Filtering real-time trending words within text stream using Apache Storm

By Fudan University Data Science Research Group

Kailun Wang, Deyu Wu, Lu Xu, Ziheng Jiang

Introduction

This project instantiates the algorithm originally proposed in *Parameter free bursty events detection in text streams.* (FUNG, Gabriel Pui Cheong, et al, 2005).

A text stream, in this case, is a set of time-serialized articles. Trending words within the text stream further form up. A trending topic at a certain time period is a combination of words.

Not all words with a high frequency is considered a trendy word. Stopwords and daily expressions are certainly not in this case. Trend, in the paper, is defined by the probability "burst" within a certain time period. Consider the probability distribution of a word against all articles, the more the probability *q* of a certain time leans to the right side of x-axis, the more "trendy" the word is in the current time window.

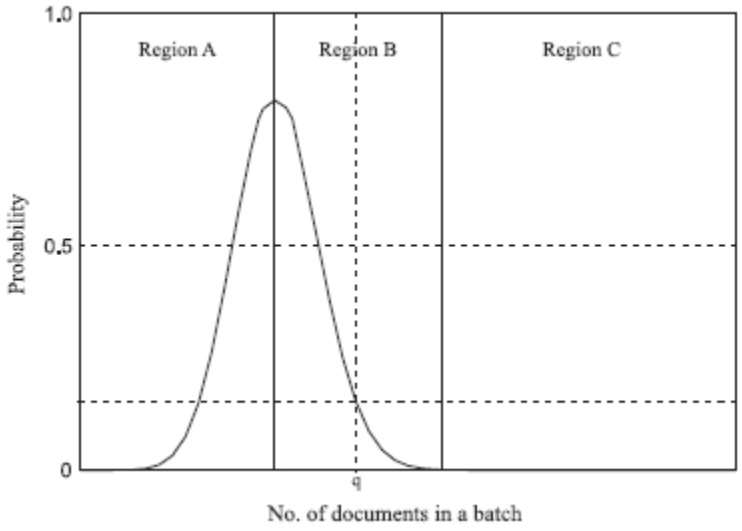

Figure 1 Region B and C are trends

Data selection

In the project we picked a decentralized SNS network mastodon, which provides both the handy “streaming API” as well as a rather large feed of instant user-generated data. The advantage of the dataset is that the topics in the network changes rapidly over time. The disadvantage is that too much slangs and daily languages are used. Also, users have their specific preferences over topics, and therefore the dataset may somehow lack generosity.

Problem transformation

Apache Storm does not have a storage system built-in, and data lives in the form of a flow. However, the original paper works on a static dataset. The trade-off point is the length of a single time slot. The proposed algorithm put out a brute force attempt to iterate through all possible combinations of trendy words, without specifying any particular time. Therefore, in our real-time case we cannot handle with an exponential complexity. Also, the majority of SNS posts are of single sentences, and it is therefore quite meaningless to further compress the hot words. We therefore transform the object into, an online algorithm that digs out hot words with a recession factor of some kind.

System architecture

Data persistency. The usual trick is to connect every Apache Storm component with an external DB, in this case, the nosql featured redis.

On the other hand, Apache Storm also provides a handy multilang API, helping the developers to better deal with customized non-jvm languages. This is because the communication protocol between Storm components is through a JSON formatted text stream. We can therefore implement this JSON output featuring our preferred language. Given the need to communicate with other components outside Apache Storm(HTTP scraping, Redis, Morphological Analyzer etc.), we chose Python as the developing language. In the later deployment, the use of Python did not bring out further apparent cost rise.

