and end pointer of the URL text in the tweet text. This is done by using the getStart() and getEnd() function calls. Once the URL position in the tweet text is obtained, the URL is skipped while retaining the remaining text in the truncated tweet (as done in the for loop). Then the noiseless tweet is returned to the calling function.

The execute() function is called everytime a new tuple comes to the WrangleBolt. This function in turn calls the filterOutURLFromTweet() function to remove noisy data as illustrated in code.

```
public void execute(Tuple input) {
        Long tweetId;
        String tweetText, tweetCountry;
        HashtagEntity hashtagEntities[];
        ArrayList<String> hashtags =  new ArrayList<String>();
        Status tweet = (Status) input.getValueByField("tweet");

        tweetId = tweet.getId();
        tweetText = filterOutURLFromTweet(tweet);
        Place tweetPlace = tweet.getPlace();
        String originalTweetText = tweet.getText();
        Long tweetDate = tweet.getCreatedAt().getTime();
        User user = tweet.getUser();
        if (tweetPlace == null) {
            tweetCountry = "Not Available";
        }
        else tweetCountry = tweetPlace.getCountry();
        //tweetCountry="my country";
        hashtagEntities = tweet.getHashtagEntities();
        if (hashtagEntities.length == 0) {
            hashtags.add("none");
            collector.ack(input);
        }
        else {
            for (HashtagEntity hashtagEntity : hashtagEntities){
                hashtags.add(hashtagEntity.getText());
            }
            Values values = new Values(tweetId, tweetDate,
tweetText, tweetCountry, hashtags, originalTweetText);
            collector.emit(values);
        }
    }
```

The execute() method also takes care of exceptions (such as null values) and extracts data such as country and hashtags. After doing all this fucking preprocessing, the data entities are packaged into a Values object, which is basically an abstraction for a Tuple in Storm. Finally, the Bolt's collector emits this Values object. This tuple is then passed to the Analysis bolt to perform the AFINN Sentiment Analysis.

CHAPTER 7

SENTIMENT ANALYSIS

7.1 Introduction

The AFINN algorithm is specifically derived from twitter data and hence it is especially designed to take into account words, expressions, modern slangs (LOL, OMG, WTF etc.) and emoticons. And thus it can parse them relevantly without losing important information. These aspects greatly enhance the algorithm's ability to assign proper polarity of sentiment to the mined data. Furthermore, the polarity range is wide enough to take numerous fluctuations into account.

The AFINN library provides a set of implemented functions to process and score the incoming wrangled datum. This library was developed specifically to handle distributed data from streams. It is an efficient implementation of word mappings in python, which makes it suitable for use in this project.

Example Usage of Library in Python Interpreter:

```
>>> afinn = Afinn(emoticons=True)
>>> afinn.score('I saw that yesterday :)')
2.0
```

The above code calls the Afinn library constructor and sets it to accept emoticons. Note that this configuration is set for the afinn object. The object then calls the library function score() with a string value containing an emoticon as an argument. The function scores the neutral sentence (neutral as it contains all neutral words) as 2.0 based only on the emoticon representing happiness.

7.2 Implementation

Apache Storm provides an interface that allows us to write Bolts in different programming languages. The bolts and JVM interact with each other using standard

input/output. We invoke the Python Bolt: SentBoltPython() from within the JVM using the code:

```
public SentBoltPython(String resourceFile)
{
     super("python",resourceFile);
}
public void declareOutputFields(OutputFieldsDeclarer
                                    declarer)
{
  declarer.declare(new Fields("tweet-id", "tweet-date",
  "sentiment","country","hashtags","tweet-text"));
}
```

The constructor for the bolt takes in the python resource file, which is a python script, as an argument. It should be ensured that the python file is executable.

The code below represents the analysis.py python script that we have passed to the constructor of SentBoltPython()

```
import storm
from afinn import Afinn
afinn = Afinn(emoticons=True)
class SentimentAnalysisBolt(storm.BasicBolt):
    def process(self,tuple):
        tweetid = tuple.values[0]
        tweet_date = tuple.values[1]
        tweettext = tuple.values[2]
        country = tuple.values[3]
        hashtags = tuple.values[4]
        tweet_text = tuple.values[5]
        score = afinn.score(tweettext)
        storm.emit([tweetid,    tweet_date,    score,
                    country, hashtags, tweet_text])
SentimentAnalysisBolt().run()
```

The python script imports base libraries storm and afinn. Then, an instance of the afinn library is created and configured to support emoticons. Now we extend the BasicBolt of the Storm's Python library and implement the process() method. In the first few steps, we extract the values from tuple passing some of the values without modification. While the tweet text string is scored by the imported afinn libraries's score() method. Then the score is passed along with the other values to the next Bolt.

