Chapter 1: Method Description

1.1 Method Introduction

The algorithm implemented in this program is an adapted and simplified version of the burst detection algorithm described in the event detection lecture. The tweets used for development and evaluation have already undergone the pre-processing and clustering stages, so the cluster output was used to create the method described here. The cluster file contains only the tweets with named entities (significant names such as ‘Obama’) that have at least one nearest neighbour.

The method starts by reading in all tweets and sorting them into lists of clusters and entities as well as creating a list of all tweets present. In each cluster and entity, several statistics are computed. For each cluster, the average of all the timestamps of the tweets within the cluster is calculated, this is referred to as the centroid time.

1.2 Three-Sigma Values

For each entity, ‘three-sigma’ values are created for each of the windows used to detect events, these windows differ in size and are 5,10,20,40,80,160 and 360 minutes long. The timespan of all the tweets entered into the algorithm is divided up by these windows and the frequency of tweets in each division is recorded for each entity i.e. a day's worth of tweets with a 60 minute long window creates 24 frequencies for each entity. These frequencies are used to find the mean and standard deviation of the window (for each entity) and the three-sigma value is set to be mean + 3*sd.

The Three-Sigma value is chosen by the application of the three sigma rule which states that a value is practically impossible if it is more than 3 standard deviations away from the mean. For an entity to ‘burst’ i.e. to be tweeted about to such a degree that an event related to the entity may be happening, the number of tweets with a certain entity within a given window must be greater than the three-sigma value for that entity/window pair.

1.3 Burst Detection

After sigma-values are computed, the algorithm runs through all of the tweets and for each tweet counts the number of tweets within each window size, starting with the largest, that match the entity of the first tweet. If the number of matching tweets is larger or equal to the three-sigma value for that entity-window pair then the tweets undergo further checks to determine if they should be outputted. If not, the window size is decreased or the next tweet is moved onto if the window size is 5 mins. The windows are cycled in descending size order as any tweets referenced in a larger sized window will already be included within a smaller window. In the

event of a burst in the larger window the tweets in the smaller window have already been included in the further checks making checking the smaller window pointless..

In the event of a burst, the tweets within the window are sorted into clusters for the window and these are checked to determine if they should be outputted. First of all, the number of tweets present in each window cluster must be greater than or equal to 10, this stops any small but noisy clusters from being included in the output. (10 is chosen as a suggestion from the Event Detection Lecture). Also, the centroid time of a cluster (for tweets inside and outside of the window), must be greater than the timestamp of the tweet that signaled the burst. This is done to make sure that tweets that are associated with background topics (and therefore not any event) are not included within the burst output. Tweets that pass these criteria are outputted as tweets associated with events.

1.4 Pseudo-Code for Method

```
algorithm TweetSelection is
    input: csv file, csv, containing clusters sorted by tweet time
    output: csv file, output, containing all tweets associated with events

    for each line in csv do
        create Tweet t for line
        sort t into Entity e
        sort t into Cluster c

    for each cluster in ClusterMap cm
        calculate average time for Tweets in cluster

    for each entity in EntityMap em
        for each window win
            calculate sigma value

    for each Tweet in TweetList tl
        for each window win
            count number of tweets numTweets equal Tweet.entity
            if numTweets greater or equal entity.window.sigma
                add window to WindowList wl
                goto next Tweet

    for each window in wl
        organise Tweets in window into cluster cl
        if cl.num greater than 10 and cl.averageTime greater than window.firstTweet
            add cl to output

    return output
```

Chapter 2: Algorithm Description

The method described in Chapter 1 has been implemented using Java and uses a main class (TweetSelection.java) that imports from four created classes; Tweet, Window, Entity and Cluster. These will described briefly as they all called upon within the algorithm. The code for the algorithm is supplied with this report which has been commented appropriately and can be read along with the algorithm description to get a clear picture of how it works.

