Estimating Trending Topics on Twitter with Small Subsets of the Total Data

Evan Miller

Kiran Vodrahalli

Albert Lee

January 13, 2015

1. Introduction

Since its founding in 2006, Twitter has grown into a corporation dominating a significant proportion of social media today. In recent years, it has become interesting to analyze tweets to determine interesting phenomena ranging from the viral spread of news to the detection of earthquakes [**Burks2014**]. The volume of existing tweets is prohibitively large for standard methods of analysis, and requires approaches that are able to either store large amounts of data in memory for fast access (including parallel approaches to data processing) or approaches that only require sublinear memory while retaining accuracy guarantees. In this paper, we investigate a problem utilizing the latter method.

1.1. Problem Statement

We would like to create an online algorithm that provides a real-time estimate of the frequencies of hashtags on Twitter time series data. Since we want to be able to run this algorithm in real time, we would like for it to be space-efficient: that is, it should not be required to store a large number of hashtag counts at any given time. Ideally, the algorithm's space complexity should be sublinear in the number of hashtags seen. Therefore, our estimate should not require knowledge of the exact frequency history of hashtags in tweets.

More specifically, we will attempt to approximate the k most popular Twitter hashtags in time intervals of length y in order to tell what is trending during that time period. The idea is that estimating the top k hashtags in some time interval provides a model for the topics that are trending on Twitter in that time period.

1.2. Previous Work

Estimating frequencies with high probability and low space complexity has been an interest in algorithms development for a fair amount of time. The Count-Min sketch, developed in the early 2000s, provided an approximate data structure able to maintain counts of keys seen in an online setting in space sublinear to the number of keys, with the size of the structure instead dependent on parameters tunable for different accuracy and guarantee settings [**Cormode2005**].

More recently in 2012, Matusevych et al. developed the Hokusai structure, which gives frequency counts for keys over a range of time [**Matusevych2012**]. The rough idea behind this approach is that in the distant past, we should only care about heavy-hitters, i.e. hashtags with high frequencies in order to estimate the likelihood that the hashtag is trending again. The goal of the time-aggregated Hokusai system is to store older data at decreased precision since the older data also has decreased value in the computation. The time-aggregated Hokusai system works by storing aggregate data in Count-Min sketches each with a 2^i day resolution for the past 2^i days. Each of these Count-Min sketches computes d hash functions from $\{0,1\}^*$ to $\{0,1,...,m-1\}$.

Analyzing what is trending on Twitter is a more recent research area. Twitter itself must give its users hashtags which it deems trending, to both spread the hashtag and to provide an awareness of what is going on in the world to its users. Knowledge of what is trending and how that distribution looks over time is also useful for the business model, which relies on ads to a certain extent. Twitter itself stores all of its frequency data, and looks at recent spikes while conditioning on all of the past frequency data in order to determine estimates of trending likelihood. In 2012, MIT Professor Devavrat Shah built a machine-learning algorithm to predict which topics will trend on Twitter an hour and a half before they do, with 95% accuracy, though the datasets they tested their algorithm on were small [**MITNews2012**].

As a preliminary starting point for our approach to approximating the past frequencies, we will utilize concepts from the Hokusai paper to generalize the Count-Min Sketch scheme to time-series data [**Matusevych2012**].

1.3. Our Approaches

Our approach was to explore whether or not adding historical data would improve the accuracy of trending hashtag detection. To that end, we implemented two algorithms: one called the naive algorithm, the other called the history-sensitive algorithm. We wanted to test whether an algorithm that uses the history (in condensed form) to better understand what is trending at the current time could outperform an algorithm with a very short term memory.

The reason why the history is important is as follows: Some hashtags are constantly present. For example, in our experiments, the hashtags #*porn*, #*gameinsight*, #*teamfollowback* had nearly always had a high frequency in the current time unit. This fact is not surprising given the nature of the first hashtag. The second hashtag, #*gameinsight*, is the name of popular video game maker that people tweet when they win achievements in the game. Finally, #*teamfollowback* is a method Twitter users apply to gather Twitter followers quickly. The rule is if you follow that tag, you follow people who post that tag. Since these sorts of tags are consistently present, they should not be considered trending: they are not new. Therefore, we need to filter these out with the history. The necessity of maintaining approximate counts for a longer history depends on exactly how constant these consistently-appearing tweets actually are.