CHAPTER 8

PIPELINING

8.1 Introduction

The data analysed so far, now has to be persisted in a database. As we mentioned earlier, our datastore is MongoDB. The OutputBolt takes care of incoming data and builds an insert query using the MongoDB Java API.

8.2 Persisting Output Data From Storm

The OutputBolt module of storm handles the incoming data and persists it on the MongoDB database.

The execute() function of the OutputBolt takes the input tuple as an argument and then modifies the data (tweet-id, hashtags etc) to make it suitable for MongoDB storage.

Then we set up the database client: localhost in our case, but it can contain a MongoDB connection string as the parameter. In our case, we decided to add our data into a database “chirrup”. The collection that will store all the processed tweets and their sentiments is called “tweets”. The “tweets” collection forms the basis for all analytical queries and gets from the datastore.

```
public void prepare(Map stormConf, TopologyContext
context, OutputCollector collector)
{
        this.collector = collector;
        mongoClient = new MongoClient("localhost");
        database = mongoClient.getDatabase("chirrup");
        collection = database.getCollection("tweets");
}
```

In the implementation part, we first initialize a connection object using the MongoClient API. We call this object “mongoClient”. Later, we connect to the database “chirrup” in the scope of that connection. Finally, we fetch the collection “tweets” to which we are going to write the output data. In the JAVA API for

MongoDB, we first build a Document object that is a set of key-value pairs, where a key is a string and value can be any other Document object. We append these objects together to form one complete document.

Finally, we insert the document into the collection by calling the function insert() on the collection object. The insert() method takes in a Database object as a parameter and executes regular MongoDB insert. Finally, for logging purposes, we build a JSON-like string to write to the Logger of our OutputBolt. This is done for debugging purposes and checking for data loss or workarounds.

```
public void execute(Tuple input) {
     Long tweetId = (Long)input.getValueByField("tweet-id");
     double sentiment = (Double)
      input.getValueByField("sentiment");
     Long tweetDateInput = (Long)
     input.getValueByField("tweet-date");
     Date tweetDate = new Date(tweetDateInput);
     String tweetText = (String)
     input.getValueByField("tweet-text");
     String country = (String)
     input.getValueByField("country");
     ArrayList<String> hashtags = (ArrayList<String>)
     input.getValueByField("hashtags");

        Document doc = new
        Document("tweetid",tweetId.toString())
                .append("timestamp", tweetDate)
                .append("sentiment", sentiment)
                .append("tweetText", tweetText)
                .append("country", country.toString())
                .append("hashtags", hashtags);
        collection.insertOne(doc);
        String ret = "{ "+
            "\"tweet-id\": " + tweetId.toString() + ", " +
            "\"tweetText\": " + tweetText +" , " +
            "\"sentiment\": " +
            Double.toString(sentiment)+","+
            "\"country\": " + "\""+ country + "\""+ ", " +
            "\"hashtags\": " + hashtags.toString() +
                " }";
        logger.info(ret);
        collector.ack(input);
    }
```

8.3 Document Structure in MongoDB

To make the document structure clear, we will now explain how the data looks like inside the MongoDB database's collection. To check the same, we fire up the Mongo shell in the Terminal by issuing "mongo" command. And finally, make a query that looks like :

```
$ mongo
MongoDB shell version: 2.6.10
connecting to: test
> use chirrup
switched to db chirrup
> db.tweets.findOne({"sentiment":{$gt: 3}})
{
"_id" : ObjectId("571edf0d65b98f1b4f83e16e"),
"tweetid" : "724800890973097986",
"timestamp" : ISODate("2016-04-26T03:22:53Z"),
"sentiment" : 4,
"tweetText" : "The #GameOfThrones cast made up lyric's
to the show's opening theme and, as predicted, it's
amazing.  https://t.co/nBbvwkd3La via @MTV",
"country" : "United States",
"hashtags" : [
"GameOfThrones"
]
}
```

CHAPTER 9

FRONTEND

9.1 Introduction

The user-facing component, the front end of this package is a web application served by default on the port 5000 of the server. The web application has three major components:

4. The Home Page
5. The per hashtag analysis page
6. 404 – not found page

Each web page is served as a Flask end point. In the further sections, we are going to illustrate how each endpoint works and how it renders web pages to the user. The webpages are designed using the responsive Bootstrap[12] framework.

