Event Detection from Big Data gathered from Social Media

Using tweets collected from Twitter to detect events

Abstract

This report documents and explains the design and the implementation of a prototype program that will attempt to detect events using previously collected tweets. This report will also analysis and evaluate the effectiveness of how well the program can detect events as well as the scalability of the program. Finally this report will also look at the problems faced with processing large datasets and how these problems where resolved.

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Table of Contents

Contents

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Table of Figures

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Introduction

Problem

In recent years social media sites such as Twitter or Facebook have become the forefront of discussion for news and events as they break out. Sometimes events are even talked about on social media sites before mainstream media can report them such as the crash of the US Airways flight 1549 in which onlookers broke news of the plane crash on social media sites 15 minutes before mainstream media could get on the scene.

Being able to detect events has in recent years become a key area of interest for researchers for its uses in the semantic web field. Social media sites generate a colossal amount of data each day (such as Facebook which produces 500 TB of data day[1]). This would be impossible to manually look through to find events so we need to turn to automatic event detection to do the job.

Twitter is a microblogging social media site in which people can create and receive tweets which are messages that can only contain up to 140 characters. Each tweet has some

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

descriptive metadata associated with it which may include information such as where the tweet was created, what time the tweet was created and hashtags which are short descriptive words or phrases that are prefixed with a "#" symbol which identify which topics the tweet is about.

Automatic event detection using datasets collected from Tweets presents a unique problem due to the sheer amount of tweets that are created each day.

On average 5700 tweets are generated per second. This is around 500 million tweets per day[2]. Therefore careful consideration needs to be made on how to process this data as divide and conquer algorithms will lead to data about an event being spread across multiple partitions which could lead to events being missed as each partition would have less information on the event then if we only had one partition.

Following on from this the actual detection of events needs to be quick and accurate. The speed of event detection is important as information about events become out of date as time passes. For instance if it takes 3 hours to detect events that happen in a one hour period then the chances are those events are already out dated as new information relating to that event will have been talked about in the 3 hour period.

A final problem that is present in detecting events from Twitter is that traditional event detection detects events on document collections such as news articles. The assumption for this type of event detection is that all the documents in the collection are related to some kind of event. This is not true for datasets coming from Twitter. In 2009 Pear Analytics[3]performed a study to find out what Twitter was really used for. Their study suggested that around 3.75% of all tweets are spam while 5.85% of all tweets are from corporations trying to self-promote and 40% of all tweets are "pointless babble".

Solution

To detect events in a given time period using data collected from Twitter in a quick and efficient manner this report proposes to find events by looking for sudden bursts of hashtags in the dataset. If a hashtag is being used more than usual then there is a high chance that an event is going on as generally there is not a high fluctuation in hashtags describing "pointless babble".

Once hashtag outliers have been found events can be built up by grouping similar hashtags together using K means clustering. Finally a tag cloud will be created for each group of similar hashtags. These tag clouds will visually represent the text data found in all the tweets that share a hashtag in this group. Words that appear more frequent in the text data will have a larger typeface then words that appear less frequently. To remove clutter from the tag clouds stop words will not be included.

Stop words are a term coined by Hans Peter Luhn in 1958 to describe a list of non-information bearing words such as “the”, “and” or “again”. As these Stop words do not add any information to the event there is no point including them in the tag clouds. The full list of words that are ignored can be found in the appendix under appendices 2.

Clustering the data like this however will definitely lead to non-event stories being built up. Therefore the tag clouds will be returned in order of the likelihood that it is talking about an event. This measurement will be found by looking at the verb to non-verb ratio of words in the event as a high frequency of verbs should suggest an event. (You need to mention a verb to describe an event this is not usually case for other types of tweets which may have an irregular hashtag frequency such as a company give away).

Aims

Given inputs of a dataset of tweets, a time range a specified boundary box as well as the number of cluster groups we should expect to derive a list of tag clouds which are ordered based on the likelihood that the tag cloud is describing an event.

In this project the likelihood of the tag clouds describing an event will be based on the number of verbs a tag cloud contains as it is impossible to talk about an event without using a verb. Therefore if a tag cloud contains a low amount of verbs it has a high probability that it is not talking about an event.

Another aim of the project is to make the program scalable. Scalability measures “how effective the program will make use of additional processes”. A program is said to be scalable if “it is possible to keep efficiency constant by increasing the problem size as the number of processors increases” where efficiency is defined as:

$$\text{Efficiency} = \frac{T_{seq}/T_{par(N)}}{N}$$

N is defined as the number of processes

Tseq is defined as time taken for the program to run sequentially on one processor.

Tpar(N) is defined as the time taken for the program to run on N processes of the parallel machine.

A final aim of the project will be to detect events given a large dataset that could not fit in memory. To process data of any size this project will make use of two CPython features Generators and Iterators. Iterators are containers which do not store values in memory. Instead iterators generate the values for their container on the fly.

As Iterators only know how to generate the next item in the collection once you have retrieved a value from an iterator you can never retrieve it again. As iterators do not store

the entire dataset in memory (they only need to store the current value of the iterator and the iterator function itself) it means the iterator can contain a dataset of any size. Generators are CPython functions that behave like an iterator. Both iterators and generators are explained in detail in the implementation section.

Background

What's an event?

For this project an event will be defined as a thing that happens or takes places. An event can be a planned public occasion, a social occasion or something that has occurred to a place or person.

How has it been tackled before?

There are two major ways in which researches have attempted to detect events in the past. The first major way that event detection has been approached by is wave analysis. In 2011 HP attempted to detect events from Twitter using Event Detection with Clustering of Wavelet-based signals (EDCoW). The idea behind this is to build signals for individual words. Signals were chosen to be used as they can be quickly computed by wavelet analysis and they require a small amount of storage space.

By comparing signals from previously built data it becomes possible to filter away trivial word. Then signals can be clustered together to build up an event. Finally "big events can be differentiate with trivial ones by looking at the events significant which can be quantitated by the number of words present and the cross correlation among words relating to the event."

EDCoW suffers from the draw back that it treats each word independently. This can lead to different events becoming clustered together this can be further seen from the results of the study.

The other major way that events have been automatically detected is to use a topic model distribution[4]. The idea behind this is that content that talks about global events is often going to follow a global topic distribution that is time-dependent while this is not true for personal posts. This means that we can separate events from personal posts through unsupervised learning and then use a state machine to detect bursts from the discovered topics to find the events.

A key limitation of detecting events with this method is that the number of topics to detect is predetermined which does not work well as it's impossible to know the number of events to discover before they are all discovered.

Tools and libraries the project solution is based on

CPython3.3 was chosen as the language to implement this project in because of the large natural language processing community that use CPython. As natural language processing is a research area in itself this project required a library that could abstraction away from natural language processing to detect verbs in phrases. By using python I had many choices in which libraries to use such as textblob or pyntlk.

Python was also chosen for its concise syntax, dynamic typing and readability. These are important features needed when there is a limited time to implement a prototype as it reduces technical debt created from "hacking a solution together" while still being quick to develop with.

CPython3.3 was chosen over CPython3.4 as CPython3.4 came out in March 16th 2014 at which point project development was well underway.

The key libraries that are used for this project are Twython which is a library for scraping tweets from Twitter, Textblob which is an abstraction from natural language processing and Pillow a library for creating pictures. More information about these libraries and their use can be found in the implementation section of the report.

For this project the dataset has been gathered exclusively from Twitter. Twitter was chosen as the social media site to gather data from because of its popularity. Twitter was also chosen because messages can only be 140 characters or less which means a lot less processing will need to be done on the text compared to messages from other social media sites. Finally Twitter was chosen because of its streaming API. This is a public API that anyone can use which makes it very easy to collect tweets from.

Benefactors for this project

As previously mentioned a key area that automatic event detection is useful for is the semantic web field as the event detection could turn unstructured social media data into structured documents which talk about events therefore helping to build a "web of data".

Other key benefactors for this project would be the media as currently there is too much information on social media to manually process through. By having automatic event detection the media will have to look through a lot smaller datasets to report and covering news as it breaks out.

A final benefactor for this project would be the general public as they could use automatic event detection to attempt to find events that are going on in their area such as school fates.

How does my method differ from these previously suggested methods?

In this project I propose a method that is similar in design to EDCoW. However instead of comparing wave signals pointless words will be filtered by outlier detection. In theory this should work as long as the assumptions made by the topic model distribution method are correct (personal tweets follow a distribution that is mostly stable). Like the EDCoW method events will be built up by clustering similar hashtags together. However unlike EDCoW events will be differentiated from non-events by looking at the verb to non-verb ratio.

Design

User requirements of program

As there are many areas that can be looked at in this project (natural language processing, event detection, big data, clustering to name a few) there was a very high risk of not meeting the imposed deadlines on the project due to feature bloat. To solve this issue clear user requirements were designed which allowed development to be focused on implementing the core necessary features. The key user requirements can be summarised as 5 main points.

1. Given a dataset the program should be able detect events in a given space and time. Possible events that are detected should be represented as a tag cloud and the tag clouds should be in ordered in the likelihood that the given tag cloud is an event.
2. The event detection program should make use of parallelism techniques to speed up the runtime of the program.
3. The event detection program should be able to handle large datasets that could not fit in memory.
4. Although the example data in this project only uses data collected from Twitter the event detection program should in theory be able to detect events from a wide range of social media sites as long as the social media sites incorporate the following meta data; creation date, geo-location and tagging.
5. Although the example data in this project is stored in Mongo DB the event detection program should not be coupled to one database implementation.

Basic overview of programs

TwitterScraper Program

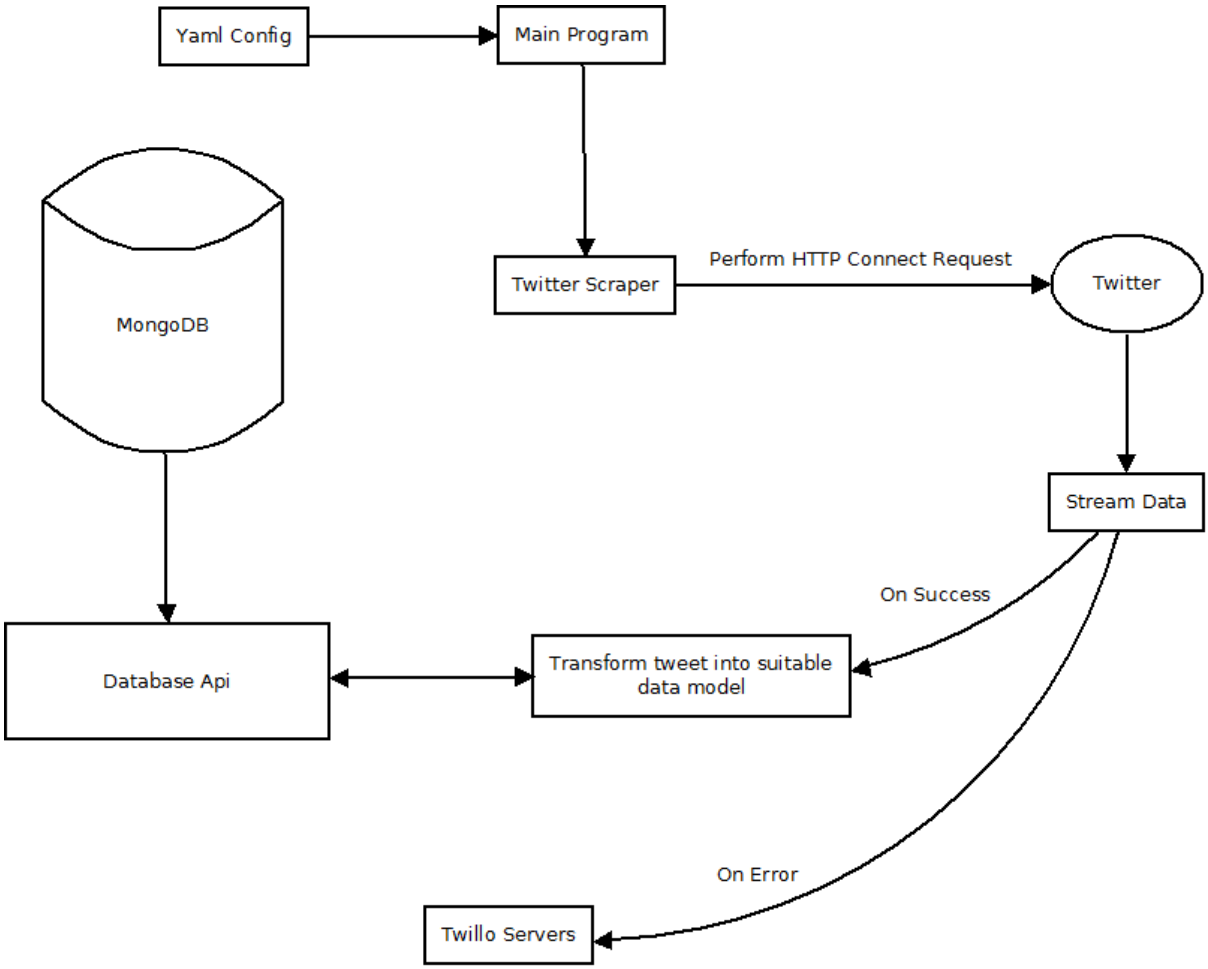

Figure 1

The Twitterscrapper's main functionality can be split into three main tasks and can be viewed as figure one. The first task that the Twitterscrapper is responsible for is to scrape tweets from Twitter. To scrape tweets from Twitter this project makes use of Twitters streaming API which allows us to make a long held HTTP request to the server. Once the Twitterscrapper has successfully connected to Twitter, Twitter can send the program tweets as they are posted. More information on using Twitters API can be found in the implementation section.

Due to the high memory usage of the Twitterscrapper once a tweet was successfully collected the program slept for around a second. This meant that on average 1.2 tweets would not be collected per minute due to the sleeping which was a total loss of around 20736 tweets throughout the whole project. This loss was not seen as significant enough for the sleep in the program to be removed.

The second task the program is responsible for is remodelling the data that is received from Twitter. As it should be theoretically possible for us to detect events using data from any form of social media (as long as the data has tags and where and when it was created) the tweet data needs to be saved in a format that means that the database is not coupled to only being able to store tweets. An example format of how a tweet is stored can be seen below:

```
{
        _id: 12345,
        Text: "this is the text of the tweet",
        HashTags: {"spam", "ham", "eggs"}
        Loc: [latitude, longitude]
        Datetime: 13974844.0
}
```

As seen above the majority of metadata has been stripped from the tweet as it is ether unnecessary or metadata that only has relevance to Twitter (such as if it has been retweeted).

The JSON format was chosen to represent the model as it is language agnostic, simple to use and text based. Finally the mark-up language itself is concise so it will take up minimal storage. The largest drawback to using the JSON format is that it is not easily read by humans however this is a non-issue as our datasets are too large for humans to manually go through anyway.

By remodelling the data and cutting useless information a lot of space was also saved. In a short experiment 10 tweets were collected. The size of each tweet in bytes as a JSON string was noted. Finally the size of the remodelled tweet in bytes as a JSON string was noted.

Size of tweet in bytes	Size of remodelled tweet in bytes
2785	222
2657	250
3315	294
2811	187
3001	266
3028	284
3001	202
2955	190
3168	311
3078	227

From this experiment we can see that on average the JSON of the remodelled tweet is 12.55 times smaller than the JSON of the actual tweet.

The final task of the Twitterscrapper is to save the remodelled tweets to Mongo DB. This is done through the Database API. More information about the Database API can be found in the design section.

As writing to Mongo DB is an expensive operation and we want to be spending as much time in the Twitterscrapper collecting tweets as we can (to collect more tweets per minute) we write our tweets to Mongo DB in a non-blocking manner (we request to save the tweet to Mongo DB however we do not make sure the tweet gets saved). Furthermore we do not save to the database once we receive the tweet. Instead every time we receive a tweet we store it in a list. Once our list is full enough we write the entire list to Mongo DB. This is a significant speed increase as Mongo DB is optimized for bulk operations.

The Twitterscrapper program itself has no GUI and needs to be run on the command line. This is because an interface was not needed for the program and implementing an interface on top of the program would of lead to increased memory usage.

The Twitterscrapper program looks for its settings in a "settings.yaml" file in the same directory as were it is run. A configuration file was needed for this program due to the sensitive nature of some of the arguments required (such as a developer key for Twitter). A sample configuration file can be found in the appendix under appendices one. YAML was chosen as the configuration language as it human readable and easy to integrate into python programs.

Finally as it is not desirable for the Twitterscrapper program to go down the Twitterscrapper is integrated with Twilio which is a service which allows developers to send texts or calls from a program. If the program detects that an error has occurred or that the python interrupter is going to be terminated then the Twitterscrapper will send you a text notifying you so you can restart the program.

Database API

The database API is a library that is shared between both the Twitterscrapper and the event detector programs. The benefit of having a Database API rather than programs making database calls themselves means that if the implementation of the database changes than the changes only need to be made in one place. Another key benefit of having a Database API is that it becomes simpler to switch databases if needed as the implementation details of communicating with the database are only in the Database API.

For this project Mongo DB was chosen to store the datasets. Mongo DB is a document orientated database system and was chosen for its shard support which would allow data

records to be saved across multiple machines. This allows horizontal scalability with the dataset (if our dataset grows too large we can meet the additional demands by adding another computer to our cluster. This is in contrast to vertical scalability which would require us to upgrade the hardware of the machines).

