Andrei-Mihai Nicolae

Web Science (M)

20 November 2017

Web Science (M) AX Report

1. Introduction

Event detection on Twitter streams is a challenging task as it is influenced by many “noisy” factors:

- huge amount of tweets;
- API is not very performant and has its limitations;
- most tweets are junk.

However, there are multiple researches that try to find solutions to these issues, such as the paper of McMinn et al. In the following sections, the approach proposed by this particular research shall be presented.

2. Approach

There are multiple steps involved in processing the data and trying to detect events:

- **data collection**: this has been already performed;
- **pre-processing**: tweets were already parsed, tagged and filtered;
- **clustering**: the tweets were assigned to clusters using weight cosine similarity scores;

- **burst detection**: check for bursts in a given window of time (minutes/hours);
- **cluster identification**: clusters are already mapped to events;
- **event merging**: merge events with linked named entities (i.e. containing same or similar/linked named entities).

In this report we shall present the process of cluster filtering and event merging in order to provide better, optimised event detection.

Firstly, the cluster filtering was rather a simple step: the whole dataset was parsed and the clusters having less tweets than a specific threshold (e.g. 10 tweets per cluster) were dropped. Then, in the final CSV which was written, only the non-dropped clusters would be included. The following pseudocode describes, at a high level, how this method was implemented:

```
def filterClusters (tweetThreshold):
    forEach cluster in clusters:
        if cluster.getNumberOfTweets < tweetThreshold:
            clusters.drop(cluster)
    return clusters
```

Even though the above process was used as a standalone technique as well (i.e. run evaluation measurements simply on filtered clusters), it was also used in conjunction with the event merging technique. In order to implement the latter, first step was to loop through all clusters and determine the centroid times (i.e. average of all tweets' timestamps). Then, these centroid times would be used in order to calculate whether the tweets were in a "window interval", which is an interval of time where a burst was detected. After this was performed, in the last step we apply the filtering technique to remove merged clusters that had less tweets than a specified threshold. This whole process can be described with the following pseudocode:

```
        def filterClusters (tweetThreshold):
            forEach cluster in clusters:
                if cluster.getNumberOfTweets < tweetThreshold:
                    clusters.drop(cluster)
            return clusters

        def computeCentroidTime (cluster):
            forEach tweet in cluster:
                sum += tweet.getTimestamp
            return sum/cluster.size()

        def mergeClusters (clusters, windowInterval):
            forEach cluster in clusters:
                forEach clusterTwo in clusters:
                    if cluster != clusterTwo and
                      (cluster.namedEntity == clusterTwo.namedEntity or
                       cluster.namedEntity.similarTo(clusterTwo.namedEntity)) and
                       cluster.computeCentroidTime() - clusterTwo.computeCentroidTime() >
windowInterval:
                    cluster.addAllTweets(clusterTwo)
                    clusters.remove(clusterTwo)
            return filterClusters(clusters)
```

3. Code Description & UML

I have written the whole code in Go as it is a very efficient, well designed and easy to understand programming language. I have modularised the whole application and commented it properly, explaining what each function does. The code is split into three:

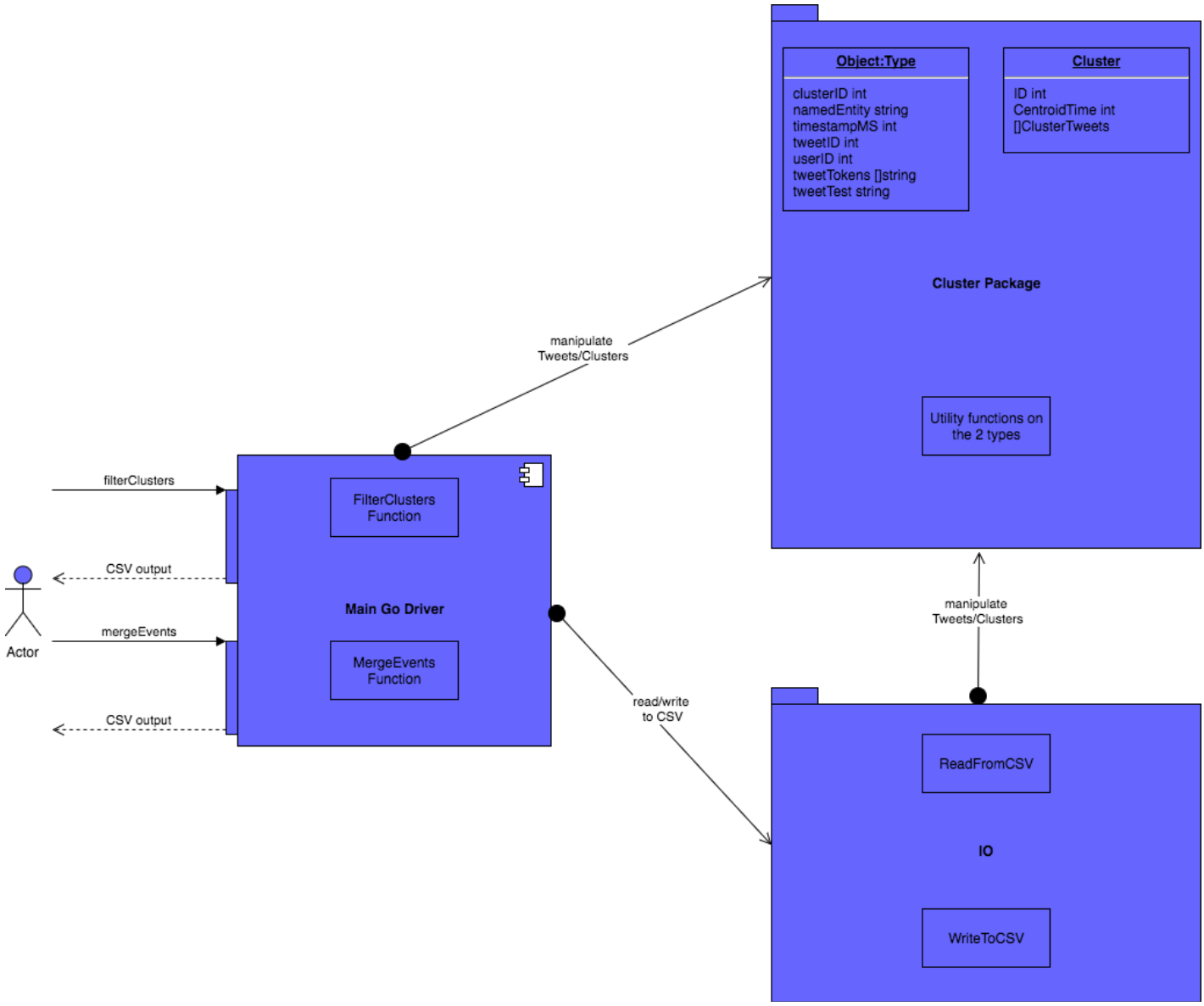
- io package - contains functions to read and write to a CSV file;
- cluster package - contains type definitions for Cluster and Tweet structs;
- main Go file - used to run the application (using whichever method the user wants).

The io package is relatively straightforward - it contains a single csv file that has 2 exported functions (ReadFromCSV and WriteToCSV) and two utility functions used to convert fields to various scalar types in Go.

The cluster package contains type definitions for Cluster and Tweet type. These are used for easier manipulation of CSV data coming/going out of the application. It is also used, for example, to create clusters from a list of tweets, convert a cluster map to a slice and other utility functions.

The main driver, main Go file, reads a specified CSV file (I used the clusters.sortedby.clusterid CSV, thus if you'd like to test it, please the CSV files sorted by clusterID) and then uses one of the two techniques described above (i.e. filter clusters or merge events based on burst detection).