9.2 The Home Page

The home page is the first web page displayed to the user when he hits the root endpoint of this application. It displays the heading, the slogan and the trending hashtags processed by the Storm topology. The endpoint also takes care of search queries made by the user. Hence, two types of HTTP method requests are accepted by this endpoint: GET and POST. The GET endpoint renders the homepage, while the POST request handles search queries. The Python code for the endpoint is :

```
@app.route('/', methods=['GET','POST'])
def home():
    if request.method=='POST':
        var = request.form['query']
        return redirect('/'+var, code=302)
    else:
        distincthashtags = collection.distinct("hashtags")
        return
render_template("home.html",distincthashtags=distincthashtags)
```

The endpoint makes a check for the request method. If POST request is made, the endpoint redirects to the analysis page of the hashtag that is queried by the user. If the

```
<div class="starter-template">
        <h1>Chirrup</h1>
        <p class="lead">A unique way of looking at data.<br>
Explore. Evaluate. Evolve.<br>Find sentiment of tending twitter
topics and do more with data. </p></div>
      <div class="starter-template"
style="background:rgba(211,211,211,0.5)">
        {% for x in distincthashtags %}
          <a href="/{{x}}">#{{x}}</a>
        {%endfor%}
      <div>
    </div>
```

request method is GET, the endpoint populates distinct hashtags from the MongoDB collection and sends the list to render_template function. This function renders the home.html page and passes the list of hashtags to it. The main content source for home.html looks like:

As clear, the second div has templating code for rendering the fetched hashtags from the endpoint. It creates hyperlinks to the hashtags so that the user can gain in depth analysis from the page for that particular hashtag. The user can also search for the hashtag using the search box provided in the tab. Figure 4 demonstrates the home page.