Another key reason Mongo DB was chosen for this project was for its ability to be able to index geo-location data. As this project requires event detection in a given location fast access to tweets stored in that region will be needed. This is only possible if we index our documents by their geolocation which allows us to retrieve the documents in O(log(N)) rather than O(N) time.

The final reason Mongo DB was chosen for this project is because of its querying system

Originally a SQL solution was used to store the datasets and the schema for this database can be seen in figure 2

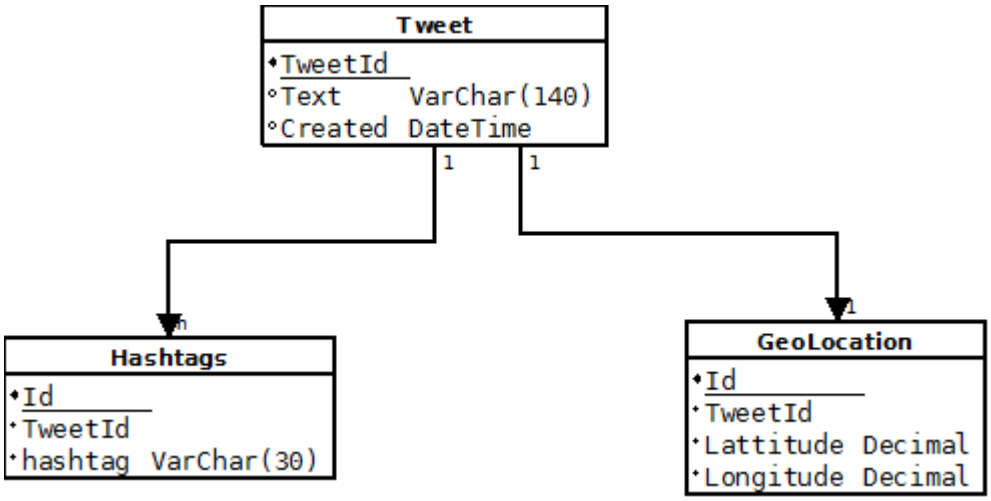

Figure 2

Using SQL was eventually scrapped as it became apparent that the main advantages of SQL could not be utilised. To get the full metadata of a tweet required two joins to the tweet table which was a performance hit. To solve this problem the database could be un-normalised however at that point it would make sense to move over to a database that can make full advantage of not using a Schema.

The final reason a SQL database was not used to store the dataset was because of Brewer's theorem which suggests that it is impossible for a distributed computer system to provide guarantees for "consistency, availability and partition tolerance". This means we cannot take full advantage of an ACID database in a distributed environment (which is what a SQL database is) and so it would make more sense to turn towards a database that uses BASE such as Mongo DB.

Event Detection Program

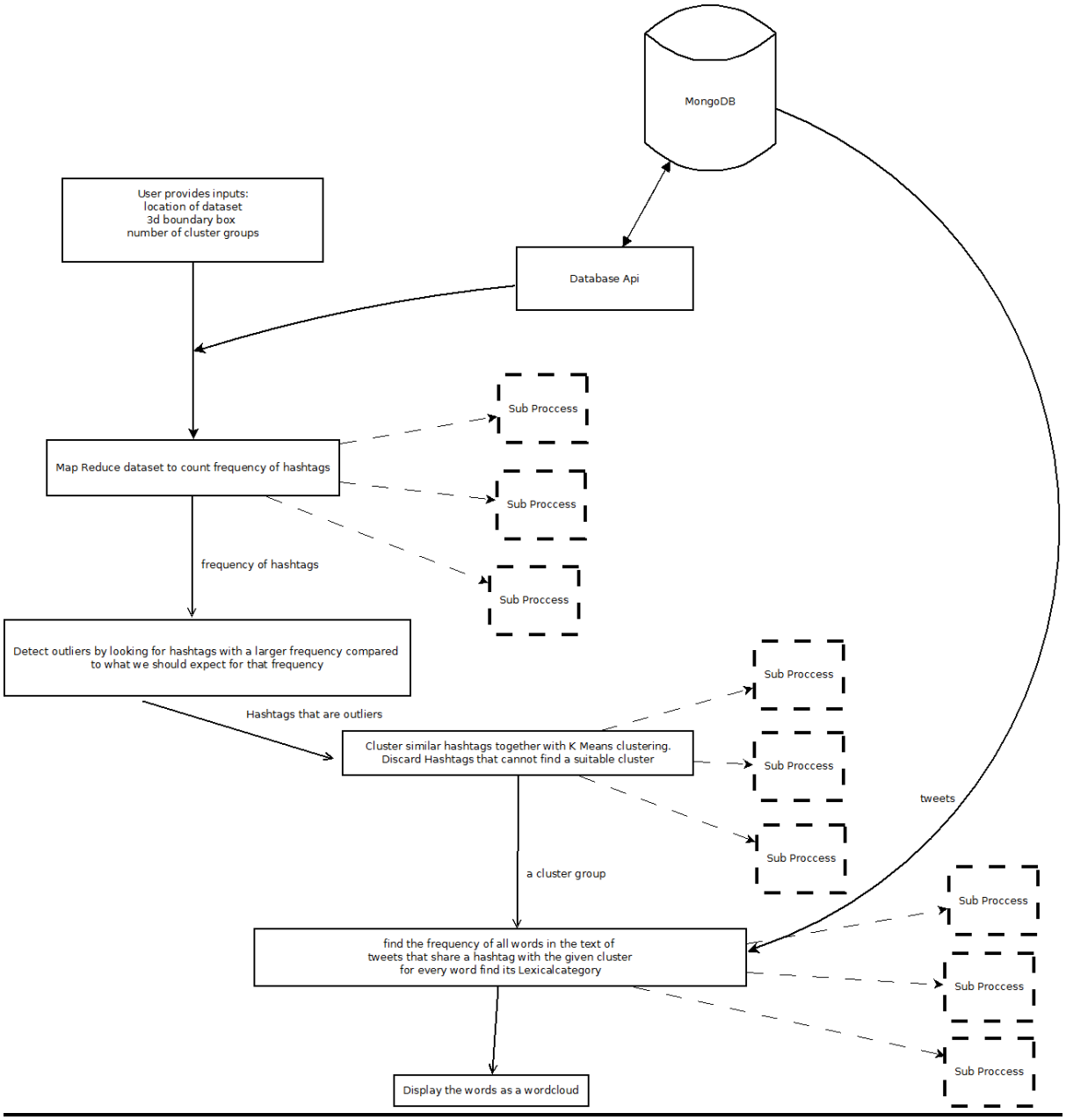

Figure 3

The event detection program needs to perform 3 main functions on the dataset to be able to detect events. A visualization of this can be seen in Figure 3. These functions are detecting hashtag outliers, clustering similar hashtag outliers together to build an event and finally to create a world cloud of the event to build a story.

To detect hashtag outliers we need to compare the frequency that the hashtag appears in a place in the given time period to the average frequency of that hashtag in the given in same time period.

To solve this in a scalable way a partial MapReduce implementation has been used to count the frequency of hashtags for any number of tweets. MapReduce was a programming model developed by Google[5] that allows for processing of large datasets over a cluster.

The MapReduce model can be broken down into two distinct functions Map() and Reduce(). The Map function's job is to "*transform the targeted dataset into a list of intermediate key value pairs*" while the Reduce() function's job is to "*merge all intermediate key value pairs that share the same key*".

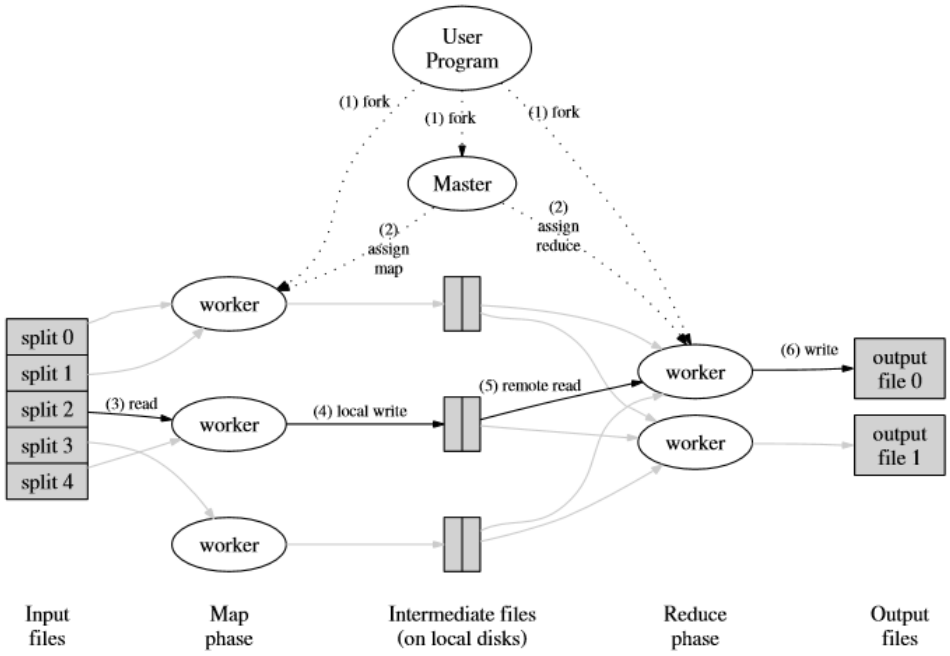

Figure 4

In MapReduce the dataset is split into chunks over the cluster and each node in the cluster will perform a Map() function on its dataset. Once a node has finished its work it will save the intermediate results to a text file. Once all the nodes have finished working the master node will distribute the dataset from the Map() function to the cluster were each node in the cluster will perform a Reduce() function on its dataset. Finally these results will be sent to the master node which will merge the Reduce() results together giving a final output. A diagram of this explanation can be seen above.

For counting the frequency that Hashtags occur we are only interested in the Hashtag itself and a list of Id's of all tweets that share that hashtag. Therefore for each Tweet in our dataset the Map() function will transform the tweet to a list of tuple represented like so "Hashtag": "ID of tweet" examples of this can be seen below:

Before	After reduce function

{
_id: 12345,
Text: “this is the text of the tweet”,
HashTags: {“spam”, “ham”, “eggs”}
Loc: [latitude, longitude]
Datetime: 13974844.0
}
	[
	{“spam” :”12345”},
	{“ham”:”12345”},
	{“eggs” : 12345}
	]

{
_id: 6789,
Text: “this is the text of the tweet”,
HashTags: {“spam”, “ham”}
Loc: [latitude, longitude]
Datetime: 13974844.0
}
	[
	{“spam” :”6789”},
	{“ham”:”6789”}
	]

{
_id: 10112,
Text: “this is the text of the tweet”,
HashTags: {“Monty Python”}
Loc: [latitude, longitude]
Datetime: 13974844.0
}
	[
	{“Monty Python” :”10112”},
	]

For counting the frequency that hashtags occur this would lead to merging all hashtags of the same name to build up a list of every tweet that has used this hashtag. Using the same examples as was used above the Reduce() output should look like this:

```
{
        “spam” {
                        “ids” : [“12345”, “6789”],
                        “freq” : 2
        }
        “ham” {
                        “ids” : [“12345”, “6789”],
                        “freq” : 2
        }
        “eggs” {
                        “ids” : [“12345”]
                        “freq” : 1
        }
        “Monty Python” {
                        “ids” : [“10112”]
                        “freq” : 1
```

```
        }
}
```

An alternative way this could have been implemented was to use Mongo DB to query the dataset using its inbuilt MapReduce implementation. However this would have forced the implementation of the project to become dependent on Mongo DB which is not ideal.

As briefly mentioned before a partial MapReduce solution has been implemented for this project as the full feature set outlined in Google's MapReduce paper was overly excessive for a prototype program. Some of the features that have not been implemented include fault tolerance if a node goes down, restoring of state of the program if a bad record causes a Map() or Reduce() function to error and status information retrieval from the master node.

Once we have the frequency of every hashtag in our given 3d bounding box and we know the frequency of every hashtag in the entire data set for that given area we can begin looking for hashtag outliers. To detect outliers we look for hashtag frequencies that are greater than 3 times the standard deviation from the mean frequency for that given hashtag.

The value 2.5 was chosen after much trial and error. Tweets in London were collected from Sat, 19 Apr 2014 18:02:04 GMT until Sat, 19 Apr 2014 19:02:04 GMT and then the outlier program was run on this data. The number of outliers for a range of standard deviations was collected as seen from the data in figure 5.

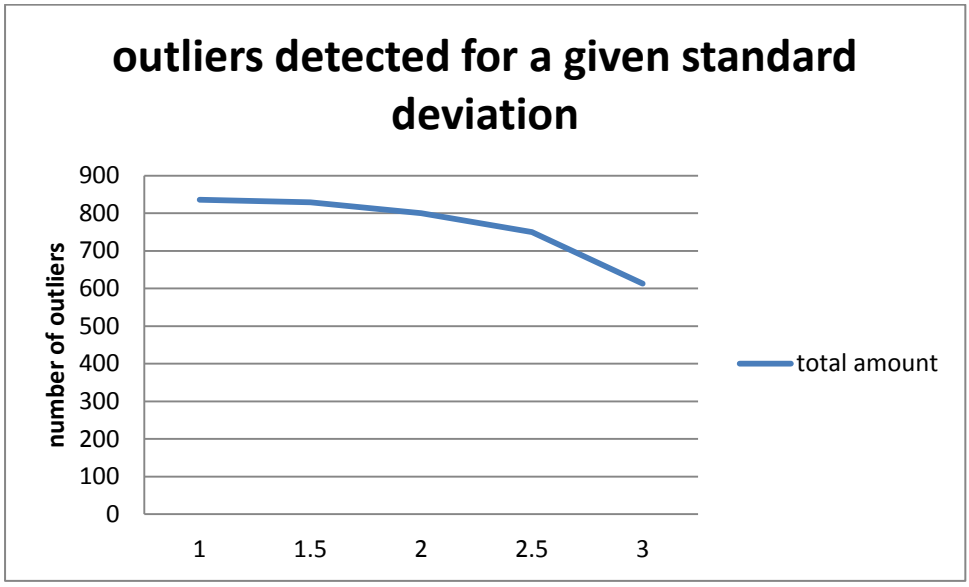

Figure 5

As we can see from the data above the majority of tweets do not get filtered until after the standard deviation hits 2.5 it is for this reason that the value 2.5 was chosen.

The next function of the program is to cluster similar hashtags to attempt to form events. The K Means algorithm was chosen to attempt to cluster hashtags due to its run time of O(K * N * I) were I is the number of iterations, N is the size of the dataset and K is the number of starting centroids chosen. This is because clustering is considered a NP hard problem and one of the key requirements of our project is to detect events in a timely manner and so the clustering needs to be done quickly. To further speed up the program the K Means clustering is run over multiple processes.

The K means algorithm works by starting off with K centroids were each centroid will have its own precomputed distance. Then for each hashtag in our list of outliers that was not chosen as an initial centroid we assign that hashtag to its closest centroid. Finally we recomputed the distances of the centroids. If any of the distances of the centroids have changed then we reassign every hashtag in our list of outliers that was not chosen as an initial centroid to its closet centroid. We keep doing this until none of our centroids move.

K means clustering requires K starting centroids. In this project the starting hashtags to use as centroids are selected randomly from the list of all possible outliers. However a hashtag will never be considered as a starting centroid if it has a frequency of one or less.

This is because if a hashtag is considered an outlier while having a low frequency it often means that the hashtag is either a misspelling or an irregularly used phrase/word. Hashtags like this do not make good centroids as there is a high chance that no other hashtag in the dataset will be seen as similar to that hashtag and so we will be left with an empty cluster.

The clustering in this project also serves as a purpose to get rid of the tweets that Pear Analytics considered as “Pointless Babble”. This is because if a tweet cannot find a good enough similarity to a centroid it is discarded. As “Pointless Babble” tweets often have very specific hashtags such as “#doingLaundry” they will often be discarded.

If K is a low number then K means clustering can often lead to no events being detected. This is because the starting centroids are selected at random. As previously discussed if 40% of all tweets are considered “Pointless Babble” than the majority of the starting centroids would also be considered “Pointless Babble”. This flaw and possible solutions are expanded upon more in the Results and Conclusion section.

The final function of the program is to turn the clustered data into tag clouds. This is done by loading the text of all tweets that have a hashtag belonging to the cluster and counting the frequency of words that appear in the tweets text. Like the hashtag frequency counter

this was also done using a partial implementation of MapReduce. As previously mentioned in the introduction, words that appear in the stop list are not include in the word counting as they are by definition words that “non-information bearing”.

Once we have the frequency for each word that appears in a cluster we need to tag the lexical category for each word. This is done by looking at lexical databases such as WordNet. Finally we can create a tag cloud by creating a new image and placing words on the image in a way so two words do not overlap. The typeface and colour of the font depends on the frequency of the word and its lexical category. The larger the words frequency the larger its Type font will be. A Table of what colour each lexical category represents can be found below.

Lexical category	Colour
Verb phrase	White
Noun phrase	Grey
Prepositional phrase	Yellow
Number	Blue
Other	Brown

An example word cloud generated from the event detection program can be found in figure 6.