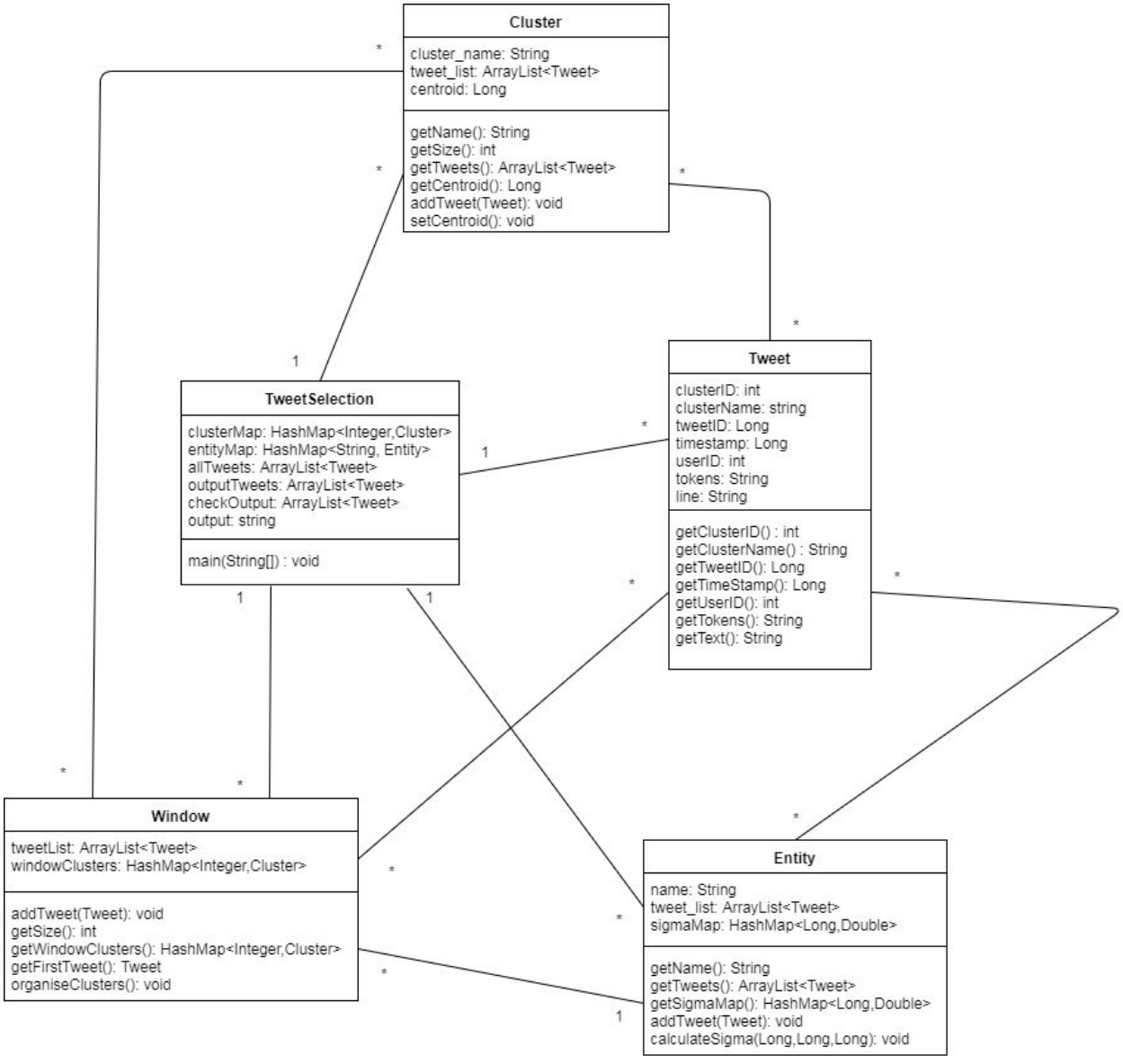

Figure 2.7: UML Diagram for the class structure of the algorithm

2.1 Tweet Sorting

The algorithm requires a csv file that contains a list of clusters with one tweet per line sorted by the time of the tweets (oldest to newest). The algorithm starts by trying to open the csv file and reads it line by line until all lines have been processed. When a line is processed a new Tweet object is created and added to an ArrayList that will contain all of the Tweets within the file. A Tweet object contains all the attributes that are contained within a line of the csv file as well as the full line represented as a string.

Next, the tweet is sorted into a cluster, if the HashMap that holds all the clusters does not contain the current cluster, uniquely identified by the number ID of the cluster, then a new cluster is created and added to the HashMap. A cluster contains the name of the cluster, an ArrayList containing all the tweets within the cluster and a Long (representing the centroid time of the cluster) which is initially set to 0. The appropriate Cluster is then retrieved from the HashMap and the Tweet is added before the Cluster is re-added back into the HashMap.