We had two different algorithms for finding out which hashtags were trending in real time, both space efficient. The naive algorithm does not take into account the history and simply looks at a y-unit before the current y-unit interval to find the current trending hashtags. To determine whether a hashtag was trending, we tested

$$\frac{\mathbf{freq}(\text{hashtag}, \text{present } y\text{-unit})}{\mathbf{freq}(\text{hashtag}, \text{past } y\text{-unit})} > c \tag{1}$$

where c is a threshold we picked to determine whether the hashtag was considered trending. We believe that in order to accurately determine trending hashtags, we will need to use more of the past in order to eliminate the hashtags which are ever-present, and therefore not trending.

The second approach differs in that we do not store all the data of the past, but instead use several Count-Min Sketches to approximate the past frequencies. We call this approach the history-sensitive algorithm. The idea is to remember the past with less accuracy to save space, while weighting the frequency counts of a hashtag in the past appropriately. We used the rule of thumb that the more often a hashtag shows up in the past, the less likely it is trending. However, we also discount the importance of the past by weighting the times long ago less and less as time moves forward.

2. Describing the History-Sensitive Algorithm and Data Structure

We separate the problem of finding trending topics on Twitter into two parts. First, we need to maintain a data structure that efficiently stores data about all occurrences of every hashtag seen in the past. We also maintain a separate data structure that allows us to quickly gather information about the most recent hashtags seen.

We want the former data structure to be very space efficient since it must store data about a very large dataset. For this structure, space efficiency is more important than accuracy since small deviations in such a large dataset should not be significant because the deviations in past data should not greatly affect what is currently trending.

For the latter data structure, accuracy is more important than space efficiency since the structure contains data which more closely relates to which topics are currently trending and the size of the dataset is much smaller.

2.1. Data Structures

2.1.1 History Data Structure

To store data about all occurrences of every hashtag seen in the past, we use a modified version of the time-aggregated Hokusai system **[Matusevych2012]**, which is an extension of the Count-Min sketch. We already described this data structure in **Previous Work**. To the Hokusai structure we add another Count-Min sketch that combines the information from the Hokusai Count-Min sketches. We call this external Count-Min sketch the Kernel, since it acts as a weighting function on the CM sketches in the Hokusai structure. Its role is to depreciate the value of older data. We denote the CM sketches of the Hokusai structure as a vector $\mathbf{M} = \{\overline{M}, M^0, M^1, ..., M^{\lfloor \log(T) \rfloor}\}$, where $\overline{M}$ is the Count-Min sketch storing the data of a unit time step (of size z), M^j is the Count-Min sketch storing the data for the sketch with a 2^j time resolution, and T is an upper bound on the number of z-units stored by the data structure. Then, the kernel function is given by

$$k(\mathbf{M}) = \overline{M} + \sum_{j=0}^{\lfloor \log(T) \rfloor} \frac{M^j}{2^j} \tag{2}$$

For each hashtag, the kernel sketch stores a positive value that approximates how often the hashtag showed up in the past. In **Correctness** we will show that this aggregate Count-Min sketch weights the hashtags that occurred $i > 0$ days ago with weight approximately $\frac{1}{i}$.

We will refer to the combined data structure of Hokusai and the Kernel as the History.

2.1.2 Current-Window Data Structure

Our data structure for holding hashtags seen in the last y-unit consists of three components: a max Fibonacci heap, a queue, and a hash table. We refer to these components collectively as the Current-Window. The keys for the hash table are the hashtags, and the values stored in the table are (frequency, pointer to corresponding node in heap) pairs. The hash table exists so that we can keep track of exact frequency counts for each y-unit. The queue contains (hashtag, timestamp) pairs, each of which is inserted upon seeing a hashtag in the input stream. The purpose of the queue is to determine when a hashtag count is too old to be maintained in the present y-unit window. Finally, we have a Fibonacci heap to prioritize the hashtags based on how trending they currently are. We provide a heap priority function, which defines what trending means for the whole data structure. For a specific hashtag, we define the priority function to be

$$\mathbf{priority}(\text{hashtag}) = \frac{\mathbf{freq}(\text{hashtag}, \text{present } y\text{-unit})}{\mathbf{History}[\text{hashtag}]} \tag{3}$$

The idea is that again, we want to be inversely proportional to the history, here corrected by more knowledge and clever approximation schemes to save space.