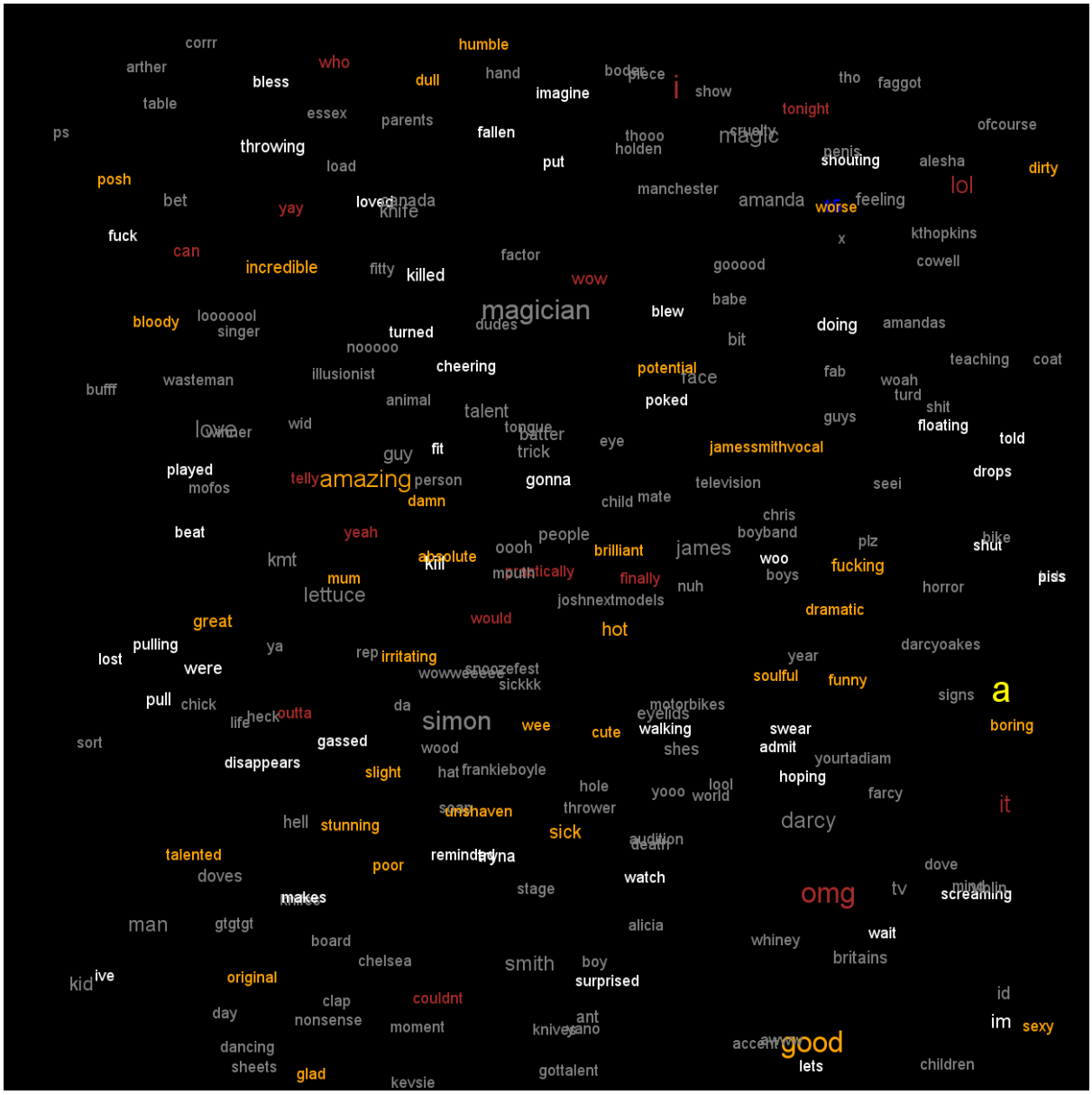

Figure 6

The event detection program runs on the command line console as no interface was needed. The program requires the following command line arguments to run:

- MinX – the latitude coordinate of the bounding box
- MaxX – the width of the bounding box
- MinY – the longitude coordinate of the bounding box
- MaxY – the height of the bounding box
- minTime – an Unix epoch timestamp of the starting time to detect events from.

- maxTime – an Unix epoch timestamp of the end time to detect events until.
- poolSize – how many processes to run the code across.
- Database con – A connection to a Mongo DB collection containing the dataset.

Architecture design

In this section of the report the outline of the classes required for the project will be shown and discussed. Although CPython is an object oriented language it does not support some of the more traditional object orientated features such as strong encapsulation (python does not have private, public protected modifiers to hide implementation details away from other objects). This has effected how the object model of the program looks.

Twitterscrapper

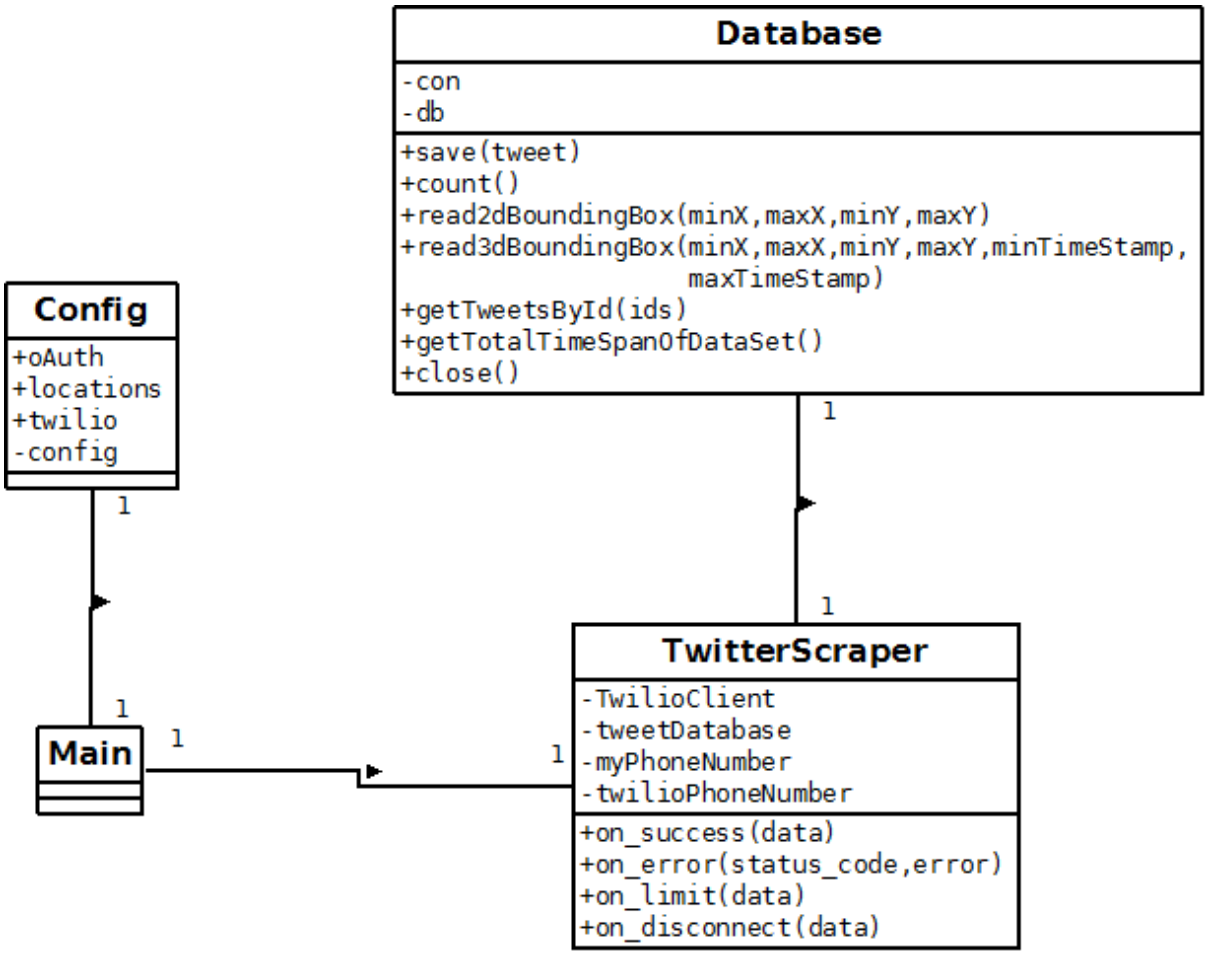

Figure 7

As previously stated the Twitterscrapper's job is to scrape tweets and store a transformed model of the tweet metadata into our database. The settings for the Twitterscrapper are loaded from a YAML configuration file as the settings require sensitive data such as private keys for Twilio and Twitter. The Config class's job is to deserialize our YAML mark-up

language from the configuration file into manipulable python objects. For instance the OAuth field in the YAML configuration file looks like this:

```
oauth:

    accessToken: removed

    tokenSecret: removed

    consumerKey: removed

    consumerSecret: removed
```

In CPython it would make sense if the OAuth field was represented as a dictionary object
`{accessToken: removed, tokenSecret : removed, consumerKey : removed, consumerSecret: removed}`

The TwitterScraper class currently has two main functions. The first function of the TwitterScraper class is to handle what happens when a tweet is successfully received from Twitter. The class also needs to be able to handle any errors that are received when talking to Twitter.

The second function of the TwitterScraper class is to transform the metadata of a tweet received from Twitter so it is of the appropriate model so it can be stored in the database. In an ideal environment the TwitterScraper class should not know how to transform the data instead this should be left up to a brand new class whose job is to transform data. Unfortunately the implementation details of transforming the data became too coupled to the TwitterScraper class and attempting to refactor this problem would cause a lot of breakages in the code.

The on_success method of the TwitterScraper class gets called whenever the Twitter stream has a new tweet. Once new data has been received it will be transformed and the transformed data will be saved to a list. If the list gets too full the on_success method will bulk write these operations to the database. The reason for this has been previously discussed in the design section sub heading: basic overview Twitterscrapper program. As the TwitterScraper class needs to be able send data to the database the class needs to know about the database API. This explains the 1 to 1 dependency as displayed on the UML diagram.

The on_error/on_disconnect/on_limit methods of the TwitterScrapper class get called whenever an error happens within the Twitter stream and these methods are in charge of contacting the user so they can get the Twitterscrapper up and running again. Due to this the TwitterScrapper class needs to know about Twilio (the service used to send text messages) and so this is why the TwitterScrapper needs the twilioClient, myPhoneNumber and twilioPhoneNumber fields.

The database class is an implementation of querying MongoDB and so the methods found in there are methods that help abstract away from the Database layer. Instead of the programs being dependent on the database the programs instead are now dependent on an API whose implementation detail could be changed to work with any database without breaking the programs.

Event Detector

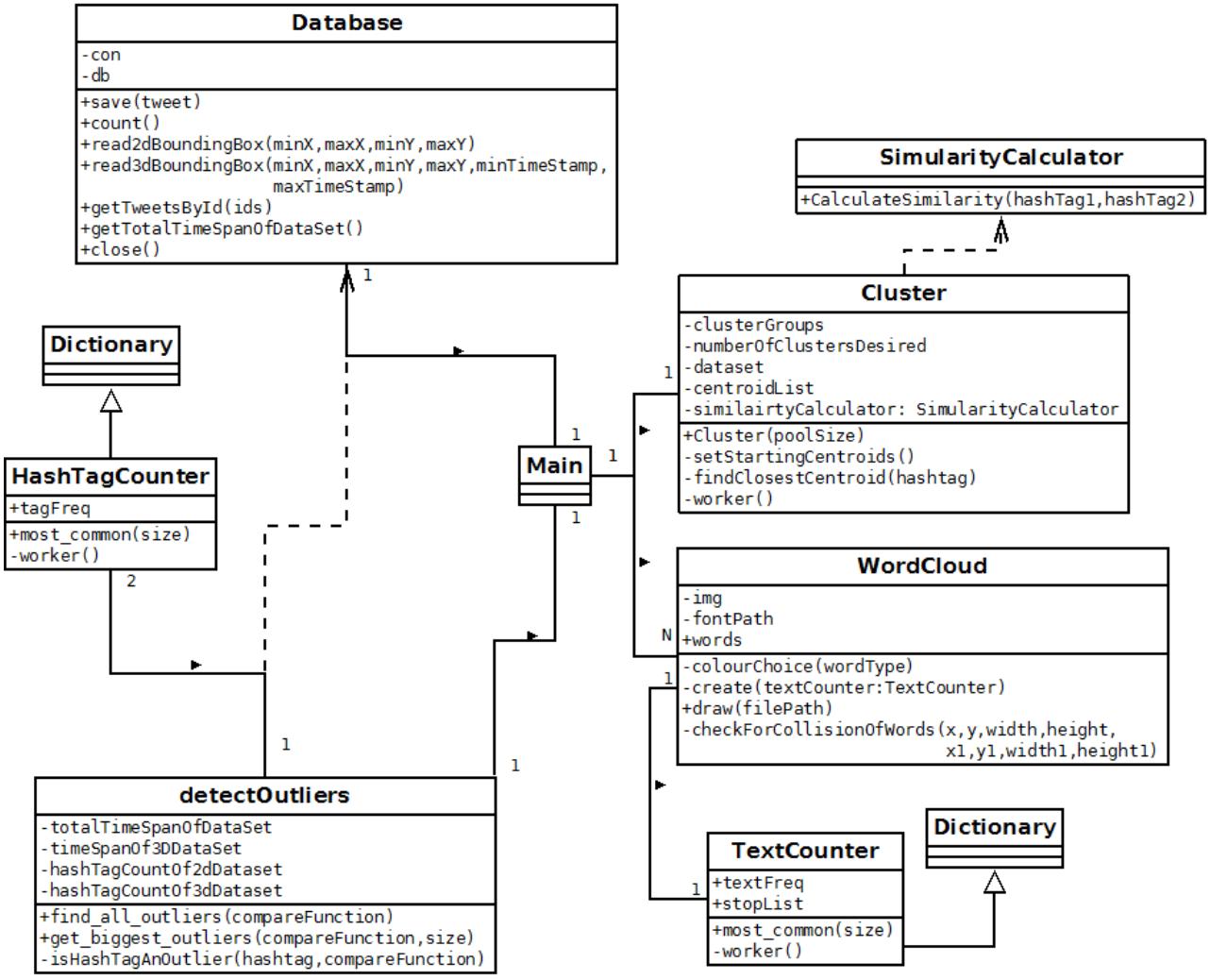

Figure 8

The TextCounter and HashTagCounter classes have a IS-A relationship with the dictionary data structure class and so they can be treated as a dictionary as they implement every method a dictionary would.

The biggest difference between these counter classes and a dictionary class is that these classes are immutable. Another key difference between these classes and the dictionary class is that the initial dataset that passed into these classes is MapReduced over multiple processes to give us a new representation of the data. As these classes will be performing parallel work they implement a worker method which gets invoked by every process that is run by the class. Finally both of these classes implement a most_common method which takes a size as a parameter and returns the X largest frequency words/hashtags from the dataset.

The detectOutliers class requires two HashTagCounter classes along with two timespans that represent how long the tweets where collected for in the counter class. Without these parameters it would not be possible to work out if a given hashtag frequency is larger than 2.5 times the standard deviation of the mean of that frequency. Therefore in the UML diagram the detectOutliers class has a 1 to 2 relationship with a HashTagCounter class.

The detectOutliers class has two methods the find_all_outliers method which returns for the given dataset a list of all hashtags that are outliers. The get_biggest_outliers method returns the X largest frequency hashtags that are outliers. Both of these methods require a compare_function argument. This should be an anonymous function that tells the methods how to determine if something is an outlier or not.

The WordCloud class requires a TextCounter class as a parameter to build a tag cloud from. Therefore there is a 1 to 1 relationship between these classes on the UML diagram. Due to the low level nature of the class the WordCloud class requires a path to a TrueType Font to render fonts which is why font is an attribute. The only visible method of this class is the draw method whose responsibility is to create a .png representation of the tag cloud. Therefore a path parameter is needed so that the draw method knows were to save the .png file.

Finally the cluster class is responsible for clustering similar hashtags together using the k means algorithm. The only exposed method is the cluster method whose job is to cluster similar hashtags together over multiple processes for performance. This method returns a list of clustered objects. Just like the TextCounter and the HashTagCounter as work will be done over multiple processes a worker method needs to be invoked to be run on these processes.

The similarity calculator class is responsible for identifying if two given hashtags are similar. This was refactored into its own class so that the clustering class was not dependent on a single possible measure to detect if hashtags are similar. Therefore these classes need a 1 to 1 relationship in the UML diagram.

Changes since the last report

No spell checking implemented

In the initial report it was mentioned that a key feature to be implemented would be the ability to detect and correct spelling errors in hashtags. For instance the hashtag “Lpndon” should probably be corrected to “London”.

By doing this we can get a greater accuracy when it comes to detecting events as the miss spelling hashtags and the correct spelling hashtags would be clustered together. To demonstrate why this was not implemented I have collected all the hashtags from 10 random tweets in my dataset. The code for this can be found in the appendix

Tweet Id	Hashtags
459626472500895745	Friday, Standard, Skinny, Gingerbread
459626480646230016	EveryCloud
459626482571415552	FF_Special
459626503433904128	MaslowSister
459626503098335232	swollenarm
459626504046264320	Cefnmably
459626511113662464	Newclassic, GKMC
459626536690520065	Wellbetheoldestonesthere, noshame
459626537109950465	ilkley
458237537689944064	Trndnl, HappyEasterSunday

We can clearly see in hashtags such as “Wellbetheoldestonesthere” or “HappyEasterSunday” we would have a problem as these words would be detected as misspellings when in fact they are not misspelt the hashtag is simply a sentenced concatenated together.