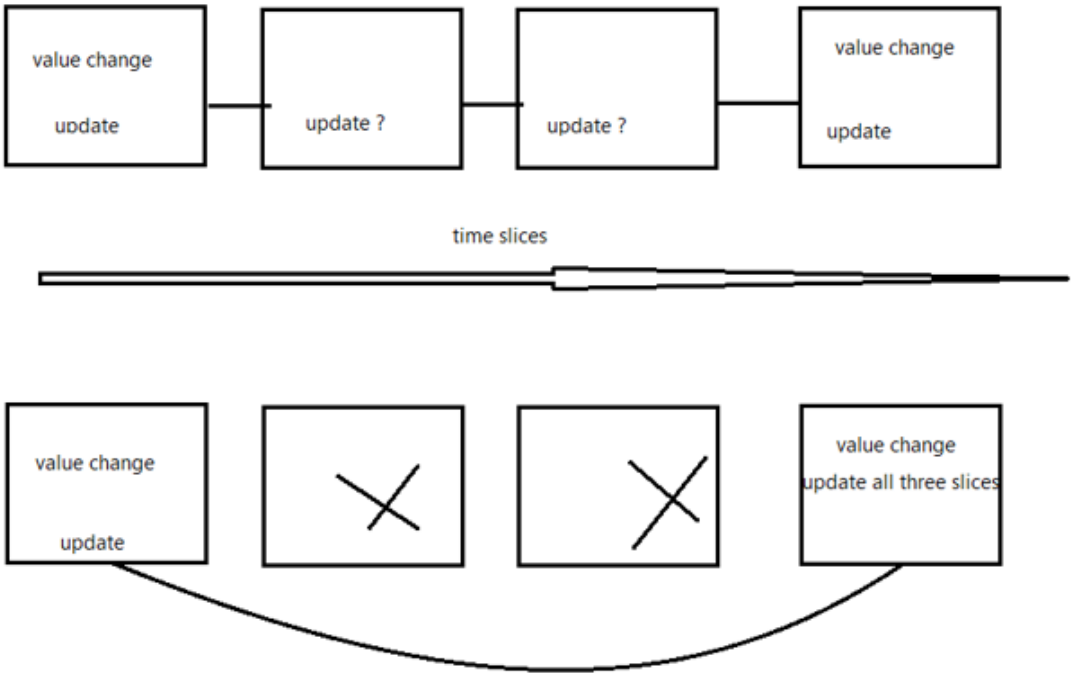

Figure 2 Lazy update within a time slot

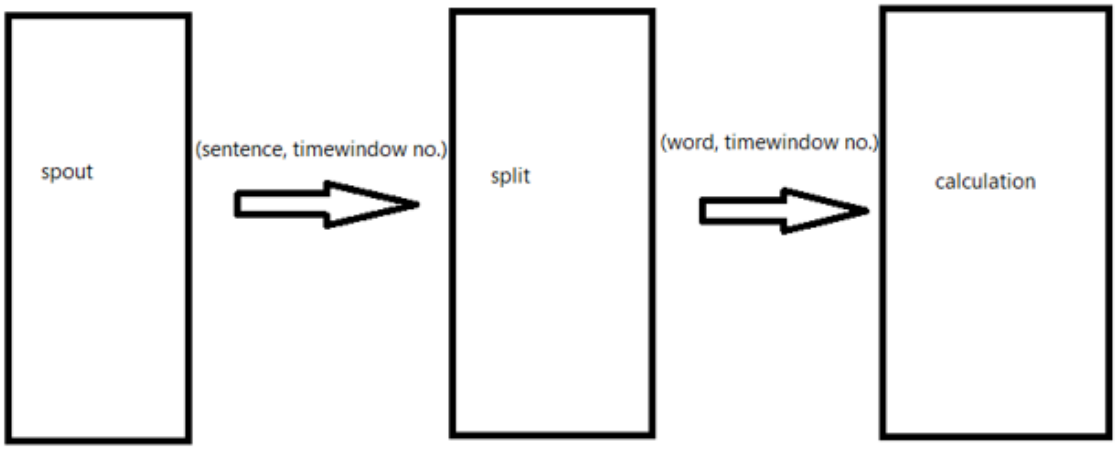

Figure 3 Storm Components

Core algorithm

After derivation, we came to a conclusion that, together with the incrementation of time, every element within a certain time slot can be updated incrementally. This also applies to several global variables used. The conclusion freed us from maintaining a large-scale queue instantly, and every single operation is of complexity O(logN).

After receiving the input of every sentence, we first delete the emojis and stopwords etc. We then put the sentence into the Morphological Analyzer, and ignore every word type except nouns. A pair (noun, timestamp) is then popped to the Storm system. The final component maintains a corresponding table recording necessary variables.

Algorithm 1 Incrementally update global and local variables (redisio.py)

Function $updatePb(p_o, p_j)$:
  $decisionpoint \leftarrow \lfloor 100 * p_j \rfloor$
  $n \leftarrow \lfloor 100 * p_o \rfloor$
  $p_g \leftarrow \binom{100}{n} * p_j^n * (1 - p_j)^{(100-n)}$
  if $n < decisionpoint$ **or** $p_j = 0$ **or** $p_j = 1$ **then**
    Return 0
  endif
  $peakVal \leftarrow \binom{100}{decisionpoint} * p_j^{decisionpoint} * (1 - p_j)^{(100-decisionpoint)}$
  Return $1 - \frac{p_g}{peakVal}$

for each $(word, time)$ **do**
  if $lastupd[word] < time$ **then**
    $p_j[word] \leftarrow p_j[word] + \frac{\frac{number[word]}{total[lastupd[word]]} - p_o[word]}{time}$
    $p_b[word] \leftarrow updatePb(\frac{number[word]}{total[lastupd[word]]}, p_j[word])$
    $p_o[word] \leftarrow \frac{1}{total[time]}$
    $number[word] \leftarrow 1$
    $p_j[word] \leftarrow \frac{lastupd[word] * p_j[word] + p_o[word]}{time}$
    $lastupd[word] \leftarrow time$
  else
    $number[word] \leftarrow number[word] + 1$
    $p_j[word] \leftarrow p_j[word] - \frac{\frac{number[word]}{total[time]} - p_o[word]}{time}$
    $p_o[word] \leftarrow \frac{number[word]}{total[time]}$
  end if
end for
{Every DB write operation is atomic.}

After getting the probability of a certain word b at a certain time slot, we do an integral with it, the factor of which follows the naïve EWMA weight (S. W. Roberts et al.). This ensures that probabilities of earlier time slots have a smaller weight. The sorted result of this accumulation is the intended rank of trendy words we want.