2.2. Algorithm Pseudocode

2.2.1 Updating History Data Structure

Algorithm 1 describes the necessary steps to maintain the History data structure as new input is provided. We perform time aggregation to fill the Kernel, K. This algorithm is only called when a z-unit has passed since the last time the algorithm was called.

Algorithm 1 Update History

```
for all $i$ do
    Initialize Count-Min sketch $M^i = 0$
Initialize $t = 0$
Initialize Count-Min sketches $\overline{M} = 0$ and $K = 0$
while data arrives do
    Aggregate data into sketch $\overline{M}$ for current $z$-unit
    $t \leftarrow t + 1$ (increment counter)
    for $j = 0$ to argmax $\{l$ where $t \bmod 2^l = 0\}$ do
        $K \leftarrow K + 2^{-j}(\overline{M} - M^j)$ (update the Kernel)
        $T \leftarrow \overline{M}$ (back up temporary storage)
        $\overline{M} \leftarrow \overline{M} + M^j$ (increment cumulative sum)
        $M^j \leftarrow T$ (new value for $M^j$)
    $\overline{M} \leftarrow 0$ (reset aggregator)
```

2.2.2 Updating heap and hash tables

We run Algorithm 2 in the main loop, every time a single piece of data arrives. For each new piece of data, we also check whether the data we are storing has grown outdated – that is, older than a y-unit before the most current piece of data arrived.

2.2.3 Finding the trending hashtags

We use Algorithm 3 to calculate the top k hashtags by querying the heap. The heap is the structure responsible for maintaining the priority of the various hashtags, and provides an efficient way of implementing a changing priority queue.

3. Analyzing the History-Sensitive Algorithm

Symbol	Meaning
d	Number of hash tables for each Count-Min sketch
m	Size of each hash table in each Count-Min sketch
s	Number of distinct hashtags in the Current-Window
z	Time resolution of the unit-sized Count-Min sketches
T	Upper bound on the number of z-units of data stored in History
x	Total number of hashtags in the Current-Window
y	Time interval of data contained in the Current-Window

Table 1: *Variables referenced in this section*

Algorithm 2 Update Current-Window

```
while data arrives do
    if the current $z$-unit is different than that of the last hashtag seen then
        Do all the end-of-$z$ aggregation for the History structure as detailed in Algorithm 1.
        for all elements in the Fibonacci heap do
            look up the hashtag corresponding to this node in the hash table
            Update the key of the node to: priority(hashtag)
    if queue is not empty then
        Peek at end of queue.
        if the timestamp + $y$ is before the current time then
            Look up the hashtag in the hash table and decrement the stored frequency.
            if the frequency is now 0 then
                Delete the node in the heap pointed to by this entry in the table.
                Delete this entry in the hash table.
            else
                Update the key of the node with the new frequency.
    if hashtag is in hash table then
        Increment the frequency stored at that entry.
        Update the key of the node in the Fibonacci heap.
    else
        Insert a new node into the Fibonacci heap with the appropriate key and value.
        Insert the hashtag into the hash table with a pointer to this node.
```

Algorithm 3 Top *k* trending hashtags

```
Perform $k$ delete-max operations on the Fibonacci heap, storing each of the nodes deleted in a list $L$.
for all nodes $N$ in $L$ do
    Announce that the hashtag associated with $N$ is trending.
    Insert $N$ into the Fibonacci heap.
```

3.1. Correctness

Our history-sensitive algorithm finds the k hashtags that have the maximum value of $\frac{\textbf{freq}(\text{hashtag,present } y\text{-unit})}{\textbf{History}[\text{hashtag}]}$.

Claim 1. *The value for hashtag x in the aggregate Count-Min sketch of the History data structure is within a factor 4 of $\overline{M}(x) + \sum_{i=1}^{T} \frac{1}{i} *$ (value for x in a Count-Min sketch using the same hash functions for all hashtags occurring i z-units ago).*

Proof. First, we use Theorem 4 in [**Matusevych2012**] which states that "At t, the sketch M^j contains statistics for the period $[t-\delta, t-\delta-2^j]$ where $\delta = t \mod 2^j$."

Let b be the location in the aggregate Count-Min sketch containing the value returned when x is queried. Let h be any instance of any hashtag that appeared on z-unit p such that seeing h incremented counters in position b.