Another reason there is no spell checking implemented is that the context of the tweet is important for determining what the hashtag is. Take the hashtag

“ilkley” this could either be talking about a town in West Yorkshire or it could be a miss spelling of the world likely. If we choose wrong we would make our results less accurate as tweets with no correlation with each other would be clustered together. Making the spellchecker context aware is not a feasible goal given the time limit to develop this project.

It is because of these two reasons I have not implemented this feature. As will be explained later on in the report most of these spelling errors will not matter anyway as they have will have a very low frequency and so they will not ever be chosen to be part of the dataset to be clustered to detect events.

No Scalability by running additional instances

Originally the programs in this project were going to be scalable by launching additional instances of the program. For instance the FrequencyCounter program works by performing map reduce on the tweet data to count the frequency of each hashtag that appeared in the dataset. If two FrequencyCounter programs were running than half of the work would be done in each program (whereas in the traditional approach we would simply perform the work over 2 processes).

This has many advantages over the traditional route. The biggest advantage that we gain from implementing scalability by running multiple instances comes from the fact that we can easily implement automatic load balance. This is because it is very easy to start up or shut down instances of a program whereas it is harder to get an already running program to use more or less processes.

Another key advantage that this offers compared to the traditional route is that we are not dependent on a single program to get our solution. If one of the FrequencyCounter program crashes then we will still be to get our solution as there will be some FrequencyCounter programs still running whereas if we were performing the work over multiple processes in CPython if the main program crashed then all our processes would die with it.

Despite these key advantages I have decided to scale my programs by running the work on multiple processes rather than to run the work on multiple

programs. The key reason I have done this is because of the extra overhead caused from dividing the work up on multiple programs. To demonstrate this I have created some test programs (which can be found in the appendix).

Each program will start with a range of numbers as an input (1 to 100, 1 to 1000 or 1 to 5000). In one program these numbers will be added to a queue. If a process is not busy it will take a number from the queue multiply it by 2, sleep for half a second (this is to represent that the program is doing complex work) and then send the result back to the master program. Once the master program has received all the results from the processes it will return the sum of all the returned numbers.

The other program will be split into two sub programs (Program A and Program B). Program A will take each number it receives as an input and send it to a series of instances of Program B. Each instance of Program B will multiply the number it receives by 2, sleep for half a second (this is to represent that the program is doing complex work) and then send the result back to Program A. Once Program A has received all the results from the instances of Program B it will return the sum of all the returned numbers. These programs will communicate with the AMQP protocol which will allow the sending, queuing and routing of messages in an asynchronous manner so we are not constantly waiting to make sure the message has been sent.

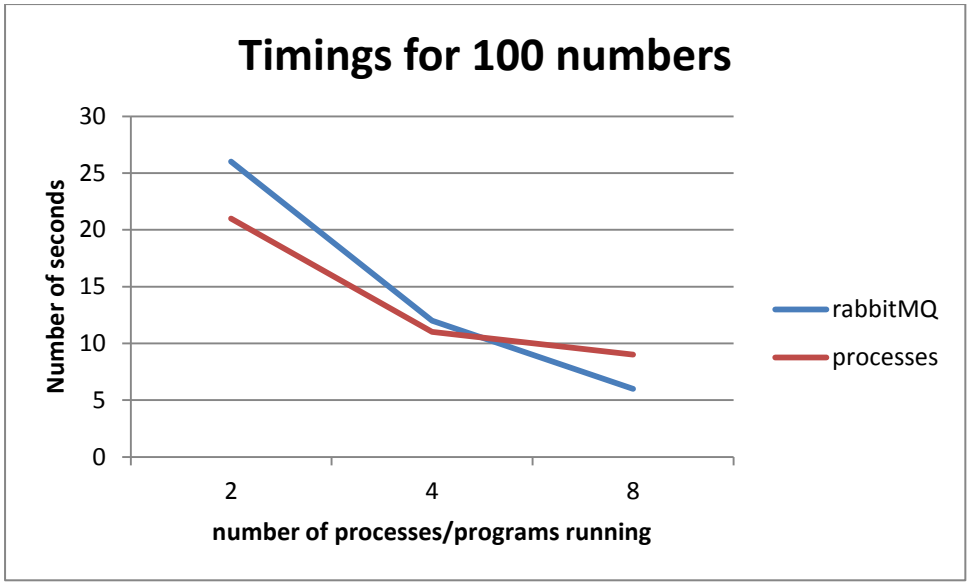

Figure 9

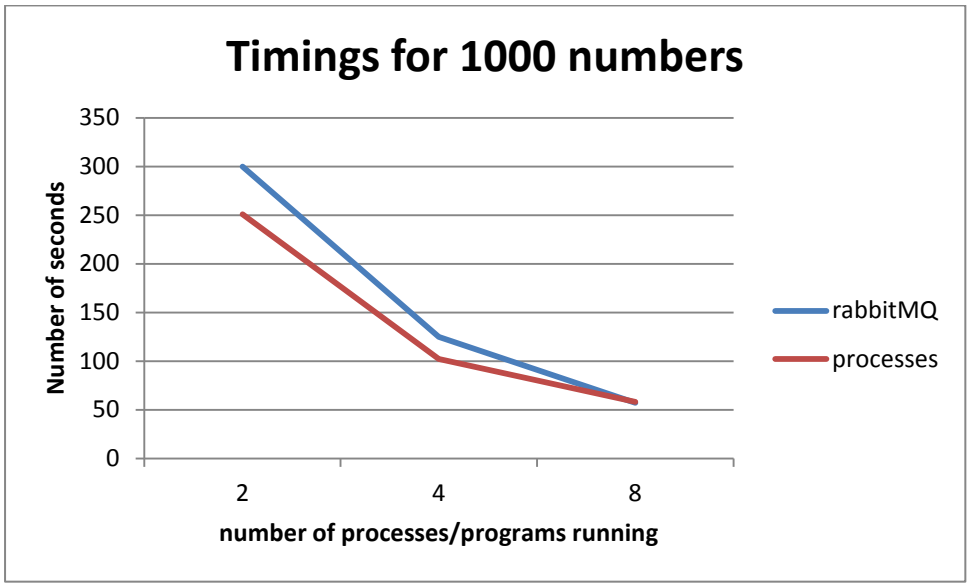

Figure 10

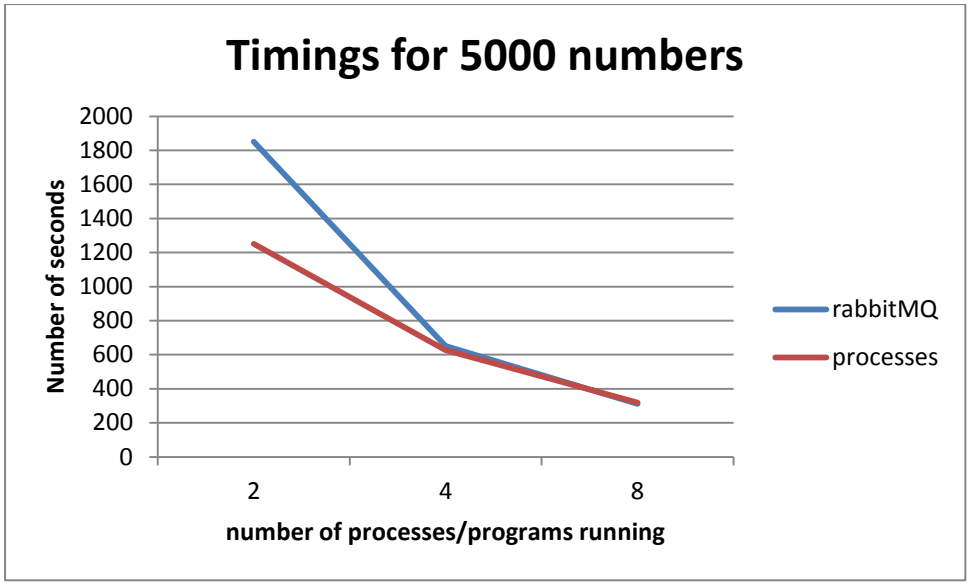

Figure 11

As we can clearly see from these results in figures 9 till 11 except for when we are running a high amount of processes (this can be attributed to the hardware I ran the tests as the laptop I ran the tests on only had two cores) performing the same tasks using multiple processes is a lot faster than if we performed the same task using multiple programs coordinated with the AMQP protocol.

Implementation

Parallelism

The multiprocessing module was used to parallelise certain tasks performed by the project. A process object in CPython is an abstraction which allows the use of running another python process on the underlying operating system while also providing it code to run. Furthermore process objects allow the new python process to be controlled by the parent program. However processes on CPython are expensive to create or destroy as each process runs its own address space.

The key advantage to using the multiprocessing module is that it is a high level abstraction to multiprocessing in general. With the help of managers sharing objects between processes can automatically be left to the python interrupter to deal with.

The multiprocessing module was chosen over threading (threads in CPython are lightweight as all the threads run in a single process) due to the global interrupter lock in CPython[6]. As memory management is not thread safe in CPython multiple threads cannot be allowed to execute bytecode at the same time. As event detection is heavily CPU bound it is quite possible for threading to become a bottleneck on the program. The multiprocessing module does not suffer from this problem as each process is run in its own address space.

Each task that requires multiprocessing in this project works the same way as every single task shares the functionality that they wish to manipulate a dataset to create a new dataset (whether the new dataset is the count of the frequency of hashtags or the grouping of similar hashtags).

The first thing that should happen is that the processes should be initialised and started. The code to run on each process should be implemented in a method called worker() which should take no parameters.

```
processes = [Process(target=self.worker) for i in range(poolSize)]

for p in processes:

    p.start()
```

Once each process has been successfully started we should loop over the dataset that we wish to manipulate and add each element to a queue. We do this as we are under the presumption that loading the entire dataset into memory cannot be done. Once items in our dataset have started to be added to the queue the processes can pop items from the queue and perform whatever transformation needs to be done on the data. This works because the multiprocessing queue data structure implements a pipe (the objects responsible in the multiprocessing module for message passing) so when new data is added to the queue each processer is in turn notified of this. The multiprocessing queue structure is processer safe so multiple processes cannot remove data from the queue at the same time.

```
for item in dataset: # loop over dataset and add the items to the queue

    self.queue.put(item)
```

As we are slowly putting items from our dataset in the queue we need a way signal to the processes that there is no more data to process. This can be solved by making the last item in the queue an empty value. If a process receives this value it will terminate. However this solution is inadequate as it would only lead to one process being closed therefore before the process terminates it will need to add an empty value to the queue which another process will eventually read and then eventually terminate. The base code for the worker method looks like this:

```
def worker(self):

    while True: #keep going until we get a value from the queue

        value_from_queue = self.queue.get()

        if value_from_queue is None: #terminate the processor

            self.queue.put(None)

            return

        #perform parallelism here
```

Each task that requires multiprocessing in this project needs to create some sort of new dataset. Therefore the results of individual workers need to be merged together. This is

done by using proxy data structures such as a proxy list or a proxy dictionary. A proxy data structure is a data structure that acts like its intended data structure without actually containing any values.

To create a proxy data structure we need to initialise a manager object. Once initialised the manager object will spawn its own process. The manager process contains the real data structure inside of it while the other processes contain fakes.

Whenever a proxy data structure is updated it will communicate with the manager object to updates itself. Finally whenever data is retrieved from the proxy data structure the proxy will ask the manager process for the real data. Unfortunately this means there is latency in access data from the proxy however this is a small trade off considering the advantages gained such as the abstraction away from message passing.

Due to a bug in CPython it is not possible to update key value pairs in sub dictionaries inside a proxy object. To solve this solution a temporary variable is created which contains the sub dictionary. The temporary dictionary is then modified. Finally the sub dictionary is overwritten by the temporary dictionary.

```
fake_dict = real_dict[key]

fake_dict[dict_key] = new value

real_dict[key] = fake_dict
```

Twitter scraper

The core functionality of the Twitter scraper is performed with the help of a library called Twython. The advantage of using Twython is that it means the scraper can abstract away from the wire protocol as it is now Twython's job to handle communicating with Twitter using REST. This saved countless hours of development as it meant Twitters API did not have to be learnt or implemented in the project. Twython also deserializes the JSON string turning it into a python dictionary.

The on_success method of the Twitter scraper is where the data is transformed into the format required for the database.

```
timeStamp = datetime.datetime.strptime(data.get('created_at'), '%a %b %d %H:%M:%S %z
%Y').timestamp()

coordinates = data.get('coordinates')['coordinates']

tweet = {

            '_id' : data.get('id_str'), 'hashtags' : data.get('entities')['hashtags'],

            'datetime' : timeStamp, 'text' : data.get('text').split('#')[0],

            'loc' : coordinates
```

```
}
```

Once we have transformed the data we can append it to a list and if the list is big enough we can perform a bulk write operation to save the contents of the list to Mongo DB. As previously mentioned by doing this we spend less time performing I/O per minuet which allows us to spend more time collecting tweets.

The YAML configuration file deserialization is handled by a library called PyYaml. This library will transform the configuration file into a dictionary object. This dictionary is further subdivided into the dictionaries oauth, locations and Twilio as previously mentioned when discussing the Twitter scrapers class diagram in the design section.

Database

The database class makes use of a library called MongoClient to implement communication between CPython and Mongo DB. One of the many features that are gained from using MongoClient is that the results from querying the database are returned as iterators created from a generator function.

Generators are created with the yield keyword in python. When the generator function is called it is actually never run instead it will return an iterator object. When we begin to run the iterator the function will execute until it hits the yield statement. Then for every time the next item in an iterator is requested it can be retrieved by rerunning the part of the function that contains the yield.

This can actually lead to a performance increase in the program as when we request the results of a query the generator returns the iterator object instantly. The query itself is actually never run until we physically require the data in the program.

The other key advantage of this technique as previously discussed in the Introduction is that we can fit a dataset of any size into our iterator as the values are never loaded into memory.

Having datasets as iterators has been challenge to implement properly as iterators can only be run once. Therefore iterators cannot physically be used when we need to cluster hashtags together as K means clustering is an iterative process. To solve this we instead store the datasets in memory once we have successfully detected outliers. The idea behind this is that the dataset should be able to fit in memory once we have filtered away the outlier especially as at that stage we only require the Id's of tweets and their respected hashtags.

As Mongo DB implements geospatial queries it becomes trivial to find all tweets in a given location.

```
data = self.db.find({"loc": {"$within": {"$box": [[minX, minY], [maxX, maxY]]}}})
```

However one major drawback to using Mongo DB is that the database can only index on 2d data (X and Y) whereas we need search for data in a 3d area (x, y and time) we can partial solve this by using two indexes (find a list of all tweets in an area in O(log(N)) time and then find all tweets in the given time period from that list O(K) time) however this is still not ideal as it is slower than if we had a 3D data index.

One solution to this might be to move the database to Couch DB which like Mongo DB is a document based database with similar features such as sharding, map reduce and replication. However unlike Mongo DB Couch DB has a more detailed feature list for geo-spatial data querying including being able to index more than 2d data. Unfortunately there is not a mature python library for communicating with Couch DB which is why the project currently uses a Mongo DB database for the backend.

Event Detection

As previously mentioned in the design report the Hashtag counter class and the Text counter classes are extended from the dictionary class. The most important feature difference between these classes and the dictionary class is that they are immutable once the MapReduce has been performed on the dataset. This is because of their internal representation. As an example imagine the internal representation of a HashTag Counter object:

```
{
"spam" : {"ids" : [1,2,3], freq : 3},
"ham" : {"ids" : [4,5,6,7], freq : 4},
"eggs" : {"ids" : [8], freq : 1}
}
```

The data structure for storing the list of tweet ids belonging to the hashtag is actually a hash set which is immutable. Therefore the entire class needs to be immutable as the internal representation cannot be changed. Another key reason these objects are immutable is so that they are thread safe which may become important in the future if improvements to the Global interrupter lock are ever made (as this is the only thing that is currently forcing the implementation to use multiprocessing module over the threading module).

The list of Id's is stored as a hash set as it allows for O(1) set operations such as retrieving a list of all tweet id's that two hashtags do not share. This is important as set operations are needed when we cluster the data to detect how similar hashtags are and as the clustering is done in O(K * N * I) time it would be wise to make detection of similar hashtags as quickly as possible.

The HashTagCounter class performs its MapReduce as one function on the parts of the data from the data received from the queue.

```
for tag in tweet['hashtags']:

        tag = tag['text']

        tag = tag.lower()

        if tag in self.tagFreq:

            counter_dict = self.tagFreq[tag]

            counter_dict["freq"] += 1

            counter_dict["ids"].add(tweet["_id"])

            self.tagFreq[tag] = counter_dict

        else:

            self.tagFreq[tag] = {"freq" : 1, "ids" : {tweet["_id"]}}
```

The map() section of the code is hit if it is a hashtag we have never come across before. If this is the case then the Tweet data is transformed into a key value pair of:

Hashtag:`{"freq" : 1, "ids" : [tweet id]}`

If that hashtag already exists then there is no need to perform a map() operation as we only need to retrieve the id from the hashtag and we can then simply merge the new id's with the list of existing id's and increment the hashtag's frequency by 1. The same logic is used by the WordCounter to count the frequency of words using MapReduce().