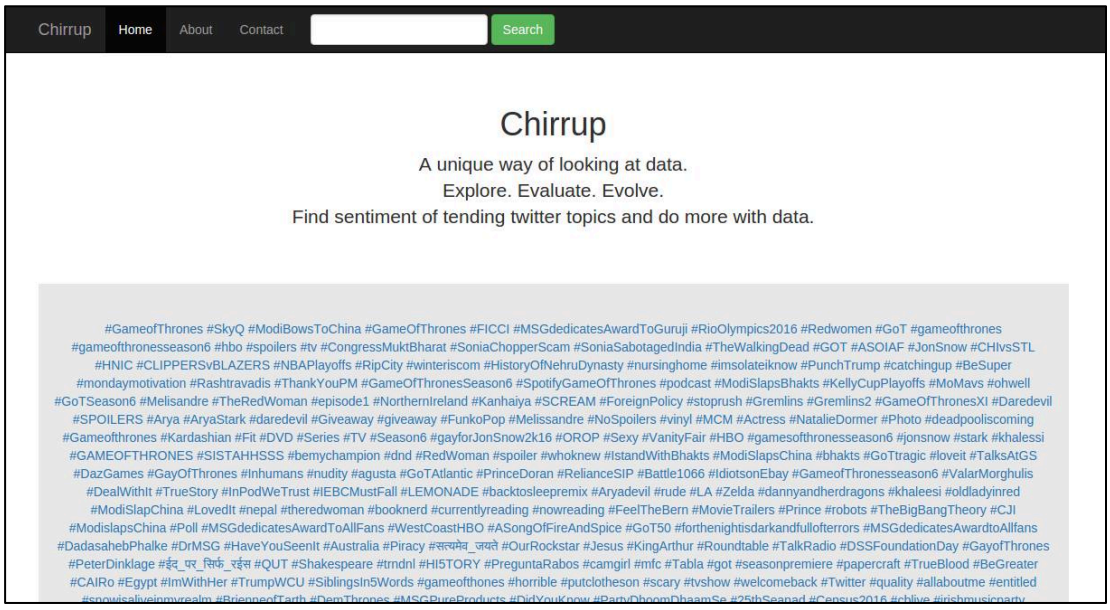

Figure 4 Chirrup Home Page

9.3 Analysis Dashboard

The dashboard serves to display statistics analyzed with respect to a particular hashtag. It displays the overall average sentiment, i.e. the averaged sentiment of all the tweets pertaining to a particular hashtag.

```
@app.route('/<input>', methods=['GET','POST'])
def analyze(input):
    hashtag = input
    country_sentiment_query =
list(collection.aggregate([{"$match":{"hashtags":hashtag}},{"$group"
:{'_id':'$country',"avgsentiment": {"$avg":"$sentiment"}}}]))
    average_sentiment_query =
list(collection.aggregate([{"$match":{"hashtags":hashtag}},{"$group"
:{'_id':'sentiment',"avgsentiment": {"$avg":"$sentiment"}}}]))
    if len(average_sentiment_query)==0:
        return render_template('fourohfour.html')
    country_wise_sentiment = json.dumps(country_sentiment_query)
    average_sentiment = json.dumps(average_sentiment_query[0])
    sorter = [('timestamp', 1)]
    last_ten_tweets =
list(collection.find({"hashtags":hashtag},{'timestamp':0, '_id':
0}).sort(sorter))[:10]
    return
render_template("analysis.html",country_wise_sentiment=country_wise_
sentiment, average_sentiment=average_sentiment, hashtag=hashtag,
last_ten_tweets=last_ten_tweets)
```

The first line for the endpoint is the function decorator that denotes that any wildcard string can be passed following the root identifier, and it will be denoted as the variable input. To simplify the readability of the code, we rename input to “hashtag”. All queries revolve around this parameter.

9.3.1 Country-wise Average Sentiment

First, we extract the country wise average sentiment for the hashtag. The corresponding MongoDB query for this looks like:

```
collection.aggregate([
{"$match":
      {"hashtags":hashtag }
},
{"$group":{'_id':'$country',
            "avgsentiment": {
                  "$avg":"$sentiment"
                  }
            }
}
]);
```

The query shown is what is called a *pipeline query* in MongoDB. It means that the results from one step are forwarded to another and the result of the query coming out is of the final aggregation. Here, we ask MongoDB to first find all the tweets matching the hashtag, group them by country and then calculate the average sentiment for every group. The result that we get is a list of records where the key is a country and the value is the average sentiment for that country. We display the average sentiment, as shown in Figure 5, by country on a world map rendered using the Google GeoCharts [13] API.