After this, the same procedure as before is done for Entity objects. An Entity also contains an ArrayList of Tweets and its name but unlike a Cluster is uniquely identified by its name and not a number ID (as an entity may be made up by many clusters). Instead of the centroid time, an Entity contains a HashMap of window sizes and three-sigma values. After Entity sorting, the first section of the program can be considered as complete.

```
System.out.println("Creating Entity and Cluster Objects");

//Tries to read in the file specified in the command line by line
try {
    br = new BufferedReader(new FileReader(args[0]));
    while ((line = br.readLine()) != null) {

        //Splits the line by commas and creates a new Tweet object that is inserted into the allTweets list
        String[] row = line.split(csvSplitBy);
        Tweet tweetInsert = new Tweet(Integer.parseInt(row[0]),row[1],Long.parseLong(row[2]),Long.parseLong(row[3]),Integer.parseInt(row[4]),row[5],line);

        //Checks if the current Tweet is newer or older than the current max/min timestamp, then adds tweet to the ArrayList of all Tweets
        if(tweetInsert.getTimestamp() < minTime) {
            minTime = tweetInsert.getTimestamp();
        }

        if(tweetInsert.getTimestamp() > maxTime) {
            maxTime = tweetInsert.getTimestamp();
        }

        allTweets.add(tweetInsert);


        //Adds the tweet to a cluster and creates the cluster if it does not already exist
        //The cluster is then added to the Map of Clusters when it is updated/created
        if(clusterMap.containsKey(tweetInsert.getClusterID()) == false) {
            clusterMap.put(tweetInsert.getClusterID(),new Cluster(tweetInsert.getClusterName()));
        }
        Cluster currentCluster = clusterMap.get(tweetInsert.getClusterID());
        currentCluster.add_tweet(tweetInsert);
        clusterMap.put(tweetInsert.getClusterID(), currentCluster);

        //Does similar to before but adds tweets to Entity Objects instead of Cluster Objects
        if(entityMap.containsKey(tweetInsert.getClusterName()) == false) {
            entityMap.put(tweetInsert.getClusterName(), new Entity(tweetInsert.getClusterName()));

        }
        Entity currentEntity = entityMap.get(tweetInsert.getClusterName());
        currentEntity.addTweet(tweetInsert);
        entityMap.put(tweetInsert.getClusterName(), currentEntity);

    }
```

Figure 2.1: Code for Tweet Sorting

2.2 Statistic Creation

The second section of the algorithm entails creating statistics for entities and clusters, these statistics are then used in the tweet selection process. The first part of statistic creation is setting the centroid time for each Cluster by iterating through the HashMap of Clusters and calling the function setCentroid(). This function averages the timestamps of all the tweets within a cluster and sets this value to be its centroid time. The Cluster is then re-entered into the HashMap with this new centroid value.

The second part of statistic creation involves calculating three-sigma values for each window inside each entity. The function calculateSigma() does this by cycling through timespan of all tweets and producing an ArrayList of window sized divisions which hold the frequency of tweets. Using this ArrayList, the mean number of tweets per window size and the standard deviation of tweets (link sd formula) are calculated. The three-sigma value is calculated as mean + 3*sd and for each entity the 7 three-sigma values are created and entered into the Entity's Sigma HashMap.

```
System.out.println("Entity and Cluster Objects created");
System.out.println("Cluster Centroid Calculation");

//Runs through each cluster and calculates the centroid time for all the tweets in that cluster
for(int i = 1; i < clusterMap.size(); i++) {
    Cluster currentCluster = clusterMap.get(i);
    currentCluster.setCentroid();
}

System.out.println("Centroids Calculated");
System.out.println("Calculate Sigmas");

//Cycles through the Entitys and calculates the 3-sigma values for each window size
//5,10,20,40,80,160,360 minutes and re-enters the Entity into the Entity Map
for(String key : entityMap.keySet()) {
    Entity currentEntity = entityMap.get(key);
    currentEntity.calculateSigma(minTime, maxTime, 300000L);
    currentEntity.calculateSigma(minTime, maxTime, 600000L);
    currentEntity.calculateSigma(minTime, maxTime, 1200000L);
    currentEntity.calculateSigma(minTime, maxTime, 2400000L);
    currentEntity.calculateSigma(minTime, maxTime, 4800000L);
    currentEntity.calculateSigma(minTime, maxTime, 9600000L);
    currentEntity.calculateSigma(minTime, maxTime, 21600000L);
    entityMap.put(key, currentEntity);
}
```