This approach is different to the traditional Map Reduce implementation because the traditional method was designed to work with multiple different Reduce functions() and as such not all of these functions can be mapped and reduced in one step this however is not the case for counting.

Both of these objects implement a most_common method which will return X words/hashtags that have the highest frequency. This method was only implemented for testing and should not be used in a real implementation. This is because the internal representation of the objects gets converted to a list so it can be sorted. This however means that the entire dataset is loaded into memory which could cause errors if the dataset is larger than the available memory.

The Text counter class also has another function then to count the frequency of words in the text. The text counter is also responsible for finding a word's lexical category and keeping track of the ratio of verbs in the text compared to non-verbs. This is done with help

from the textblob library which compares words to a previously collected lexical database to find the words category.

Using a previously collected lexical database was a fast method on working out the lexical category for a word. This method however is not ideal as it is dependent on the database not being wrong or outdated. Furthermore the majority of lexical databases only support the English language which would mean the event detection program can only detect words in English. Unfortunately there is no easy solution to this as natural language processing is a very complex research area additional time would be needed to investigate what the best option would be to detect verbs.

The detect outliers class works by looping over two Hashtag counter objects (one stores all the hashtags for the given space and time while the other stores all the hashtags for the given space). As we know how long the datasets as a time is and we know frequency of the individual hashtag we can work out the standard deviation with the following formula.

$$= \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Originally the standard deviation used N as the sample variance instead of N-1. Friedrich Bessel[7] proved that by Using N as the sample size we get an increased mean squared error in the estimation of the standard deviation and therefore to reduce population bias we should use N-1 instead of N.

Once we know the standard deviation and the mean frequencies for a given hashtag we can compare the hashtag for our given space time area with these values to see if it is larger than mean * 2.5 standard deviations. If it is then we have an outlier and we can add it to a list otherwise we can discard the hashtag.

As previously mentioned the DetectOutliers class returns the outliers as a list rather than an iterator. This is because clustering is an iterative process and so we would need to loop over the list of outliers more than once (which is something an iterator is not capable of).

The get_biggest_outliers() method uses CPython's inbuilt sorted function to automatically sort the hashtag outliers by how large their frequency was. The list was sorted on frequency rather than standard deviation as hashtags with a very high outlier are often hashtags that are used rarely rather than hashtags pointing to an event.

The clustering implementation uses the K means algorithm to attempt to cluster similar hashtags together. The degree to which two hashtags are similar is calculated by the similarityCalculator class. To calculate the similarity we use the Jaccard similarity coefficient and look at the hashtags Id's. If two hashtags share Id's then there is a high chance that the two hashtags are related. The formula for working out the similarity looks like this

$$\frac{A \cap B}{A \cup B}$$

The key reason we use this formula is that we binaries the Id data (a list of two hashtag's ids can be represented as a list of 1's and 0's where 1's are matches and 0's are non-matching). If we had used a different formula such as the cosine coefficient we would have received distorted results as the 0's would have matched tricking the formula into believing that every single hashtag is 100% similar to any other hashtag. The result from using Jaccard's similarity coefficient will give us a number between 0 and 1. The closer the number is to 1 the closer it is that the 2 hashtags match.

K means clustering[8] "*is a method used to automatically partition data into k clusters".* The algorithm psudo code can be found below

```
Build initial centroid list

While a centroid in the list has moved:

        Start X proccess that listen for work from the queue

        For each hashtag that is not a centroid:

                Add to the work queue


        Wait for processors to finish

        calculate new positions of the centroids

def worker():

        if no more values in queue terminate

        else retrieve a hashtag from the queue

        For each centroid:

                calculaterJaccardSimilarityCoefficient(centroid's ids, hashtag's ids)

        compare the coefficients to find closest centroid

        if all of the distances are lower than 0.1 discard the hashtag

        otherwise add it to the centroid list
```

As previously mentioned the starting centroids for the clustering are chosen at random (as long as they have a high enough frequency). As outlined in the Parallelism sub heading of the implementation section the clustering uses a queue system and multiple processes to attempt to speed the run time of the program up. This works well as the K means clustering process do not need to communicate with each other (except to notify each process to finish) which makes the algorithm very scalable.

One large problem with this clustering implementation is that it requires the user to specify the number of clusters to generate. This does not work well because users will not know how many events they need to find and so they will not know the number of clusters to generate.

The final part of the event detection program is the generation of the word cloud. This is done through a library called Pillow which is an image creation library in python. Creating the tag clouds is a simple process and can be seen in the algorithm below

```
For each word in the word counter:

        Create our typeface size by 14 + (2* the words frequency)

        Find the words lexical category and select the font colour based on that

        Loop:

                Attempt to draw the word to the image at a random position using the given
typeface

                If the word is touching any other word move it to a new position

                Else draw it to the screen

Save the image()
```

In the WordCloud program we detect if two words intersect by using rectangle collision detection. Two rectangles are considered not touching if one rectangle is either absolutely left, right, above or below the other rectangle. Therefore we can draw imaginary rectangles around each word on the screen and check if any of the words rectangles are touching.

Using this brute force method is problematic as there might not be any empty space to put the word down. If a word cannot find a place after 250 iterations it will be placed in a random position. In an ideal world this should be handled more elegantly however the WordCloud is just there to give a visual representation to the event and should not really be considered a major part of the program.

Future considerations

Although much has been achieved in this project there are a few features that could have been implemented to make the results more accurate and make the project more scalable. Furthermore there are some implementation details that could have been changed that could have increased the performance of the program.

The biggest change that should have been made to the project was to develop the project in PyPy instead of CPython. PyPy is a Python implementation that makes use of JIT compiler to transform the byte code of the project into machine code. This means that we no longer have to wait for all of the byte code to be interpreted before we can start executing instructions. Instead byte code gets interpreted into machine code when it is needed which leads to faster runtime performances.

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

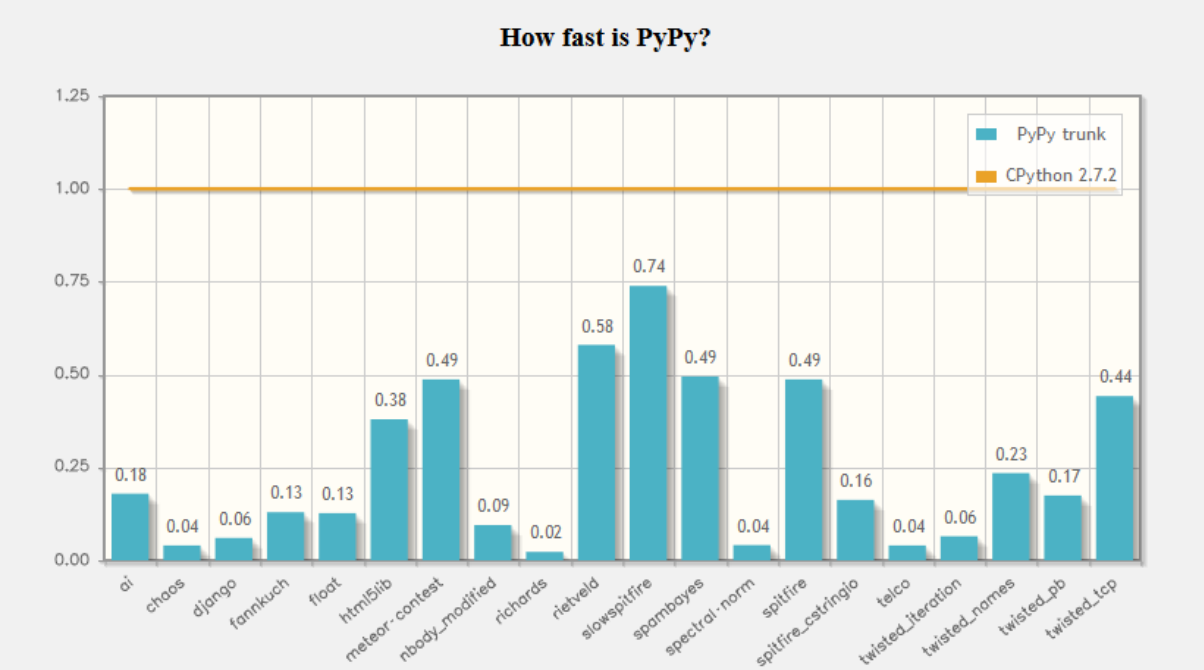

Figure 12

As seen in figure 12[9] which is a benchmark of PyPy programs compared to a python 2.7 program by using a JIT compiler PyPy on average performs 6.3 times faster than CPython on the benchmark tests.

This project was not originally developed in PyPy as some of its dependencies were not compatible with PyPy. However this has since been fixed and so there is no reason to not move to PyPy compared to CPython.

A feature that should be looked into as a future consideration would be to ignore tweets in the dataset that do not have any likelihood of mentioning an event. This could be done by looking at the structure of the sentence. If the tweet is talking about an event it will need to mention a location, a person or a planned public/social occasion. We can use Bing's synonyms API (which returns any alternative reference names to a real world entity) to detect if any words in the sentence match these criteria. As the synonyms API is built up by Bing's search engine there would not be a problem of us comparing to outdated synonyms. However this would need further investigation to see if it is possible to implement while still maintaining suitable performance.

As discussed in the implementation section this project makes use of CPython's multiprocessing module. A large drawback to this approach is that each process has its own memory space which creates a large overhead for the program. Unfortunately the multiprocessing module is the only suitable way to perform parallelism in CPython.

However it is possible to write the performance critical code in C and to then parallelise the code using OpenMpi or OpenMp. CPython (and PyPy) can then call the C code and treat it as a CPython extension. This is all possible due to the “Python.h” header file for C which allows you interface code between CPython and C.

This was not originally implemented to keep the project simple as problems can be caused when we interface between C and CPython. For instance CPython garbage collector collects objects by looking at their reference count. When an object is referenced its reference count is increased and vice versa. However this can lead to a situation in which CPython clears the object from memory while the C program is still using the object. Edge cases like these need to be handled but because of the complex nature of the edge cases it would be impossible to implement them in time given to develop the project.

In an ideal environment users should be able to choose which database to store their dataset in. Therefore a final future consideration that should be looked into is to decouple the implementation of communicating to Mongo DB from the database class as currently users are forced to collect and store their dataset in Mongo DB due to the limits of Mongo DB and geospatial data. This is further expanded upon in the results and evaluation section.

Results, Evaluation and conclusion

To successfully see if it is possible to detect events with the event detection program 2 weeks’ worth of data was collected over a period of time in the following locations London, Birmingham, Manchester, Leeds and Liverpool. These places were chosen due to their high populations.

This program will attempt to discover events from 19th April 2014 at 17.30 until the 19th of April 2014 at 20.30 at London, Leeds and Birmingham. These 3 places were selected at random. The Date and time was chosen as Saturday from that time is traditionally when Twitter has the most people online and tweeting.

Each run of the program will attempt to find 25 events. The main comparisons of the test shall be done on what the program recognises as the 3 biggest events. Secondary comparisons of the program will be performed by manually inspecting the 25 tag clouds and attempting to find events by hand.

In the first comparison test we are primarily interested in seeing how likely that the top 3 tag cloud suggested are actually events. While In the secondary comparison we are interested in data such as how many of the 25 clusters are actually events and is there an overall correlation between high ranking and the tag cloud being about an event.

A final test will be performed to attempt to see how scalable the program is. This will be done by increasing the number of processes used to work on the data while gradually increasing the sample sized used.

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Results London

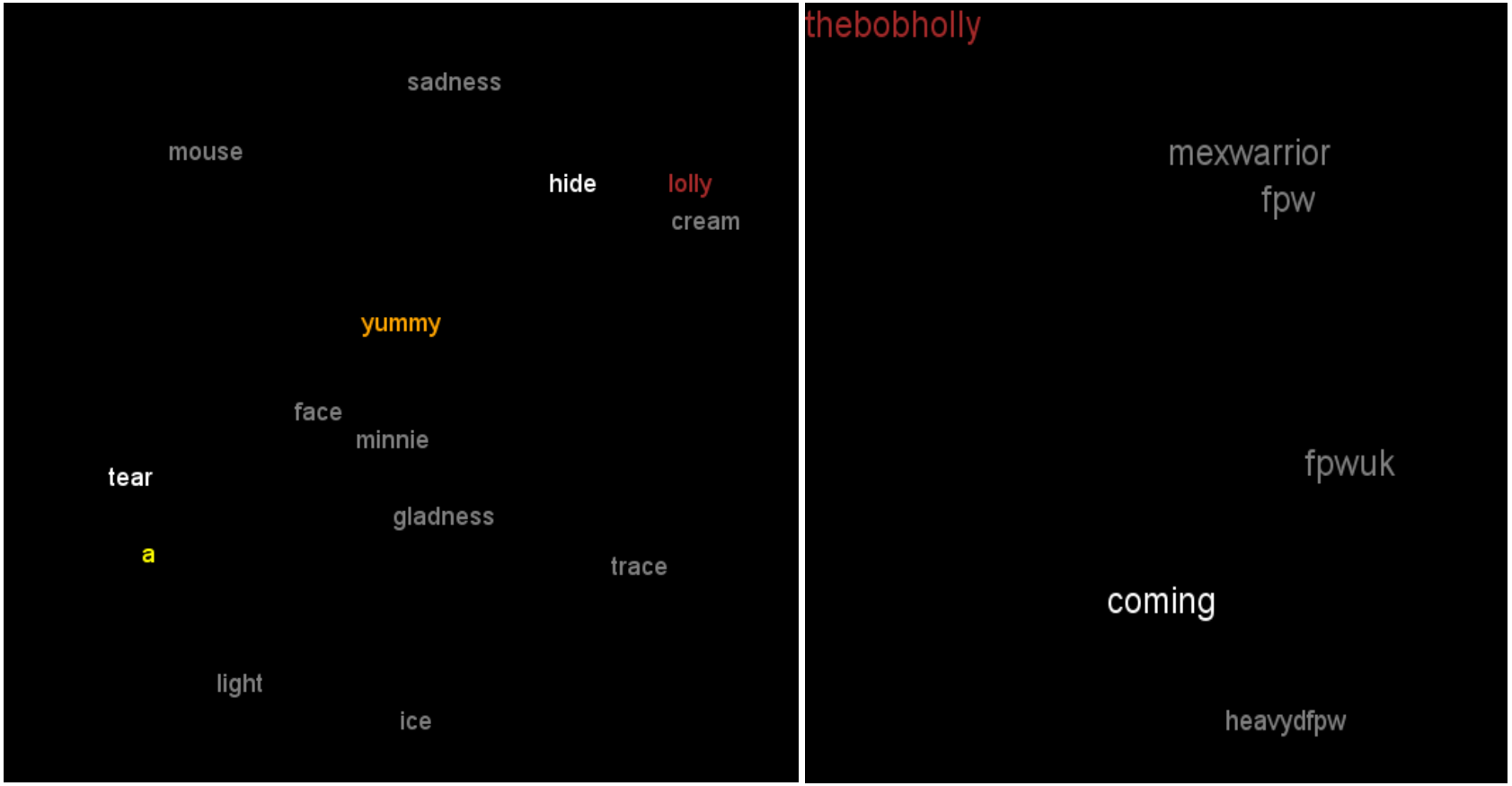

Figure 13

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Figure 14

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

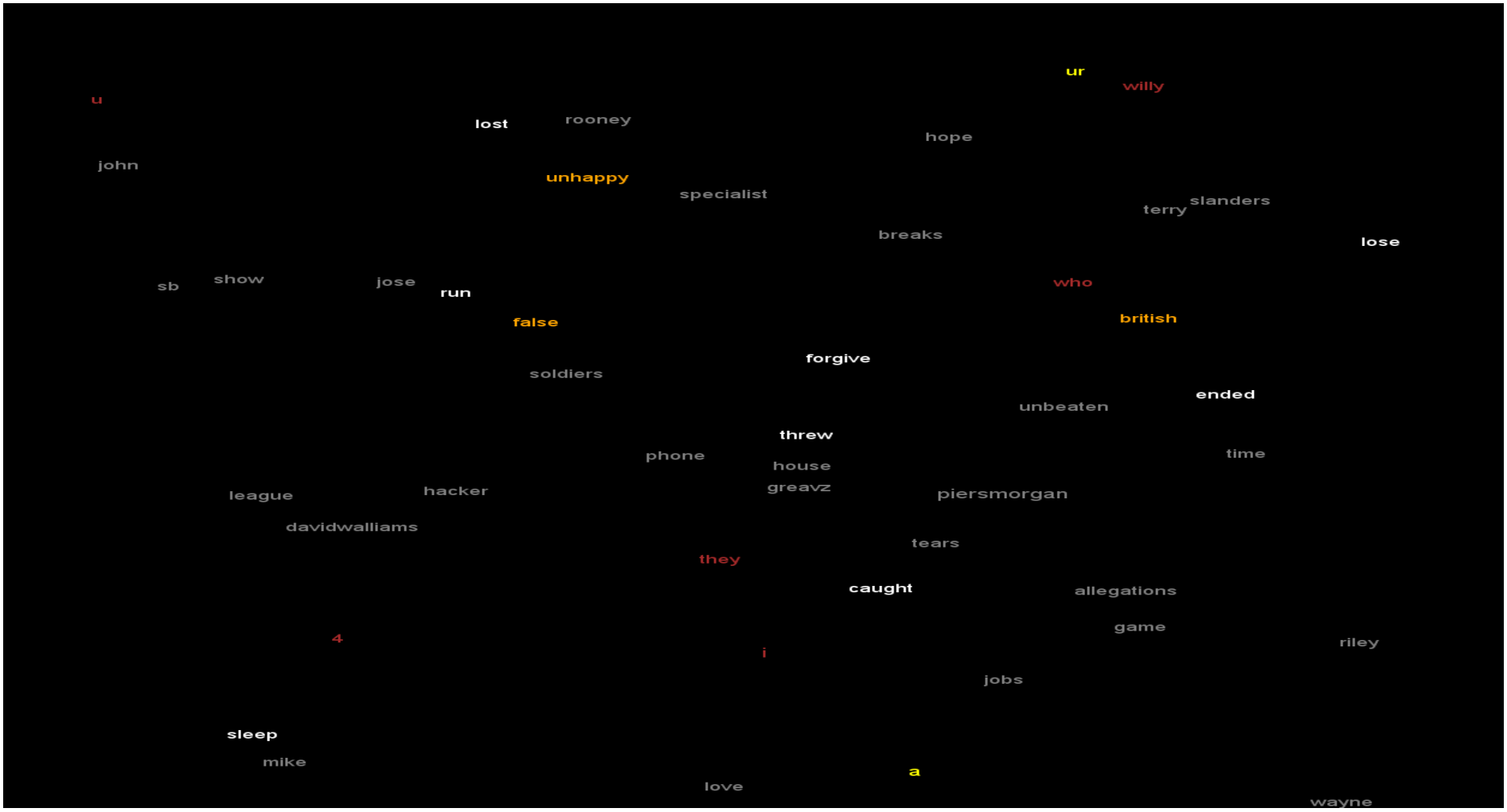

Figure 15

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Results Notable events detected in London