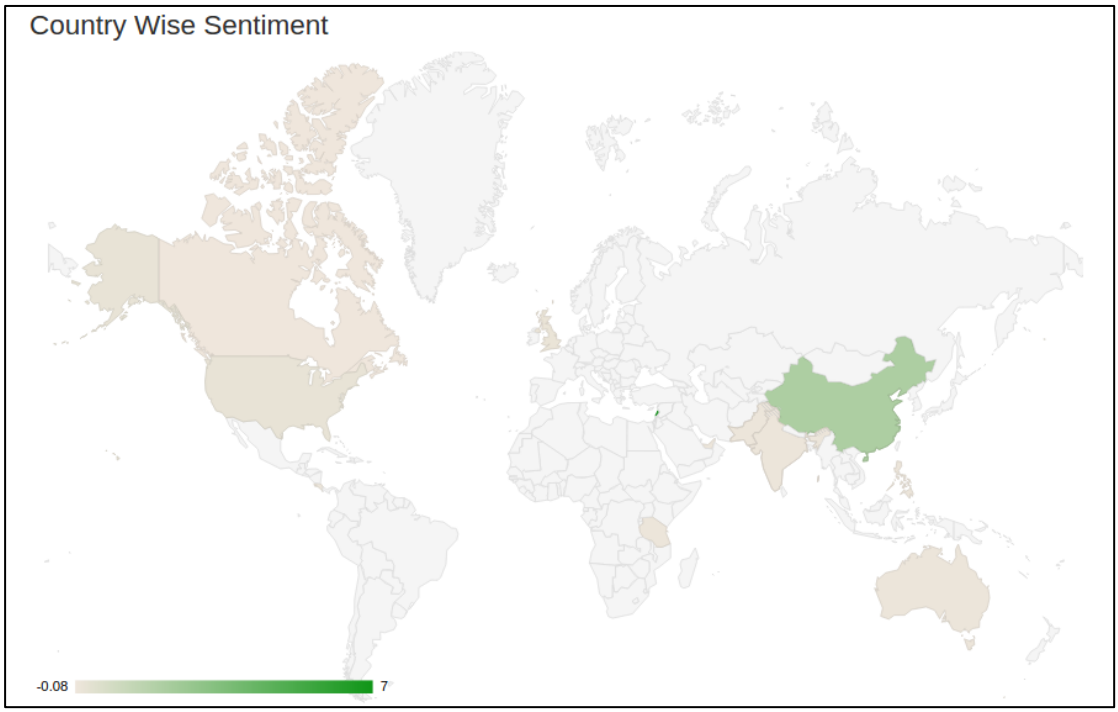

Figure 5 World Map displaying country wise sentiments. The sentiment value ranges from -0.8 to 7

As seen from Figure 5, the sentiment values range from -0.08 to 7 and we notice that the Chinese are happy (green color) about this particular hashtag as compared to the North American and Indian Subcontinents (yellow color).

The JavaScript Code for the GeoChart from GoogleCharts API is:

```
<!-- map code start -->
    <script type="text/javascript">
      google.charts.load('current',
{'packages':['geochart']});
      google.charts.setOnLoadCallback(drawRegionsMap);

      function drawRegionsMap() {
        var datum = eval({{country_wise_sentiment|safe}})
        var arr = [['Country','Sentiment']]
        for(var i=0; i<datum.length; i++)
        {
          country = datum[i]._id
          sentiment = datum[i].avgsentiment
          var cont = [country, sentiment]
          arr.push(cont)
        }
        console.log(arr)
        var data =
google.visualization.arrayToDataTable(arr);

        var options = {};

        var chart = new
google.visualization.GeoChart(document.getElementById('re
gions_div'));

        chart.draw(data, options);
      }
    </script>
    <!-- map code end -->
```

The script populates the data array using the list returned by the average sentiment query executed by the backend on MongoDB.

9.3.2 Overall Average Sentiment

Then we extract the overall average sentiment from the tweets. To do this the sentiment score of all the tweets relating to a particular hashtag is averaged by the query:

```
average_sentiment_query = list(collection.aggregate
([{"$match":
      {"hashtags":hashtag}
  },
  {"$group":
      {'_id':'sentiment',"avgsentiment":
            {"$avg":"$sentiment"}
      }
  }
 ]))
```

This is another *pipeline query* in MongoDB. It means that the results from one step are forwarded to another and the result of the query coming out is of the final aggregation. Here, we ask MongoDB to first find all the tweets matching the hashtag, and then calculate the average sentiment in the whole dataset for the hashtag. We have selected a wildcard grouping parameter so that it just considers the whole output from the $match query as a single group. The output of this query is a JSON document having the key 'sentiment' and the value as the average sentiment of the hashtag.

We display the result of this as a gauge with range -10 to 10 and the arrow pointing to the calculated overall average.