Figure 2.2: Code for Statistic Creation

```
//Calculates Sigmas for a specified Window Size, starts by creating a endPoint for a window
//and a list of counts for a set of windows
public void calculateSigma(Long minTime, Long maxTime,Long timeStep) {
    Long stepEnd = minTime + timeStep;
    ArrayList<Integer> stepAmounts = new ArrayList<Integer>();

    //Cycles through the timespan of allTweets and counts the number of tweets in each individual window
    while(minTime < maxTime) {
        int count = 0;
        for(int i = 0; i < this.tweet_list.size(); i++) {
            if(this.tweet_list.get(i).getTimestamp() > minTime) {
                if(this.tweet_list.get(i).getTimestamp() < stepEnd) {
                    count = count + 1;
                }
            }
        }

        //Adds the count to the List and updates the window limits
        stepAmounts.add(count);
        minTime = stepEnd;
        stepEnd = stepEnd + timeStep;
    }

    //Calculates the mean number of tweets per window size
    double sum = 0.0;
    for(int j = 0; j < stepAmounts.size(); j++) {
        sum = sum + stepAmounts.get(j);
    }
    double mean = sum/stepAmounts.size();

    //Calculates the standard deviation of the counts
    double sumSq = 0.0;
    for(int k = 0; k < stepAmounts.size(); k ++) {
        sumSq = Math.pow((stepAmounts.get(k) - mean),2);
    }
    double sd = Math.sqrt(sumSq/(stepAmounts.size() - 1));

    //Calculates mean + 3*sd and rounds it up to the nearest integer (can't have a decimal of a tweet)
    //then adds this sigma to the sigmaMap
    double sigma = Math.ceil(mean + (3*sd));
    this.sigmaMap.put(timeStep, sigma);
`
```

Figure 2.3: Code for the method calculateSigma()

2.3 Burst Detection

The next section of the algorithm is burst detection, this part selects tweets that are determined to be ‘eventful’. A tweet is removed from the allTweets ArrayList and for each window size, the number of Tweets within the window whose Entity matches the removed Tweet’s Entity are counted. This is done via creating a Window object which contains a list of tweets within the window and a HashMap of the clusters.

The number of tweets in the window is checked against the three-sigma value for the window-entity pair and if larger or equal then the Window is added to an ArrayList of bursting windows. The window must also have at least 10 tweets as the minimum number of tweets in a window cluster is 10 and the minimum number of clusters within a window is 1. The algorithm then moves to the next tweet in the allTweets ArrayList.

```
//that starts at the time of the tweet
while(allTweets.size() != 0) {

    //Starting with the largest window size, the tweet is removed from the allTweets
    //(size of allTweets is adjusted), the Sigmas for the entity are retrieved and
    //bursting is set to false.
    long step = 21600000L;
    Tweet currentTweet = allTweets.remove(0); //Removal is done so that only tweets after the current tweet are included in the window
    allTweets.trimToSize();
    HashMap<Long, Double> entitySigmas = entityMap.get(currentTweet.getClusterName()).getSigmaMap();
    boolean burst = false;

    //Iterates through the windows sizes, reducing each time unless the window has been
    //indicated as bursting. A List of possible tweets that may be outputted is created, first tweet is
    //automatically added
    while(step >= 300000L && burst == false) {
        ArrayList<Tweet> possTweets = new ArrayList<Tweet>();
        possTweets.add(currentTweet);
        Long endPoint = currentTweet.getTimestamp() + step;

        //If the Window End Point is greater than the maximum Tweet Time then the end point is
        //set to the maximum Tweet Time
        if(endPoint > maxTime) {
            endPoint = maxTime;
        }

        //A new window is created and the current tweet is added to it
        int j = 0;
        Window newWindow = new Window();
        newWindow.addTweet(currentTweet);

        //Cycles through all the tweets upto the end point of the window and adds the tweet to the
        //window if the entities of the removed tweet and the current tweet in the cycle are equal
        while(j < allTweets.size() && allTweets.get(j).getTimestamp() <= endPoint) {
            if(allTweets.get(j).getClusterName().equals(currentTweet.getClusterName())) {
                newWindow.addTweet(allTweets.get(j));
            }
            j = j + 1;
        }

        //If the number of tweets in the window is greater than 10 and greater than the
        //three-sigma statistic for that entity and window then the window is added to the
        //outputWindows structure. Burst is changed to true so that the algorithm doesn't check
        //smaller windows where the tweets have already been added to the output
        if(newWindow.getSize() >= 10 && newWindow.getSize() >= entitySigmas.get(step)) {
            outputWindows.add(newWindow);
            burst = true;
        }

        //Changes the step value if no burst has been detected
        if(step == 21600000L) {
            step = 9600000L;
        } else {
            step = step/2;
        }
    }
}
```