Britain's got talent ranked 12th

http://www.dailystar.co.uk/showbiz-tv/britain-s-got-talent/375145/Britain-s-Got-Talent-2014-Simon-Cowell-risks-death-with-knife-throwing-pre-teen

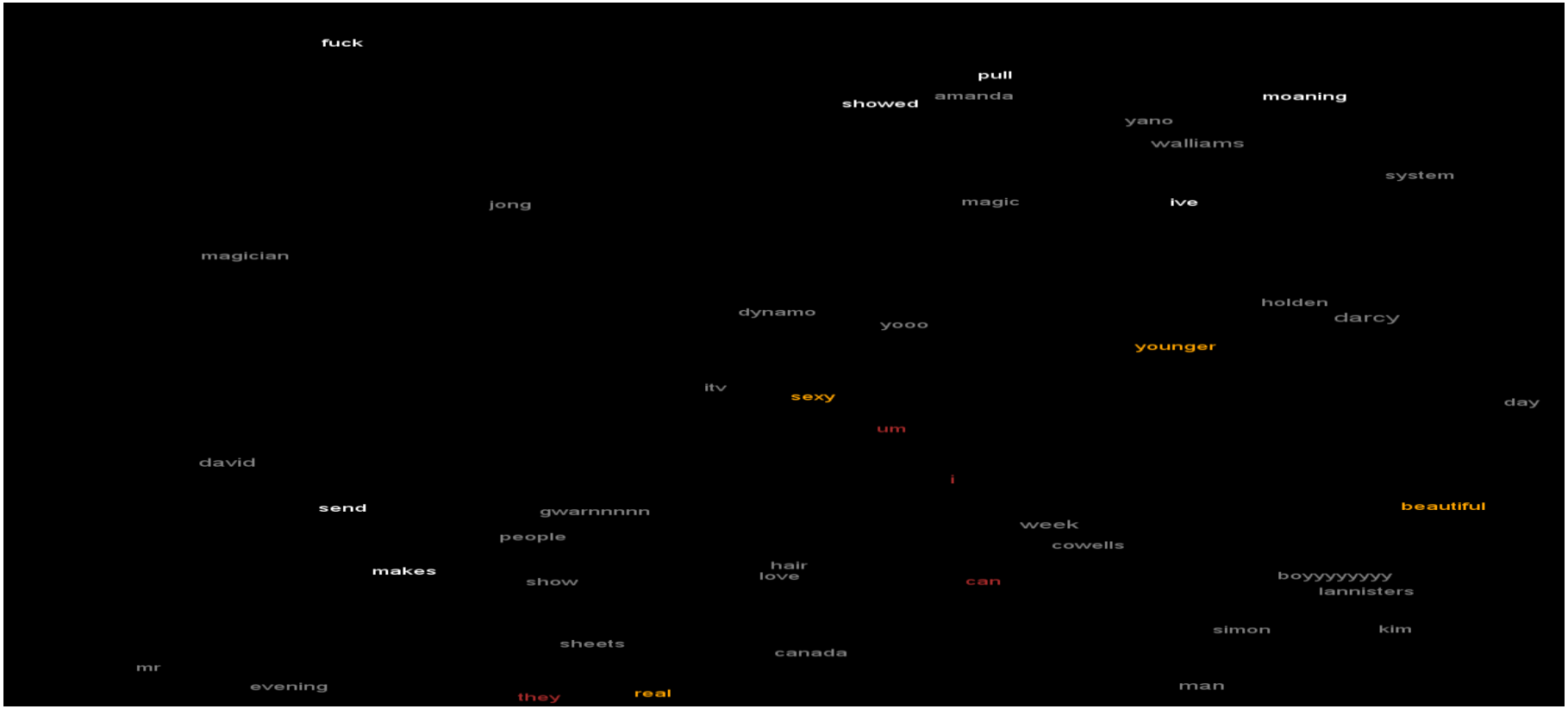

Figure 16

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Sunderland beat Chelsea 2: 1# Ranked 13th

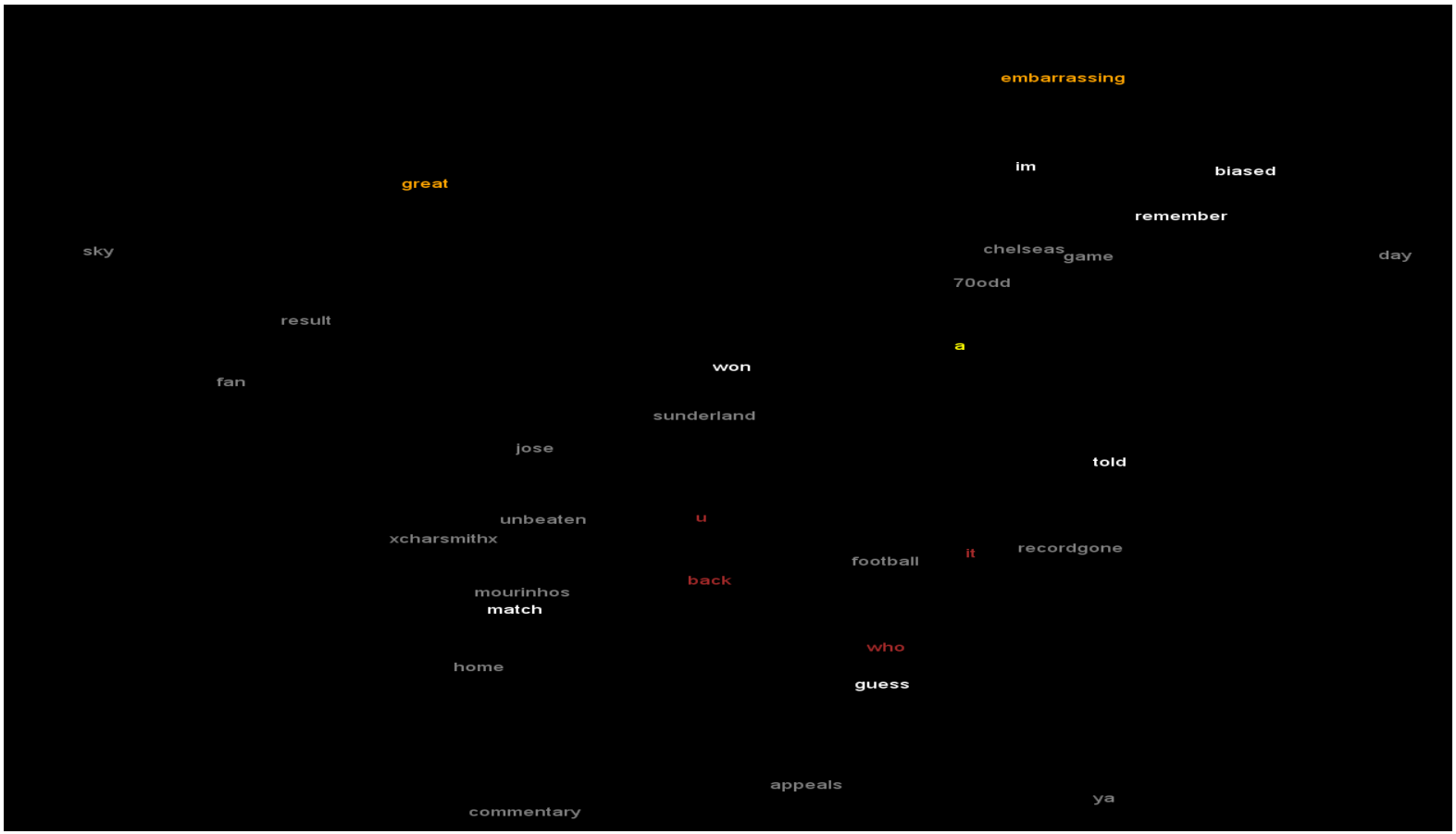

Figure 17

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Channel 4 show Taken Film

#Taken ranked 25

http://www.channel4.com/tv-listings/daily/2014/04/19

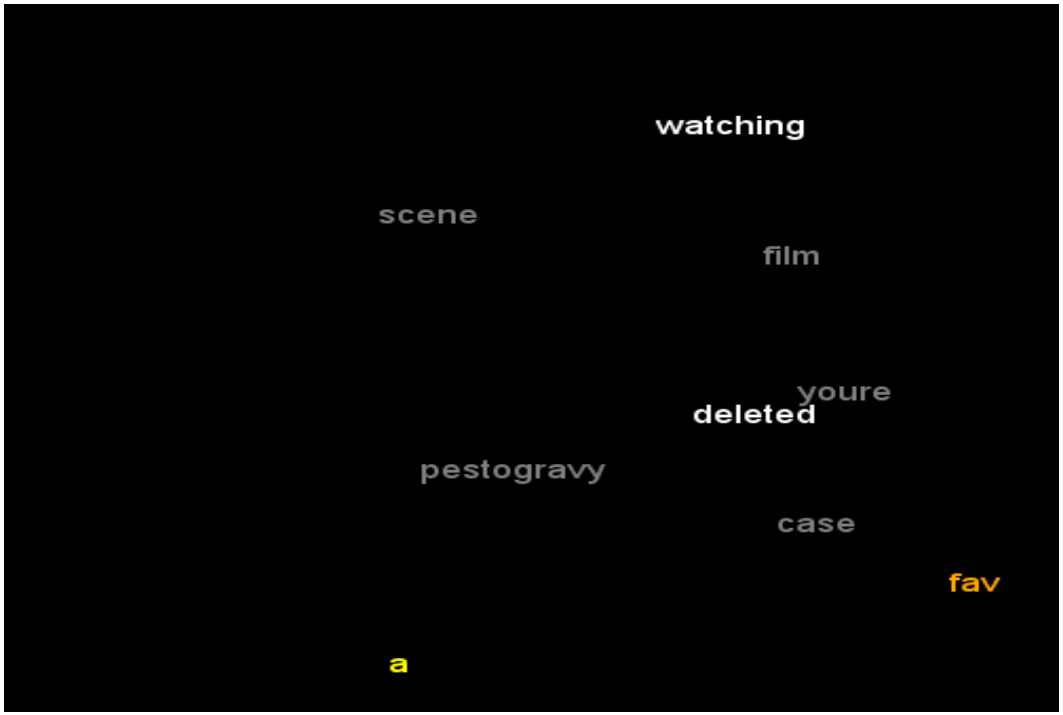

Figure 18

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Blur Park life 20th anniversary special

Results Birmingham

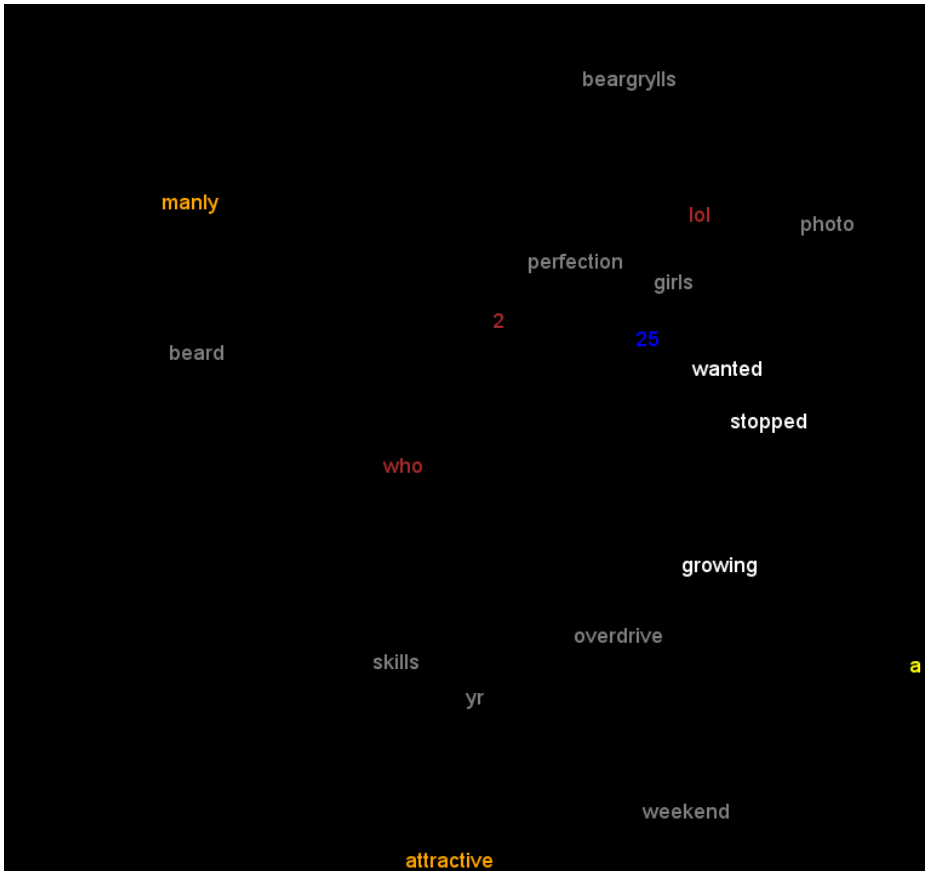

Figure 19

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Figure 20

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

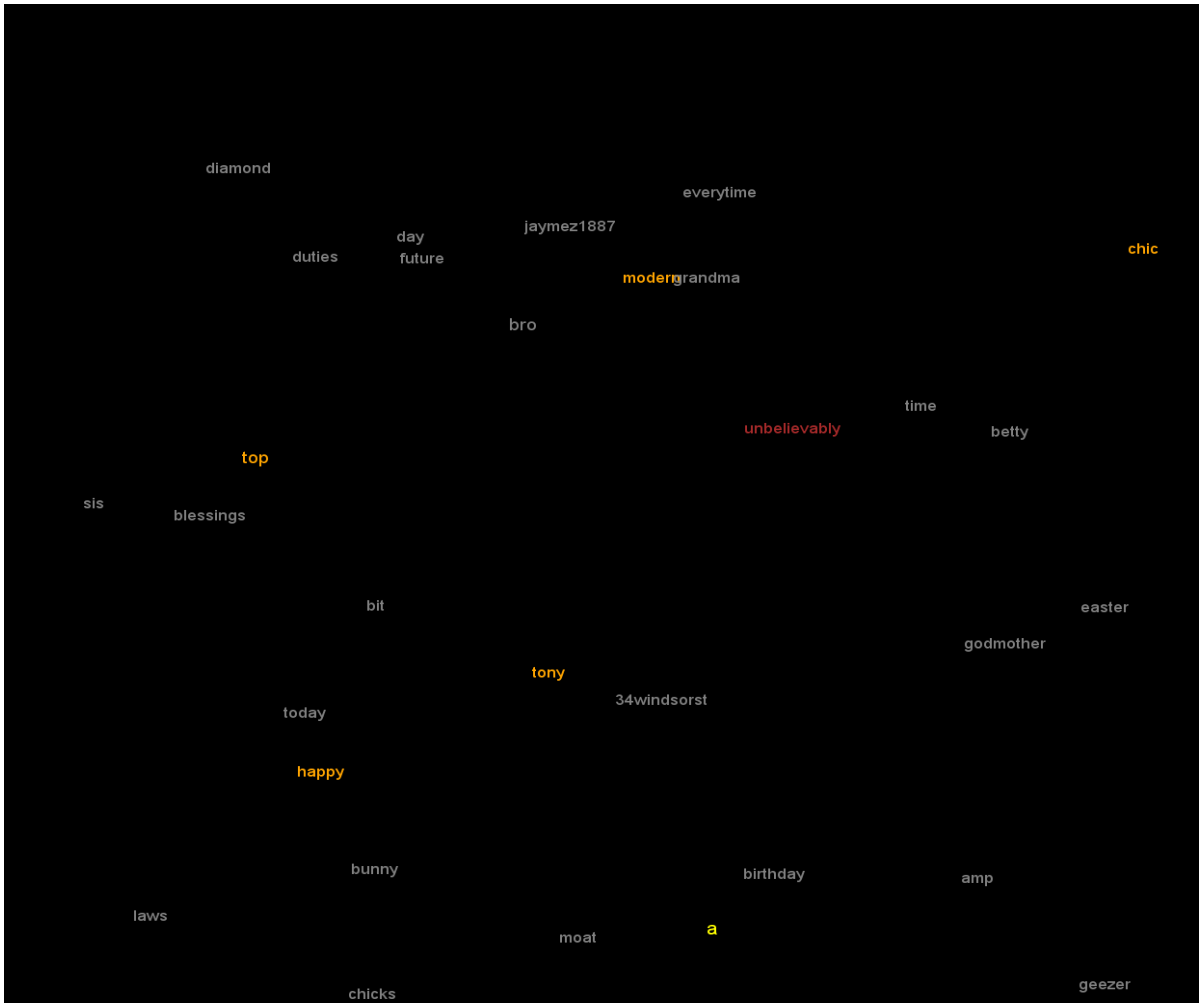

Figure 21

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Results notable events detected in Birmingham