In the actual run, our algorithm is proofed workable. In our test, TV dramas and anime, as well as weather alert, train info and earthquakes are all captured.

Experimental results

We receive approximately 20-30 sentences every 5 seconds in day time. In the single thread model, the process would eventually come to a congestion. After adding paralleled components to each processing phase, the problem is solved, and each step costs no more than 10ms.

To test the system's performance limit at a larger input scale, we simply enlarge our input by repeating every sentence K times. Since our result depends on probability distribution, this will not affect the calculated result. When K=50, we can still solve the problem in real time by adding more components. K=500 will halt the 4GB memory server.

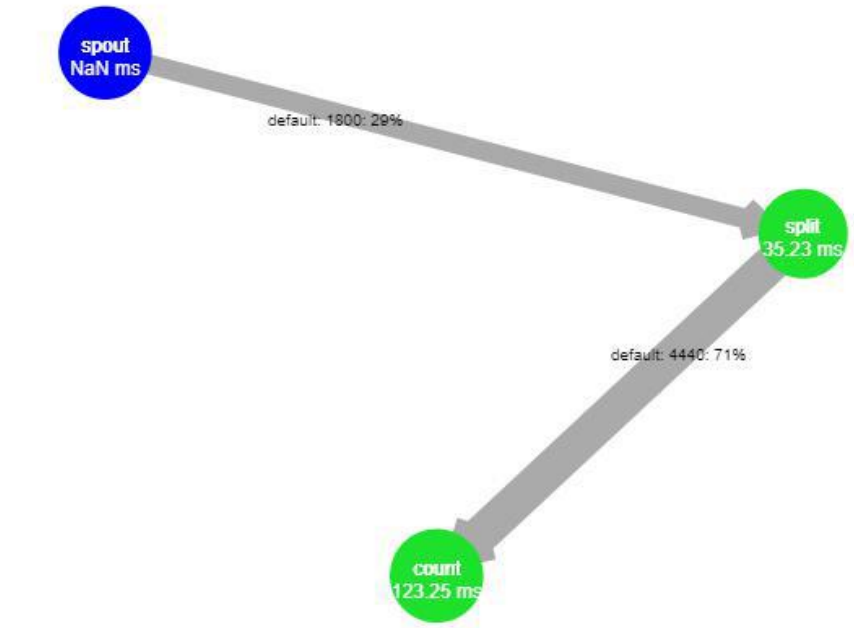

Figure 4 Processing speed when k=50

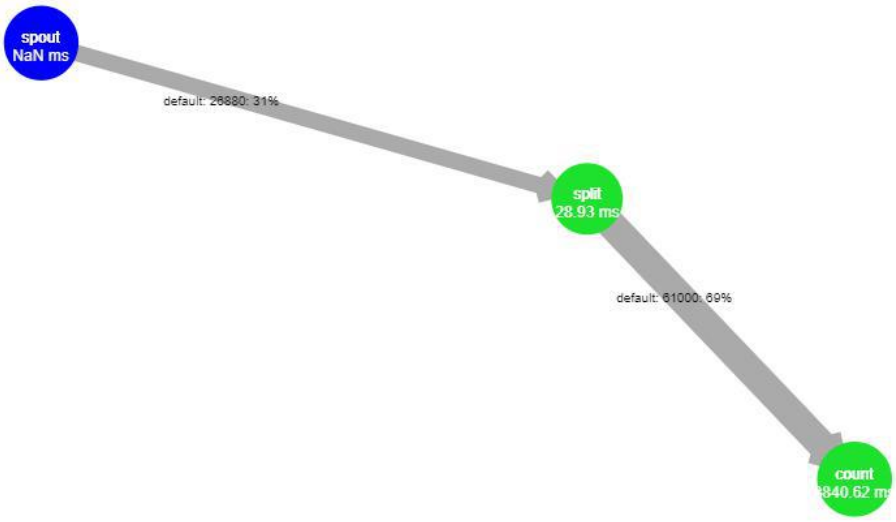

Figure 5 Processing speed when k=500

Developing procedure

Different to Hadoop projects which works on static data, our project is related to real-time data. With real-time results vital to the development process, collaboration between group members takes great effort. The collaboration of coding lasts over 50 hours, with the front end goal and the backend goal achieving simultaneously. The final version of the system worked without any runtime error for over 5 days, having processed over 3 million sentences.

The code is available on:

https://github.com/ckniubi-bigdatateam

排名	单词	最后活跃	最后五分钟词数	趋势
1	さ	5 mins ago	1	
2	文句	55 mins ago	1	
3	リア充	55 mins ago	3	
4	一番	0 mins ago	1	
5	ほも	55 mins ago	1	
6	ヨドバシ	55 mins ago	2	
7	ねこ	5 mins ago	3	
8	ろ	10 mins ago	1	
9	頭	10 mins ago	3	
10	PC	20 mins ago	2	

Figure 6 Results from an actual run