The following figure describes the simple architecture of the application:



4. Evaluation

The application was then ran on various inputs on the both datasets (i.e. 1 day and 7 days). We first need to define the variables we are interested in measuring:

- recall - fraction of relevant clusters from the returned set;
- precision - fraction of clusters that are relevant from the entire set of events;
- f-measure - weighted harmonic mean of precision and recall.

The tests were run on a variety of inputs (5 - 50 tweets threshold) and the results were surprising. The findings will be divided into the 2 methods applied:

- simple filtering - during evaluation of this technique it was concluded that while the threshold increased, the precision measure has increased, slowly though, as well as recall. Moreover, the f-measure increased as well along with the threshold, but started to decrease after threshold went over 15.
- merging events - the f-measure increased especially in the 200 - 550 minutes window interval, recall increased as well as precision while the threshold was increased.

5. Discussion

When using the filtering technique, the results were well increased above the baseline. This was a rather simple method to implement, but it brought good improvements to our event detection algorithm without any impact on the recall values. Moreover, precision and f-measure increased as well by high rates, but in the end one need to pay attention on what value the data will be sorted by.

On the other hand, surprisingly enough, using the event merging approach did not bring much improvement, even though the recall increased compared to its baseline value.

6. Traffic Detection

Traffic detection is a very interesting problem. I do believe strongly that some particular instances of traffic congestion/accidents detection via social media streams, such as Twitter, could help drivers avoid affected areas. However, this imposes several issues:

- when an accident happens, there are various types of companies that manage the roads - in the Western world, it might be fixed very quickly, while in other countries the fix could take much longer, thus the window interval where to detect bursts in Twitter streams needs to be adjusted properly;
- not all drivers will tweet about a traffic congestion as this is not a typical Twitter post category, thus the number of tweets related to it might be decreased;
- if, however, we have a reasonable number of drivers (and, of course, authorities or road construction/maintenance companies Twitter accounts) tweeting about it, we need to include in our search terms related to anger (e.g. swear words together with key words like traffic, congestion, accident) as the drivers wouldn't be happy to be late or similar situations.

Thus, in order to adapt to these conditions, we can start by defining a rather small burst detection window interval and then start analysing the tweets coming in. Then, we need to apply filters for anger/swear words used in conjunction with traffic keywords. Also, we need to monitor for radio stations/tv channels Twitter accounts that might likely report such events. Finally, we can use geo-tagging, if available on tweets, to correlate the affected area with targeted Twitter users and then monitor people twitting from those specific areas and check if there is indeed a traffic problem. We can apply the merging technique described above on tweets that have similar/linked named entities (e.g. accident, congestion) in correlation with available geo-tagging and then

check for properly sized windows (i.e. depending on the gravity of the situation, the affected area - maybe it is a busy street or a less visited area).

The following lines describe a high-level pseudocode for the above-mentioned approach:

```
def detectTrafficDetection (data):
    pre-process(data) // remove any noise and waste from our stream
    tag(parse(data)) // tag and parse data
    filter(data) // remove all tweets with no named entity
    cluster(data) // create inverted index for all named entities and compute
                  //  compute clusters based on similarity threshold
    burstDetection(data, 10min) // create a small window interval for
                                //  detecting bursts
    clusterSelection(data) // see how entity evolves and infer if incident
                           // (e.g. congestion) just started or has been fixed
    eventMerging(data) // merge events that relate to linked named entities;
                       // see if anger-related named entities/geo-tagging/
                       // key-terms (e.g. traffic accident, congestion) could be
                       // merged into the same event
    return clusters
```

Unfortunately, the geo-tagging functionality will most probably be missing as there is a very small number of tweets which are geo-tagged. Moreover, there might be only anger-related words in the tweet that will not contain anything related to traffic congestion, thus we might miss these tweets (however, we might also look for terms such as “going to be late” or “missing my meeting” or anything like that). Moreover, depending on the area affected, people might not even have internet access, which would also decrease the number of tweets related to the incident.

I believe traffic detection is quite challenging as the window will probably be small due to usually fast resolution to the congestion/accident and we need to process a rather limited number of tweets for most incidents (except those where, as mentioned above, they affect very frequented areas).