Case 1: $p = t$

Then, the statistics for h are recorded in $\overline{M}$ and are not in any M^j.

Case 2: $2^i > t - p \geq 2^{i-1}$ for some $i > 0$

By Theorem 4, for all $j \leq i-2$, M^j does not contain statistics for p since $p \leq t - 2^{i-1} \leq t - \delta - 2^{i-2}$.

Therefore, the increments that occurred in the Count-Min sketch for hashtags occurring i z-units ago contribute at most $\sum_{j=i-1}^{T} 2^{-j} < 2^{2-i}$ to the value in position b.

Let q be the largest j such that $t - \delta - 2^j \geq p$. Then $p \leq t - \delta - 2^q$. Let $\lambda = t \mod 2^{q+1}$. Then $\lambda = \delta$ or $\lambda = \delta + 2^q$. Since q is the largest j such that $t - \delta - 2^j \geq p$, $p > t - \lambda - 2^{q+1}$. Also, $p \leq t - \delta - 2^q \leq t - \lambda$, so M^{q+1} contains statistics about h. For all $j \geq i$, $t - \delta - 2^j < t - 2^j < p$, so $q \leq i - 1$. Thus, incrementing the counter in M^{q+1} contributed at least 2^{-i} to the value in position b. Thus, the contributions to the sum are within a factor 4 of $\frac{1}{t-p}$.

Therefore, summing over all hashtags that increment counters in position b gives $\overline{M}(x)$ for all hashtags that occurred on z-unit t, and within a factor 4 of $\sum_{i=1}^{T} \frac{1}{i} *$(value for x in a Count-Min sketch using the same hash functions for all hashtags occurring $i > 0$ z-units ago). □

Given this proof and the fact that Count-Min sketches return approximate frequency counts, we know the priority function to be approximately:

$$\textbf{priority}(\text{hashtag}) \approx \frac{\textbf{freq}(\text{hashtag, present } y\text{-unit})}{\textbf{freq}(\text{hashtag, present } z\text{-unit}) + \sum_{i=1}^{T} \frac{1}{i} * \textbf{freq}(\text{hashtag}, i\ z\text{-units ago})} \tag{4}$$

This seems to be a desirable function to maximize since it finds hashtags that are common in the last y-unit that have been comparatively infrequent in the past. This function is good since it especially emphasizes hashtags that are different than those seen in the past few z-units. This ensures that the same items do not stay trending for too long.

3.2. Runtime Analysis

We analyze the time require for processing the insertion of a hashtag. It takes amortized $O(d)$ time to update the History. It takes expected $O(1)$ time to check if the hashtag is in the hash table.

If it is, it requires $O(1)$ time to increment the frequency, $O(d)$ time to compute the new key, and $O(1)$amortized time to update the key since it will be non-decreasing.

Otherwise, it requires $O(d)$ time to compute the key value, $O(1)$ time to insert the new node in the heap, and $O(1)$ time to insert into the hash table.

Thus, our algorithm takes $O(d)$ amortized time + expected $O(1)$ time to process a new hashtag.

Case	Amortized Time
Processing the insertion of a new hashtag	$O(d)$
Processing the removal of a hashtag from the Current-Window	$O(d + \log(s))$
Updating History and Current-Window at the end of a z-unit	$O(md + ds + s\log(s))$
Querying for the top k trending items	$O(k\log(s))$

Table 2: *Time analysis summary*

Next we analyze the required time to process the removal of a hashtag from the Current-Window. It takes $O(1)$ time to verify that the queue is not empty. It takes $O(1)$ time to peek at the end of the queue and verify that the timestamp $+y$ is before the current time. It takes $O(1)$ time in expectation to look up this hashtag and decrement its frequency. Then, it takes $O(1)$ time to check if the frequency is 0.

If so, it takes $O(\log(s))$ amortized time to delete the node in the heap and $O(1)$ time to delete the entry in the hash table.

Otherwise, it takes $O(d)$ amortized time to compute the new key for the hash table and $O(\log(s))$ amortized time to update the heap given this key.

Thus, our algorithm requires $O(\log(s))$ amortized time + expected $O(1)$ time $+O(d)$ time to remove a hashtag from the Current-Window.

