# LSH based New-Event-Detection on twitter

# (python version)

# Introduction

This is an implementation of New-Event-Detection on twitter microblogging system in python. The algorithm we implement we presented in:

Petrović, S., Osborne, M., & Lavrenko, V. (2010). Streaming first story detection with application to twitter. NAACL HLT 2010 - Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Proceedings of the Main Conference, (June), 181–189.

# Main Modules

In this section I present the main modules in this implementation.

TwitterParser

OR

Text Reader

DB Reader

Streamer

Properties

Main

Clustering

LSH-Cosine Module

LSH Hashtable

Parameters:  
# hashtables

Parameters:  
# documents

mongoDB

Parameters:  
# hyper planes  
# max bucket size

Parameters:  
\* Clustering threshold

json

## Main

## Streamer Framework

### TweetListener

### DB Streamer

### Text File Streamer

## NED Main Module

## LSH Module

### LSH-Cosine Similarity Module

### LSH Hashtable

## Performance Test

I have turned off all logs and then ran the process with different sets of tweets measuring the time every time. The graph plotted shows that the performance is linear to the number of tweets every time.

Figure 1: response time (in minutes) per number of tweets.

The test was run on 64 Bit Windows 7 PC with Intel i5 - 2.6 GHz and 8 GB Memory

In a test of memory consumption it was found that memory consumption is linear to the number of tweets used. Although, we started to face memory issues when running on one million tweets.

The memory performance test is plotted and presented in the next figure.

Figure 2: memory usage (in MB) per number of tweets.

The test was run on 64 Bit Windows 7 PC with Intel i5 - 2.6 GHz and 8 GB Memory

# System Parameters

The following are set of parameters needed in this implementation which can impact the resulting threads. Each parameter is explained and I put here the value that was used in our run.

|  |  |  |
| --- | --- | --- |
| Parameter | Description | Selected Value |
| N features | ???? | ???? |
| k | Number of hyper-planes | Recommended 13.  We used 20 |
| L | Number of hash tables.  “Given k, we can use equation (2) to compute L. In our case, we chose k to be 13, and L such that the probability of missing a neighbor within the distance of 0.2 is less than 2.5%. The distance of 0.2 was chosen as a reasonable estimate of the threshold when two documents are very similar. In general, this distance will depend on the application, and Datar et al. (2004) suggest guessing the value and then doing a binary search to set it more accurately. We set k to 13 it achieved a reasonable balance between time spent computing the distances and the time spent computing the hash functions” |  |
| bucket max size | bucket max size  The last three columns in Table 1 show the effect that limiting the bucket size has on performance. Bucket size was limited in terms of the percent of expected number of collisions, i.e., a bucket size of 0.5 means that the number of documents in a bucket cannot be more than 50% of the expected number of collisions. The expected number of collisions can be computed as n/2^k, where n is the total number of documents, and k is the LSH parameter explained earlier. | ???  50% \* 100,000 / 2^k = 0.5 \* 100,000 / 2^20 = 0.5 \* 0.095 = 0.047  ?? |
| ? | probability of missing a neighbor within the distance of 0.2 is less than 2.5%. | ? |
| max tweets | A window of 100,000 tweets | 100,000 |
| Entropy | entropy (< 3.5) to the back of the list, while we order other threads by the number of unique users.  A sign test showed this approach to be significantly better (p ≤ 0.01) than all of the previous ranking methods. Table 3 shows the effect of varying the entropy threshold at which threads are moved to the back of the list. We can see that adding information about entropy improves results regardless of the threshold we choose | < 3.5 |
| ? | fixed number of most recent documents. We set this number to 2000; preliminary experiments showed that values between 1000 and 3000 all yield very similar results. | ? |
| t | links relation: tweet a links to tweet b if b is the nearest neighbor of a and 1 −cos(a, b) < t for each tweet a we either assign it to an existing thread if its nearest neighbor is within distance t, or say that a is the first tweet in a new thread. If we assign a to an existing thread, we assign it to the same thread to which its nearest neighbor belongs.  threshold for tweets closeness If t is set very high, we will have few very big and broad threads, whereas setting t very low will result in many very specific and very small threads.  In our experiments, we set t = 0.5. We experimented with different values of t and found that for t ∈ [0.5,0.6] results are very much the same | 0.5 |
| Max threads | Maximum number of threads to print at the end | 1,000 |

# Missing parts

* missing "we order the elements of S according to the number of hash tables where the collision occurred. We take the top 3L elements of that ordered set and compare the new document only to them"
* missing fastest growing? How it is being calculated? Quoting: “Rate of growth of a thread is measured by the number of tweets that belong to that thread in a window of 100,000 tweets, starting from the beginning of the thread.”
* Sliding window: the paper does not mention sliding window mechanism . In our implementation we run the algorithm on x tweets but we jump each time with x/2. In order case the max tweets is set to 1,000. So we take 1,000 for processing but in the next round we skip 500 tweets in our database. This shall cause a “sliding window” effect.

# Examples of output

`