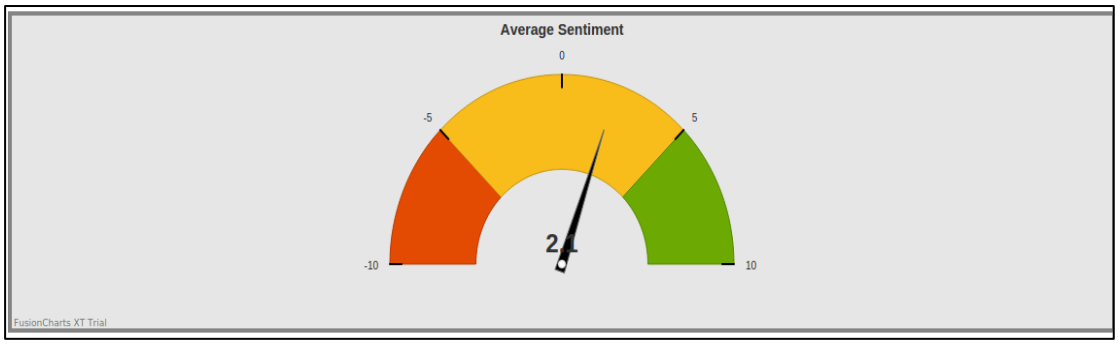

Figure 6 Average Sentiment is 2.1

This gauge is designed in JavaScript using the FusionCharts [14] API as represented in Figure 6. The JavaScript Code is:

```
<script type="text/javascript">
        FusionCharts.ready(function () {
            var w = eval({{average_sentiment | safe}})
            var csatGauge = new FusionCharts({
                "type": "angulargauge",
                "renderAt": "chart_div",
                "width": "100%",
                "height": "300",
                "bgColor": "FFFFFF",
                "bgAlpha":"0",
                "dataFormat": "json",
                "dataSource":{
                       "chart": {
                          "caption": "Average Sentiment",
                          "bgColor": "000000",
                          "decimals":"1",
                          "bgAlpha": "10",
                          "lowerLimit": "-10",
                          "upperLimit": "10",
                          "theme": "carbon",
                          "chartBottomMargin": 50,
                          "showValue":1,
                          "showBorder": "1",
                          "borderColor": "#666666",
                          "borderThickness": "4",
                          "borderAlpha": "80"},
                       "colorRange": {
                          "color": [{
                                "minValue": "-10",
                                "maxValue": "-5",
                                "code": "#e44a00"
                             },{
                                "minValue": "-5",
                                "maxValue": "5",
                                "code": "#f8bd19"
                             },
                             {
                                "minValue": "5",
                                "maxValue": "10",
                                "code": "#6baa01"
                             }]},
                       "dials": {
                          "dial": [
                             {
                                "value": w.avgsentiment,
                                "showValue": "1",
                                "tooltext": "Average
Sentiment"
                             }]}}});
            csatGauge.render();});
        </script>
```

9.3.3 Tweets With Sentiment Values

Then we extract the top ten tweets along with their sentiment values from MongoDB and represent them on the dashboard as shown in Figure 7.

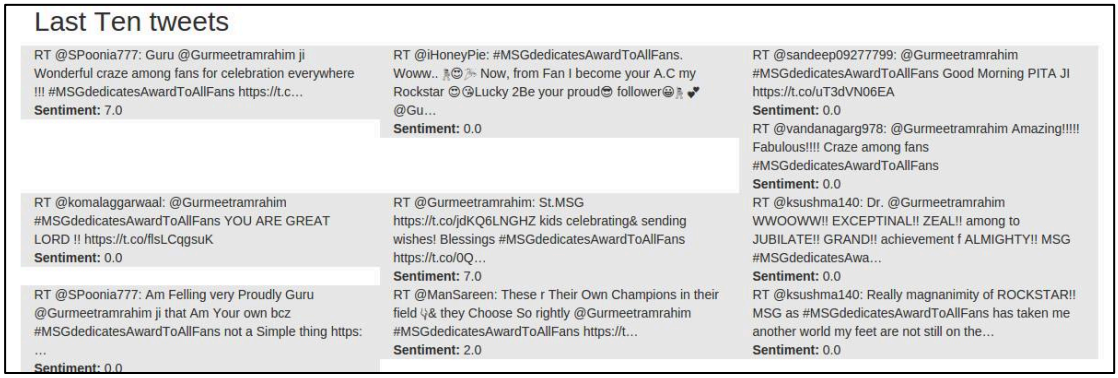

Figure 7 Top Tweets with Sentiment Values

The HTML code which is templated by the Flask API :

```
<div class="row">
      {% for x in last_ten_tweets %}
        <div class="col-md-4"
        style="background:rgba(211,211,211,0.5)">
         {{x.tweetText}} <br>
         <b>Sentiment:</b>
        {{x.sentiment}} <br>
       </div>
      {% endfor %}
</div>
```

9.3.4 The Dashboard

Figure 8 represents the full dashboard.