Figure 2.4: Code for Burst Detection

Following on from this, each window in the outputWindows ArrayList is retrieved and has its tweets organised into clusters. If a cluster matches the size and centroid time requirements as described in Chapter 1 then all the tweets inside the cluster are added to an ArrayList of tweets which are being considered for output.

```
//Runs through all the windows that have been indicated as bursting and may be output
for(int i = 0; i < outputWindows.size(); i++) {
    //Organises the clusters within each window
    outputWindows.get(i).organiseClusters();
    HashMap<Integer,Cluster> windowClusters = outputWindows.get(i).getWindowClusters();

    //Runs through each cluster within a window
    for(Integer clusterKey : windowClusters.keySet()) {
        Cluster currentCluster = windowClusters.get(clusterKey);
        //If the size of the cluster in greater than 10 and the centroid of the same overall
        //cluster is greater than the bursting time then all the tweets in the cluster
        //are considered for output
        if(currentCluster.getSize() > 10) {
            Cluster masterCluster = clusterMap.get(clusterKey);
            if(masterCluster.getCentroid() > outputWindows.get(i).getFirstTweet().getTimestamp()) {
                outputTweets.addAll(currentCluster.getTweets());
            }
        }
    }
}
```

Figure 2.5: Code for Window Sorting

2.4 Output

The algorithm finally reads through the ArrayList of Tweets selected for output and checks that each Tweet is not present in the ArrayList of Tweets checkOutput which holds those that have already been added to the output string. If the Tweet is not already in checkOutput, it is added, and the line associated with the Tweet is added to the output string. Finally, the algorithm tries to write the output string to a file Output.csv and on success this file can be used for evaluation.

```
for(int i = 0; i < outputTweets.size(); i++) {
    if(checkOutput.contains(outputTweets.get(i)) == false) {
        output = output + outputTweets.get(i).getText() + '\n';
        checkOutput.add(outputTweets.get(i));
    }
}

System.out.println("Output Generated");

//The program finally tries to write the output string to a file called Output.csv
try {
    fw = new FileWriter("Output.csv");
    bw = new BufferedWriter(fw);

    bw.write(output);
} catch(IOException e) {
    e.printStackTrace();
} finally {
    try {
        if(bw != null) {
            bw.close();
        }

        if(fw != null) {
            fw.close();
        }
    } catch(IOException ex) {
        ex.printStackTrace();
    }
}
```

Figure 2.6: Code for the Output of Tweets

Chapter 3: Evaluation

To evaluate the algorithm, the provided program eval.py was used which produces statistics that can describe the effectiveness of the event detection algorithm. During development, the csv file used as an input featured only one day's worth of tweets. For the final evaluation statistics reported below one week's worth of tweets was used and this is evaluated against one month's worth of 'ground truth' data to create statistics that can describe the effectiveness of the algorithm.

3.1 Description of Statistics

The main three statistics used to evaluate the method are :

- **Event Recall**, this statistic is calculated by taking the number of tweets that are both relevant and retrieved and dividing these by the total number of relevant tweets, taken from the groundtruth. The higher this statistic, the better the algorithm has performed at retrieving tweets that are associated with an event where 1 indicates that every tweet retrieved is present in the groundtruth and 0 indicates that no tweets could be associated.
- **Cluster Precision**, this statistic is calculated by taking the number of tweets that are both relevant and retrieved and dividing these by the total number of retrieved tweets. The higher this statistic the smaller the number of tweets that are considered as 'noise' are present in the algorithm's output.
- **Overall F-Measure**, this statistic is a combination of the previous two and provides a balanced measurement of the effectiveness of the algorithm. This is calculated via the formula 1/(alpha(1/Precision)+(1-alpha)(1/Recall)) where in an ideal situation, alpha is 0.5.

As well as these statistics, eval.py also outputs a cluster by cluster summary which indicates what percentage of tweets within the cluster are present in the groundtruth, this has not been displayed in the report as it is too large to report. Eval.py also produces a category breakdown of the events detected which can be used to determine the usefulness of the algorithm across different types of events but again these are not quoted as they only serve to further breakdown the Event Recall statistic.

3.2 Results of Evaluation

Running the algorithm on 7 days worth of tweets and checking the output given against a groundtruth containing one months worth of events and tweets associated with these events gives the following output:

EVENTS / CLUSTER STATISTICS:
- Of 506 events, 94 were detected.
- Of 1266 clusters, 435 could be matched back to an event.