Record store day rank 12

http://www.recordstoreday.co.uk/

Figure 22

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Easter Sunday tomorrow Rank 19th

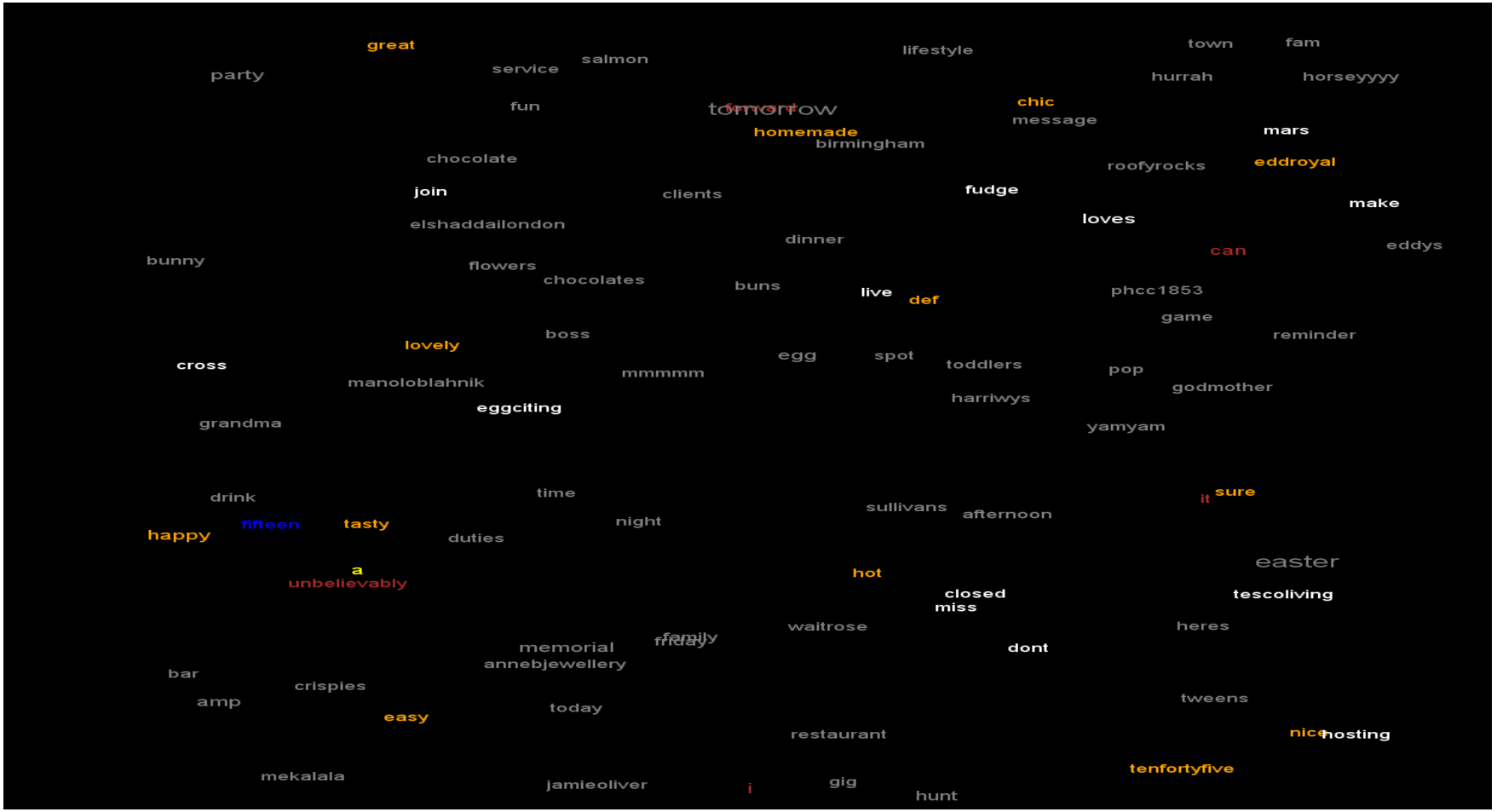

Figure 23

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Results Leeds

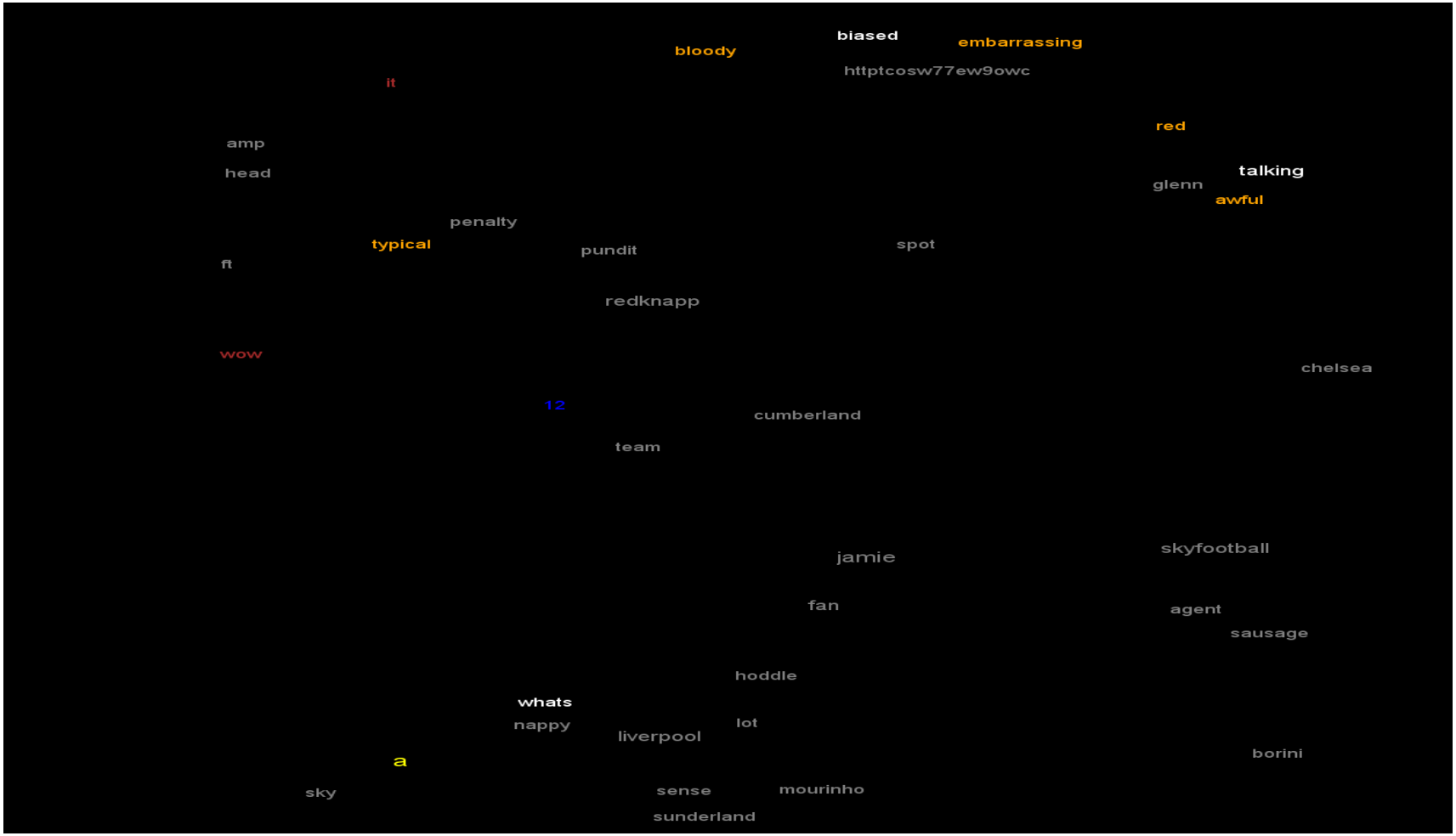

Figure 24

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Figure 25

Figure 26

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project
Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Results notable events found in leeds

Britain's got talent Ranked 11

http://www.dailystar.co.uk/showbiz-tv/britain-s-got-talent/375145/Britain-s-Got-Talent-2014-Simon-Cowell-risks-death-with-knife-throwing-pre-teen

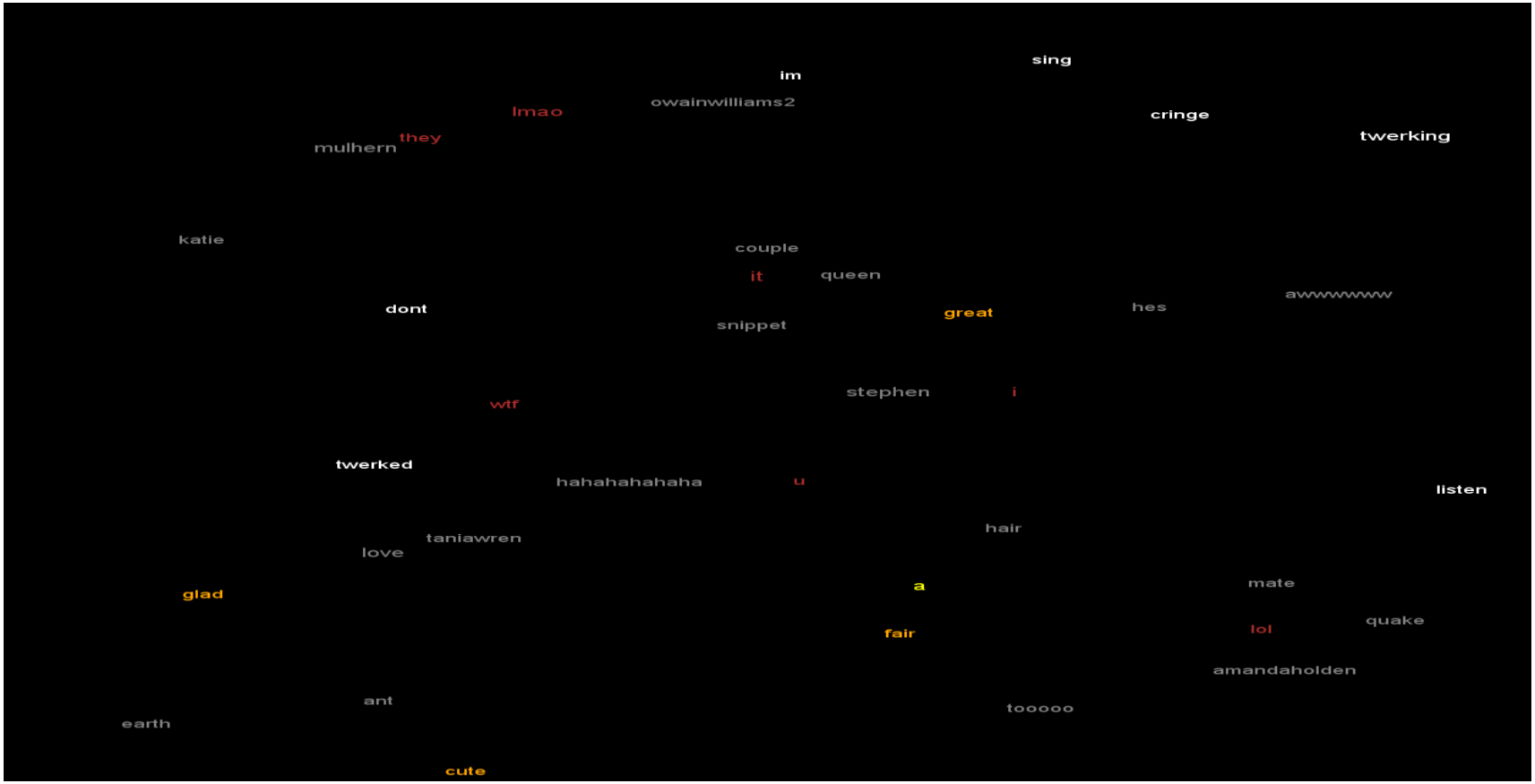

Figure 27

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40

Evaluation of results

As we can clearly see from this data there is no correlation between events and where they appear on the list of tag clouds. This is suggested by the fact that in 3 cities none of the top 3 spots are occupied by any kind of event. This is further proved by looking at the manually picked out events which have rankings ranging from 11^{th} all the way to 25^{th}. These two tests prove that trying to filter the event clusters from the non-event clusters by looking at verb usage does not work at all.

In the dataset of the 25 tag clouds London had 3 events, Birmingham had 2 events and Leeds had 1 event detected. According to the study by Pear Analytics at least 49.6% of the tweets should not be events as 40% are “pointless babble”, 3.75% are spam and 5.85% of tweets are for self-promotion. That said on average 0.06% of all of the word clouds that were generated are event based. This suggests that the outlier detection algorithm was not working very well as none of the “pointless babble” was removed. From these results we can therefore draw the conclusion that detecting outliers by looking for hashtags that have a frequency greater than 2.5 standard deviations away from the mean does not work to suitable remove outliers.

It is highly likely that these results are due to a poor algorithm design rather than their being bugs in the code as the implementation was programed with Test Driven Development in mind. For each of the core parts of the project there is suitable unit test coverage and each unit test checks for edge cases. If there was a major bug in the code that was skewing with the results it would be unlikely that it would remain hidden while all the unit tests pass.

Although the full dataset has not been provided another key problem we can see is that not all similar events have been tagged together. A key demonstration of this can be found with the tag clouds of BGT2014 and Britain’s got Talent. These tag clouds are describing the same event however they have not been clustered together. This could be attributed to the fact that people are unlikely to tag a tweet as both #BGT2014 and #Britans got talent as they mean the same thing.

Although there was an oversight on how the hashtags where clustered together as previously discussed overall the clustering was a success. This can be seen from the tag clouds in which the words inside the clouds make sense for that particular topic. This can be further seen from the more miscellaneous tag clouds such the one in figure 28 in which lots of different conversations have been merged (we have talks about fishing, daffodils, the forest, ducks and geese to name a few)to talk about a related topic (in this case its talking about nature).