Now we analyze the time due to the end-of-day updates to the History and the resulting updates to the heap. By Lemma 5 in **[Matusevych2012]**, the amortized time required to do all end-of-day aggregation is $O(md)$. Then, for each of the s nodes in the heap, it takes $O(d)$ time to compute each updated key and $O(\log(s))$ amortized time to update the heap given the new key.

Thus, it takes $O(md + s\log(s))$ amortized time $+O(ds)$ time to do all necessary updates at the end of the day.

Querying the data structure for the top k trending items takes $O(k\log(s))$ amortized time for **delete-max** operations, $O(k)$ time to announce that these items are trending, and $O(k)$ amortized time to reinsert these nodes into the heap.

Thus, it takes $O(k\log(s))$ amortized time to determine what is trending.

3.3. Spatial Analysis

The History requires $O(md)$ space for each Count-Min sketch, so it requires a total of $O(md\log(T))$ space. Each node in the heap requires a constant amount of space, so the heap requires $O(s)$ space. The hash table always contains at most s entries with each entry requiring a constant amount of space. Also, in order to maintain an expected $O(1)$ lookup time, the hash table needs to have $O(s)$ bins. Thus, the hash table requires $O(s)$ space. The queue requires an entry for every hashtag seen in the last y-unit, so it requires $O(x)$ space.

Thus, everything requires $O(md\log(T) + x)$ space since $s < x$.

4. Design Choices

4.1. Choosing the parameters y and z

For y, the unit time period we attempted to find the trending hashtags for, we chose a 3-hour interval. For z, the unit time period for the History data structure, we chose day-length intervals. These values made sense because the testing data we used occurred over a 30-day interval, and we wanted to avoid too fine a temporal resolution so that we could run our code on the data in a reasonable amount of time.

4.2. Parameters of the History Data Structure

Since $T = 30$ (our data is for the month of September 2014, therefore $\sim 2^{(6-1)}$ days is definitely enough), we store 6 Count-Min sketches in the Hokusai portion of the History data structure, with an extra Count-Min sketch for the Kernel.

Each of these CM sketches has $m = 3500$ and $d = 20$. From the proof of CM sketch, we have that

$$m = \lceil \frac{e}{\epsilon} \rceil \tag{5}$$

and

$$d = \lceil \ln(\frac{1}{\delta}) \rceil \tag{6}$$

where the error in the query is in factor of ϵ with probability $1 - \delta$. By Theorem 1 in [**Matusevych2012**], we have that if n_x is the true count of a hashtag stored in the CM sketch, and c_x is the estimate, we are guaranteed $n_x \leq c_x \leq n_x + \epsilon \sum_{x'} n_{x'}$. By choosing $m = 3500$, we have that $\epsilon = \frac{e}{3500}$ and $1 - \delta = 1 - \frac{1}{e^{20}}$, which is to say that we are within that error with very high probability. The error term is a multiplier on the total number of hashtags, of which there are ~ 15 million, as an upper bound. Thus we are within very high probability of the true count with range ~ 12000, on a given count.

By using less memory we increase the amount of error. These error terms are added to the denominator of our priority function, and as a result, they prevent hashtags with very relatively low frequency from being identified as trending, which is a very desirable property.

4.3. Choosing the Flavor of Heap

We chose a Fibonacci heap in the hopes that there would be a larger number of **decrease-key** operations, which has a better theoretical bound in Fibonacci heaps.

5. Testing Procedure

We downloaded the September 2014 archive of Twitter data to serve as test data for our algorithms [**Twitter2014**]. We then parsed the data into (timestamp, hashtag) pairs for all thirty days worth of data. Using these files as pseudo-real-time input, we used the last week of the data as a benchmark for comparison between the naive and the history-sensitive algorithms. We divided the data into three-hour chunks, and queried for the top ten hashtags in each of the three-hour time blocks in the last week of the data using both algorithms. Note that the history-senstive algorithm was required to parse through the first three weeks as well in order to build its history, whereas the naive algorithm was only required to pay attention to six-hour chunks at a time, and thus never needed to look a bit beyond the scope of that week.

We used the Jaccard similarity to arrive at a similarity score between the two methods. We also investigated the hashtags that both algorithms came up with for a given 3-hour interval, and verified that they were in fact trending at that time using news and Twitter media.