Event Recall: 0.186
Cluster Precision: 0.344
Overall F-Measure: 0.241

3.3 Results Discussion

The results show that the algorithm returns roughly a 35% success rate in determining which clusters are useful over those that are considered to be noise. While this gives the impression that the method is not very precise it is an improvement on the cluster precision given by the raw file of 0.014 which indicates that some useful selection has taken place.

The algorithm also slightly reduces the value of Event Recall from 0.209 for the raw file to 0.186 in the post-algorithm file. This is due to clusters for 12 events being removed by the algorithm, this can happen due to those clusters having less than 10 tweets or the event being associated with a popular or frequent entity which has a high three-sigma value therefore causing the events' tweets to be considered as noise.

The F-Measure also shows a major improvement post-algorithm moving from 0.026 to 0.241. This improvement is also reflected in the alpha values which the raw file calculated to be 0.505 and the post-algorithm file calculated to be 0.497. The post-algorithm alpha value is closer to 0.5 indicating that the algorithm did have a positive effect on the selection of tweets.

This is furthermore supported by the number of clusters returned by the algorithm as eventful, evaluating the raw file gives 50631 clusters with 696 being matched back to an event. The algorithm reduces the number of returned clusters by 97% with only 38% of useful clusters being rejected. While, with more development, this value can be reduced it shows that the algorithm has performed well at removing clusters that are not associated with events.

Chapter 4: Critical Discussion

4.1 Method Discussion

While this method effectively reduced the number of 'noisy' clusters within the dataset, it did so at a poor speed. For a list of tweets taken from one day, the algorithm took around 20 seconds to complete. When the amount of tweets is increased to 7 days, the time increased to up to 5 minutes. This means that trying to process any datasets with a large number of tweets or with a complex distribution of clusters may result in an extended execution time which is undesirable.

The part of the algorithm that takes the most time is the generation of an output, roughly 75% of the algorithm's execution time is generating output. Due to the storage of the tweet objects, the algorithm has to iterate through all the objects in an ArrayList to generate an output, this is costly. Due to time constraints, a better output method could not be implemented, a slow but working solution was prioritised over a fast but ineffective one. If the output of tweets to a csv file is not required then the algorithm would run faster and therefore be used as part of a wider method to detect events.

One application of this method is as a filter for the returned clusters following the clustering procedure of tweets. The algorithm could remove the majority of noisy clusters with the output being passed onto humans further selection using crowdsourcing. This would decrease the cost of crowdsourcing as analysing around

1200 clusters requires less time and effort then analyzation of over 50000. Crowdsourcing could also be used to remove clusters that are noisy but not detected as such by the algorithm. For Example; the algorithm will, in most situations, return tweets with the entity Facebook which reference users tweeting about uploaded photos. This is hard to automatically detect as non-eventful because Facebook is too broad of an entity (the tweet may reference a news story about Facebook and photos which could be considered as an event). Via the application of crowdsourcing, humans may be able to determine that the tweets returned by the algorithm do not reference events and hence remove to further improve the eventful tweet collection.

4.2 Event Detection Discussion

Event Detection has many applications within the wider world. One such application could see a sports distributor such as Sky Sports using event detection to determine which matches in the Premier League interest fans the most. By using event detection, Sky Sports may be able to determine which matches have the most fanfare surrounding them and therefore possibly attracting a higher number of viewing figures. Television and Film companies could also use event detection to gain an early understanding of how popular a program or film is with audiences and also possibly determine if there are certain dates when releasing a film would not be viable/recommended due to other, wider events garnering more attention.

Event Detection could also be used to determine the impressions and opinions of users for product or service. While feedback can be gathered using hashtags and searching for key terms, a company could use event detection to determine if there were/are times when the tweets referencing a product or service are overwhelmingly positive or negative. This could be useful in modifying a service so that it is improved during the times when the majority of negative tweets are being posted.

One possible advancement in twitter's technology that may modify the usefulness of Event Detection is the recent introduction of letting users tweet up to 280 characters instead of the previous 140. While the increase in characters allows for more information to be gathered about possible events, the downside is that this increase in information may result in several stages of the Event Detection process becoming increasingly complex and therefore more costly to implement and run. A certain tweet may double in the number of entities/clusters that it is associated with and therefore running an algorithm such as above may become extremely time consuming for even one day's volume of tweets.