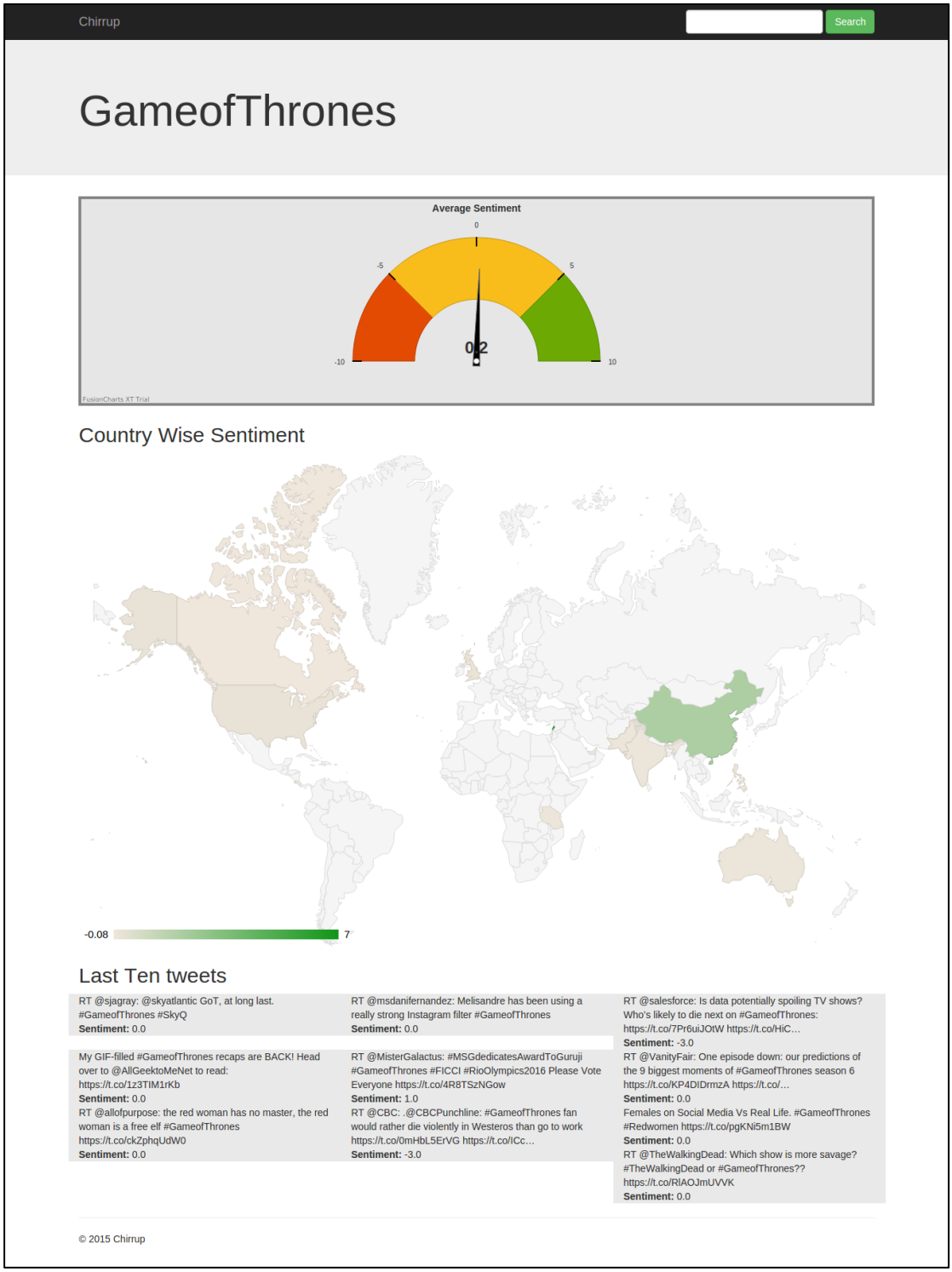

Figure 8 Analysis Dashboard representing the GameofThrones Hashtag

9.4 404 Page

We all know that the user is not perfect. The hashtag queried might or might not exist in the database. This non-existence of a given hashtag can be due to its unpopularity or it has recently come into existence. Or it may sometimes be due to erroneous typing.

We take care of such an event by redirecting the user to a 404: Not Found page. This page displays the error message and also has a search box so that the user may try again. The page is shown in Figure 9.