6. Results

6.1. Runtime Measurements

We use the space complexities of the data structures and the sizes of the respective inputs to determine approximately how much space is used by each approach, and compare them. The naive algorithm's runtime on one week's worth of hashtags was ~ 15 minutes. Note that the naive algorithm did not have to take into account the previous 3 weeks, since it only ever depended on the frequencies from 3 hours before the present time. The history-sensitive algorithm on the other hand first had to build a history from the first three weeks of the month before running on the last week for comparison to the naive algorithm. The total running time for the history-sensitive algorithm was ~ 2 hours.

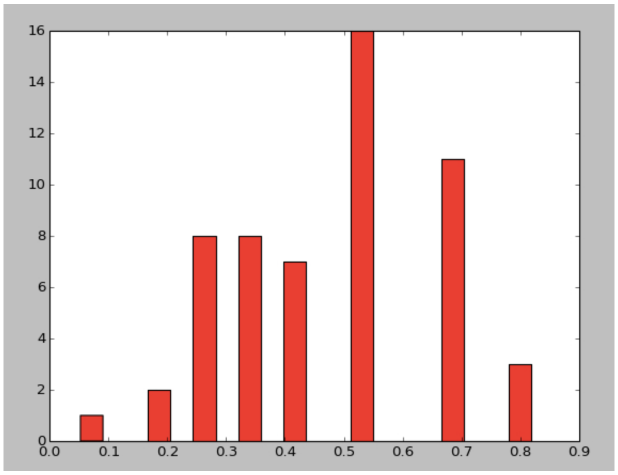

Figure 1: *Distribution of Jaccard Similarities*

6.2. Are the Algorithm Outputs Actually Trending?

6.2.1 The Intersection Distribution

First, we see how many hashtags were marked as trending by both the naive and history-sensitive algorithms in the same time slots. For each 3-hour interval, we calculate the Jaccard Similarity, a value in the range $[0, 1]$, given by

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{7}$$

where A and B are the sets of top 10 hashtags for a given 3-hour interval for each algorithm. We then calculate the mean and standard deviation of the Jaccard similarity over the 56 different 3-hour intervals of the last week of September 2014.

This value gives us a metric for understanding how similarly the two algorithms performed.

We plot this data in the histogram shown in **Figure 1**. The x-axis is the Jaccard similarity, and the y-axis is the count. The mean was 0.47, and the standard deviation was 0.18. From the plot, we can see that the distribution of the Jaccard similarities is roughly a centered normal distribution around 0.5. A Jaccard similarity of 0.5 means about half the hashtags are the same in each 3-hour time interval between the two algorithms.

6.2.2 Checking the Algorithm-Defined Trends Against Real Life

In **Figure 2**, we have superimposed pictures from news articles that the tweets describe on a time axis, as the hashtag referencing the article starts trending. The Jaccard similarities for each of these three-hour intervals were all above 0.5, so both algorithms found the hashtags that we graph in the figure. We checked that the news was released (via Yahoo! News, Twitter, Google News, ESPN, and others) at around the time our algorithms say the hashtag relating to the news begins to trend. A fair number of the trending hashtags are related to sports and TV shows.

As we can see in the figure, our algorithm detected the start of the TV show "How to Get Away With Murder", which was released on September 25, 2014. Another TV-related hashtag was $\#dailyshowgonetoofar$, when Jon Stewart called One Direction member Zayn Malik a terrorist (on the 26^{th}) and caused a huge outrage. Sports events that occur real-time on TV also have the trending profile. There were three in this section of six three-hour intervals. First, Derek Jeter hit a walk-off homerun in the final time at bat on the Yankees, and naturally on Twitter, fans were excited. This event occurred on the 25^{th}. The Ryder Cup is