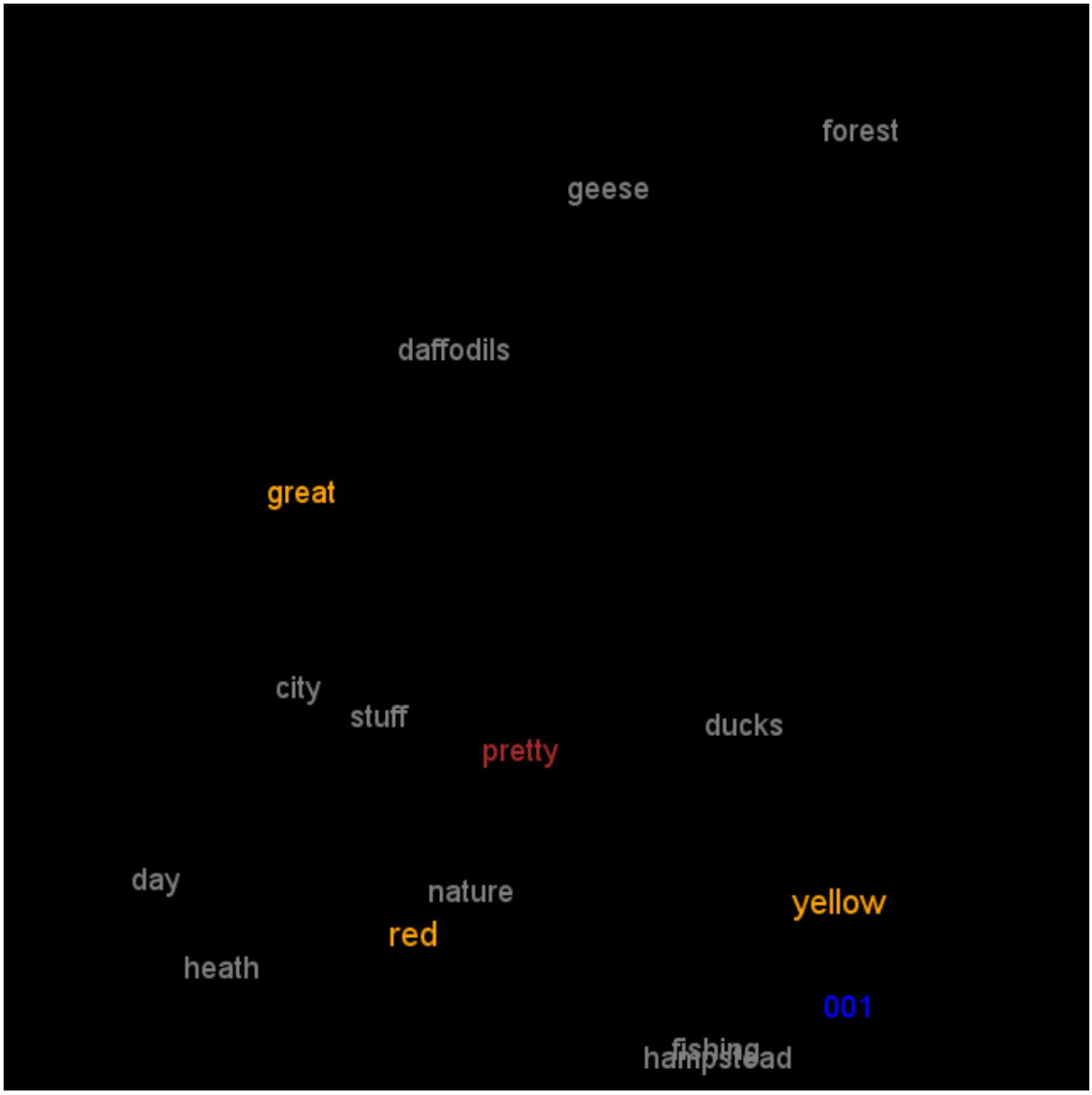

Figure 28

How scalable is the program?

The third program test was all run on the same hardware which can be found below.

Description of Hardware/software environment	
Operating System	Windows 8
Ram	4GB

Cpu	3.14GHZ Duo Core

Each program run was run 5 times to attempt to get an accurate timing average. The event detection was done over 1, 2, 3 hours and the program was run on 1, 2, 3 or 4 processes.

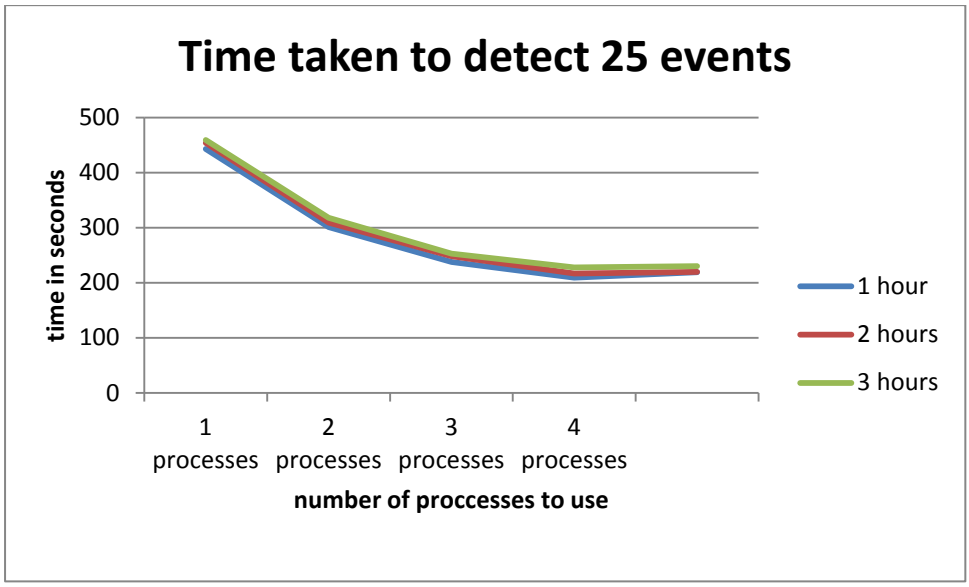

Figure 29

As we can clearly see from the results in figure 29 we can see that the program is highly scalable. One major point that we can see from these results is that increasing the time range on the event detection does not have a large impact on the results. This is probably because we have to load all the data in any way to detect mean and standard deviation frequencies of the data.

Another trend we can see in the data is that as the number of processes got higher the run time actually got worst. This is probably due to the law of diminishing returns. As we increase the number of processes the time spent communicating between the processes increases while the time to solve the solution hardly changes.

Overall Evaluation of the project and conclusion

One flaw of the event detection program is that currently all clustering is done with K Means. The problem with this approach is that the starting centroids are selected at random and some of the outliers that the project will have detected will have nothing to do with events. If these get selected as centroids then the clusters returned will often not have any data about events.

To solve this we could implement Agglomerative hierarchal clustering. Agglomerative Hierarchal clustering starts with each hashtag being in its own centroid. A cluster will then be merged together with another cluster if it is its closest partner. This process keeps going until X clusters remain were X is a specified number chosen by the user.

Using Agglomerative hierarchal clustering could be seen as a better choice than K means clustering as there is no longer any random element involved in the clustering. The major problem with Agglomerative hierarchal clustering is its runtime which is $O(N^3)$. A runtime complexity that large would cause major problems for scalability and it probably wouldn't be able to handle large datasets. Furthermore the runtime would be even slower than suggested as the dataset would be split up over multiple processes and each process will need to communicate with every over process to find the closest centroid to pair with.

A solution to this is to only cluster the top 100 outliers as this is generally enough to build events. Further testing will need to be done to see if it is suitable to switch over to agglomerative hierarchal clustering and even more testing will be needed to see if events can be reliable detected with only 100 outliers.

Another flaw of the event detection program is that the stop word list which is used to filter out unnecessary words can actually cause more harm than good. This can be seen with bands such as "The Who" in which both words would be removed the text as they are stop word even though in this context they mean something.

When it comes to scalability I feel as if this project was mostly a success as it has been proven to be able to handle large datasets without problem in a timely manner. Furthermore as previously discussed in the future works section there is a lot of room to make the program more scalable if needed.

Even thought it was costly to make processes with the multi-processing module I feel as though using this module was largely a success overall. As seen from the timings it has proved that the multi-processing module can be to scale a solution. Furthermore due to the abstractions created by the multi-processing module it will be easy to move from running on multiple processes to being able to run the program over a cluster of computers.

In conclusion I feel that although it has been proven that the algorithm for detecting events itself is flawed there has definitely been key areas in which there has been success in this project such as getting the program to scale and clustering similar hashtags together.

Reflection on learning

This project was the first major project that I have had to complete. Overall the course of the last term I have learnt a multiple of lessons. The biggest lesson I learnt along the way was to plan everything out and not attempt to rush into the project to cut corners. This could be seen with me having to re-implement the database layer and port all the tweets over as I originally tried to store the tweets into a SQL-LITE database as it was simple to implement. This caused me a great deal of headaches as it wasn't until I ran a profiler that I discovered that it was the database that was being a bottleneck for the program.

Another scenario in which I rushed was that I did not plan how I should model my data. In the original draft of the program all of the tweet data was serialized into bytes with python's pickle library and then stored in the SQL database. This was originally done as I was not sure what Meta data I would need. Unfortunately it turns out that transforming the byte data from pickle back into a dictionary was very slow and this became another bottleneck for the program.

In the future I will make sure to plan out my systems before I begin to develop them because rushing can lead to large unnecessary setbacks in the project which means that more time is wasted from an already difficult deadline to achieve.

Another major lesson I have learnt from working on this project is how to work when I became demotivated with the project. Becoming demotivated with a project this size was only natural (especially as run times where around 8-9 minutes long before I moved over to Mongo DB).

To solve my motivational problems I used two methods. The first method that I used was the chain method. The idea behind the chain method is that every day that you work on the project you get to mark it with a large cross. After working for a few days straight you become motivated to work as you now have the mentality that you do not wish to break the chain of crosses.

The second motivational method I used was to break down my project into very simple problems. I would then do the easiest and simplest task on the list. Once I had finished that I would perform the next easiest task. After a while you gain the motivation you need to work on harder features as you can see all the progress that you have managed to make.

In the future I feel like I will be better able to tackle difficult projects as these methods that I have learnt will better help me tackle any demotivation issues that arise from working on a large project.

Over the course of this project I have learnt about python best practises for programming. Before I started implementation I only knew a bit of python from my first year. I feel that my skills in python have significantly increased and I now have an appreciation for functional style programming such as using lambdas, maps(), reduce() and filter ().

Learning python has definitely translated into me being able to pick up and learn new languages in the future. This is especially true as I have learnt new programming paradigms such as functional programming.

I have also gained a strong appreciation for test driven development. Test driven development is a programming style in which you write your unit tests first before the code. By writing your unit tests first it forces you to write your code in an object oriented manner as you are forced to decouple your classes very early on as unit tests test a single logical concept and so you cannot depend on outside factors such as databases.

Finally in this project I have gained valuable experience with data mining techniques such as K means clustering, transforming the dataset and vectorization of data.

I feel that I have also managed to develop some of my personal development skills such as time management, task management and the ability to balance multiple commitments at the same time. I feel that it will be a valuable skill to be able to look at a large task and be able to split it up into smaller chunks. Likewise time management and being able to balance multiple commitments at the same time will always be viewed as a valuable asset in the working world.

Appendices

Appendices one

```
config:

    oauth:

        accessToken: removed

        tokenSecret: removed

        consumerKey: removed

        consumerSecret: removed

    twilio:

        sid: removed

        auth: removed

        myPhoneNumber: removed"
```

```
        twilioPhoneNumber: removed "
    locations:
        -0.3952,51.4101,0.1439,51.6069: London
        -2.4204,52.3022,-1.3424,52.6872: Birmingham
        -2.7727,53.2537,-1.6946,53.6304: Manchester
        -2.0840,53.6353,-1.0059,54.0086: Leeds-Bradford
        -3.1881,53.2989,-2.6491,53.4874: Liverpool-Birkenhead
```

Appendices 2

'b v a', "a's", 'able', 'about', 'above', 'according','accordingly',

'across', 'actually', 'after', 'afterwards','again', 'against',

"ain't", 'all', 'allow','allows', 'almost', 'alone', 'along', 'already','also',

'although', 'always', 'am', 'among','amongst', 'an', 'and', 'another', 'any','anybody',

'anyhow', 'anyone', 'anything', 'anyway','anyways', 'anywhere', 'apart', 'appear',

'appreciate','appropriate', 'are', "aren't", 'around', 'as','aside', 'ask', 'asking', 'associated',

'at','available', 'away', 'awfully', 'be', 'became','because', 'become', 'becomes', 'becoming',

'been','before', 'beforehand', 'behind', 'being', 'believe','below', 'beside', 'besides', 'best',

'better','between', 'beyond', 'both', 'brief', 'but','by', "c'mon", "c's", 'came', "can','can't",

'cannot', 'cant', 'cause', 'causes','certain', 'certainly', 'changes', 'clearly', 'co','com',

'come', 'comes', 'concerning', 'consequently','consider', 'considering', 'contain', 'containing',

'contains','corresponding', 'could', "couldn't", 'course', 'currently','definitely', 'described',

'despite', 'did', "didn't','different", 'do', 'does', "doesn't", "doing','don't", 'done', 'down',

'downwards', 'during','each', 'edu', 'eg', 'eight', 'either','else', 'elsewhere', 'enough',

'entirely', 'especially','et', 'etc', 'even', 'ever', 'every','everybody', 'everyone','everything',

'everywhere', 'ex','exactly', 'example', 'except', 'far', 'few','fifth', 'first', 'five', 'followed',

'following','follows', 'for', 'former', 'formerly', 'forth','four', 'from', 'further', 'furthermore',

'get','gets', 'getting', 'given', 'gives', 'go','goes', 'going', 'gone', 'got', 'gotten','greetings',

'had', "hadn't", 'happens', 'hardly','has', "hasn't", 'have', "haven't", 'having','he', "he's",

'hello', 'help', 'hence','her', 'here', "here's", 'hereafter', 'hereby','herein', 'hereupon', 'hers',

'herself', 'hi','him', 'himself', 'his', 'hither', 'hopefully','how', 'howbeit', 'however', "i'd",

"i'll','i'm", "i've", 'ie', 'if', 'ignored','immediate', 'in', 'inasmuch', 'inc', 'indeed','indicate',

'indicated', 'indicates', 'inner', 'insofar','instead', 'into', 'inward', 'is', "isn't','it", "it'd",

"it'll", "it's", 'its','itself', 'just', 'keep', 'keeps', 'kept','know', 'known', 'knows', 'last',

'lately', 'later', 'latter', 'latterly', 'least', 'less','lest', 'let', "let's", 'like', 'liked',

'likely', 'little', 'look', 'looking', 'looks','ltd', 'mainly', 'many', 'may', 'maybe','me', 'mean',

'meanwhile', 'merely', 'might','more', 'moreover', 'most', 'mostly', 'much','must', 'my', 'myself',

'name', 'namely','nd', 'near', 'nearly', 'necessary', 'need','needs', 'neither', 'never',

'nevertheless', 'new','next', 'nine', 'no', 'nobody', 'non','none', 'noone', 'nor', 'normally',

'not','nothing', 'novel', 'now', 'nowhere', 'obviously','of', 'off', 'often', 'oh', 'ok','okay',

'old', 'on', 'once', 'one','ones', 'only', 'onto', 'or', 'other','others', 'otherwise', 'ought', 'our',

'ours','ourselves', 'out', 'outside', 'over', 'overall','own', 'particular', 'particularly', 'per',

'perhaps','placed', 'please', 'plus', 'possible', 'presumably','probably', 'provides', 'que',

'quite', 'qv','rather', 'rd', 're', 'really', 'reasonably','regarding', 'regardless', 'regards',

'relatively', 'respectively','right', 'said', 'same', 'saw', 'say','saying', 'says', 'second',

'secondly', 'see','seeing', 'seem', 'seemed', 'seeming', 'seems','seen', 'self', 'selves',

'sensible', 'sent','serious', 'seriously', 'seven', 'several', 'shall','she', 'should', "shouldn't",

'since', 'six','so', 'some', 'somebody', 'somehow', 'someone','something', 'sometime', 'sometimes',

'somewhat', 'somewhere','soon', 'sorry', 'specified', 'specify', 'specifying','still', 'sub', 'such',

'sup', "sure','t's", 'take', 'taken', 'tell', 'tends','th', 'than', 'thank', 'thanks', 'thanx','that',

"that's", 'thats', 'the', 'their','theirs', 'them', 'themselves', 'then', 'thence','there',

"there's", 'thereafter', 'thereby', 'therefore','therein', 'theres', 'thereupon', 'these',

"they','they'd", "they'll", "they're", "they've", 'think','third', 'this', 'thorough', 'thoroughly',

'those','though', 'three', 'through', 'throughout', 'thru','thus', 'to', 'together', 'too',

'took','toward', 'towards', 'tried', 'tries', 'truly','try', 'trying', 'twice', 'two', 'un','under',

'unfortunately', 'unless', 'unlikely', 'until','unto', 'up', 'upon', 'us', 'use','used', 'useful',

'uses', 'using', 'usually','value', 'various', 'very', 'via', 'viz','vs', 'want', 'wants', 'was',

"wasn't','way", 'we', "we'd", "we'll", "we're','we've", 'welcome', 'well', 'went', "were','weren't",

'what', "what's", 'whatever', 'when','whence', 'whenever', 'where', "where's", 'whereafter','whereas',

'whereby', 'wherein', 'whereupon', 'wherever','whether', 'which', 'while', 'whither', "who','who's",

'whoever', 'whole', 'whom', 'whose','why', 'will', 'willing', 'wish', 'with','within', 'without',

"won't", 'wonder', "would','wouldn't", 'yes', 'yet', 'you', "you'd','you'll", "you're", "you've",

'your', 'yours','yourself', 'yourselves', 'zero'

References

1. http://www.cnet.com/uk/news/facebook-processes-more-than-500-tb-of-data-daily/ published 22/08/12 date last viewed 06/05/14
2. https://blog.twitter.com/2013/new-tweets-per-second-record-and-how published august 16 2013 date last viewed 06/05/14
3. https://www.pearanalytics.com/wp-content/uploads/2012/12/Twitter-Study-August-2009.pdf published august 2009 date last viewed 06/05/14
4. http://aclweb.org/anthology/P/P12/P12-1056.pdf published 2010 date last viewed 06/05/14
5. https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf published April 2004 date last viewed 06/05/14
6. https://wiki.python.org/moin/GlobalInterpreterLock published 06/05/14date last viewed 06/05/14
7. https://en.wikipedia.org/wiki/Bessel%27s_correction published 06/05/14 date last viewed 06/05/14
8. https://web.cse.msu.edu/~cse802/notes/ConstrainedKmeans.pdf published march 2001 date last viewed 06/05/14
9. http://speed.pypy.org/ published 06/05/14 date last viewed 06/05/14

Student: Adam Flax
Student Number: C1115629
Module: CM2303 One Semester Individual Project

Supervisor: Dr Steven Schockaert
Moderator: Dr David W Walker
Module Credits: 40