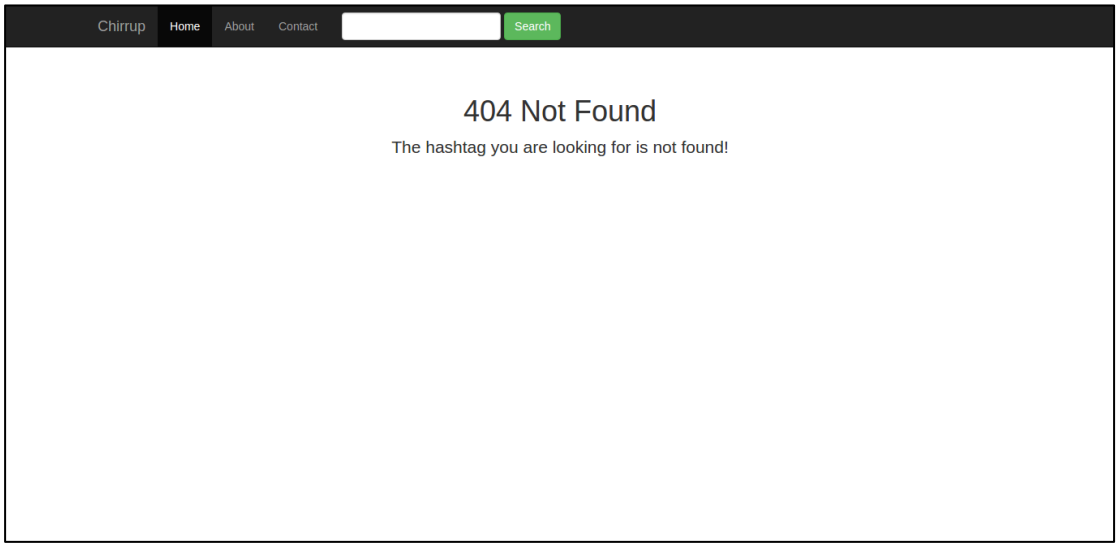

Figure 9 404 Page: Hashtag not found.

CHAPTER 10

CONCLUSION

We have successfully built a stream processing system using Apache Storm and a web service to display results from our analysis. We have run the service through a benchmarking technique that measured the overall performance of our streaming system. We tested this system in both single-node and distributed environment. The results that we received were excellent. Though excessive logging did have some bottleneck on the performance, but it did not matter much as Apache Spark did wonderful job in managing JVM resources and executions.

Hence, we performed some the benchmarking on a single node cluster. The results that we found were astounding and very satisfactory as seen in Table 1.

Table 1 Performance Characteristics

Property	**Single Node**
Number of tweets processed per second	~3000
Round trip time of database insert requests	752531 nanoseconds = 0.7ms
Average growth in size of database	2GB/hour

As we can see, the data-storing overhead is not so much as compared to other Big Data applications. Thanks to MongoDB's data compression and intelligent indexing techniques that saves lots of disk space. Given the computational overhead from Storm, no worker node totally ran out of memory or disk space even when giving the entire Twitter stream without filtering out the keywords. These statistics made it clear that the tools chosen were right for the task.

The tool seems to have a promising future as it could be used for various purposes as required by the developer. The tool's source code is hosted on Github (https://github.com/streampizza/chirrup) as an open-source project and developers are enthusiastic about it. People have reported bugs and given suggestions.

At the end of the project, we had learned much about stream processing, handling Big Data tools and setting up the correct environments for the correct purposes. It was fun learning about and working with so many tools and frameworks.

REFERENCES

[1] The Social Skinny(www.thesocialskinny.com)

[2] Business2Community(www.business2community.com)

[3] Social Bakers (www.socialbakers.com)

[4] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, "Storm@Twitter," In SIGMOD pages 147-156, 2014. (http://db.cs.berkeley.edu/cs286/papers/storm-sigmod2014.pdf)

[5] Finn Årup Nielsen, "A new ANEW: Evaluation of a word list for sentiment analysis in microblogs", Proceedings of the ESWC2011 Workshop on 'Making Sense of Microposts': Big things come in small packages 718 in CEUR Workshop Proceedings : 93-98. 2011 May.

[6] http://arxiv.org/abs/1103.2903 Apache Kafka (kafka.apache.org)

[7] MongoDB (www.mongodb.org)

[8] Flask (http://flask.pocoo.org)

[9] Apache Zookeeper (http://zookeeper.apache.org)

[10] Twitter4J (http://twitter4j.org/en/index.html)

[11] Twitter API (api.twitter.com)

[12] Bootstrap(www.getbootstrap.com)

[13] GoogleCharts (http://developer.google.com/charts)

[14] FusionCharts (www.fusioncharts.com)