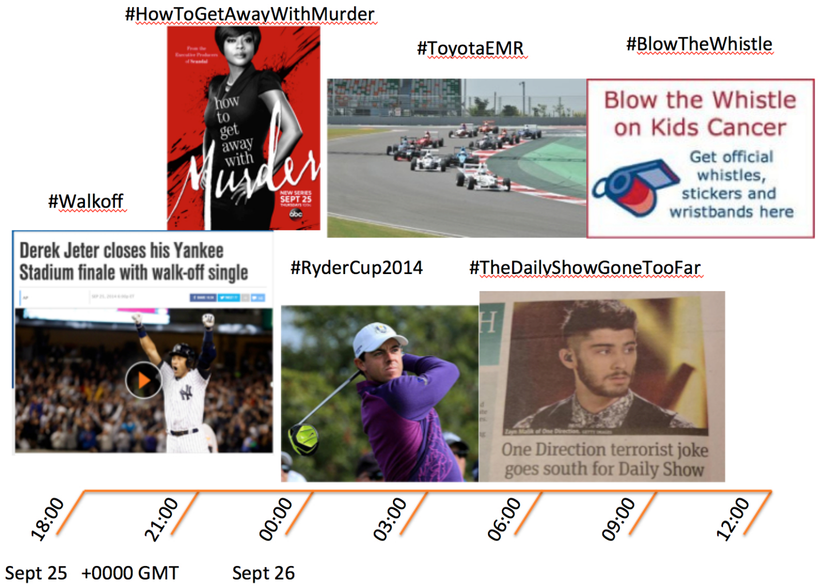
#HowToGetAwayWithMurder
how to get away with Murder
#ToyotaEMR
#BlowTheWhistle
Blow the Whistle on Kids Cancer
Get official whistles, stickers and wristbands here
#Walkoff
Derek Jeter closes his Yankee Stadium finale with walk-off single
#RyderCup2014
#TheDailyShowGoneTooFar
One Direction terrorist joke goes south for Daily Show
18:00
21:00
00:00
03:00
06:00
09:00
12:00
Sept 25 +0000 GMT
Sept 26

Figure 2: *News Visualization of Trending Tweets*

a major golf tournament that began on the 26^{th} and lasted through the 28^{th}, and its beginning launched a wave of tweets. The last sports-related hashtag was the beginning of the ToyotaEMR car race that took place in India on September 26^{th}. Finally, *#blowthewhistle* related to a kids cancer campaign that had lasted throughout September, which perhaps got a boost as September ended to raise some last minute funds.

7. Discussion

7.1. Comparison of Naive and History-Sensitive Algorithms

Based on the approximately normal, centered distribution of the Jaccard similarities, we observe that the history-sensitive approach performed about as well as the naive approach in terms of getting trending tweets. However, the naive algorithm uses considerably less space and time (since we only need to check the y-unit before the current y-unit) to get similar results. Thus we conclude that the space efficiency of the naive approach outweighs the benefits of the history-sensitive approach for this particular task. Furthermore, in our development of the history-sensitive algorithm, we utilized larger errors as a method of smoothing to perform decently, suggesting that our approach for evaluating the history may not be the best one for tweet-trending detection.

However, we note that from inspection the history-sensitive algorithm has more staying power than the naive approach. While in the naive approach, it is rare for adjacent y-units to have the same trending tweets, it is a more common occurrence in the history-sensitive approach when it makes sense (for instance, there was an event going on for the whole day: the naive algorithm would immediately reduce showing the hashtags related to that event, while the history-sensitive algorithm would not discard that hashtag so easily).

Because Twitter limits access to its data, the datasets on which we tested our algorithms only contained 1% of the total number of tweets. If we were to run our algorithms on the entire dataset, we would expect that the space efficiency of the history-sensitive algorithm would improve relative to the naive algorithm because the naive algorithm would need to store 100 times more data while the history-sensitive algorithm could just reduce the size of the y-units to keep it nearly as space efficient. We would not expect this to greatly affect the history-sensitive algorithm's accuracy because we could still process the same number of recent tweets to compare against the History structure. Thus, on the full dataset, we could still reasonably expect the history-sensitive algorithm to be at least as space efficient as the naive algorithm.

7.2. Applicability to Real-Time Data

In this paper, we only simulated our algorithm running on live data, since we did not have good access to the live Twitter firehose. In terms of performance, despite being overshadowed by the simpler naive method due to their similar behavior (capturing a fair number of the same hashtags), the history-sensitive algorithm did decently well. It took a few hours to run our algorithm on a 15 million hashtag dataset, with relatively large data structures to be updated and run in Python no less, a slower language.

Were we to apply our algorithm to real-world real-time data, we would re-implement our code into C or another space and time efficient language.

Furthermore, for real data from all of the Twitter firehose, we would require a better hardware infrastructure dedicated to processing all the data. Ideally we would use distributed systems and would parallelize our code before running it. The fact that the code is broken up into disparate parts makes it easier to implement in parallel.

Therefore, given some adjustments, the algorithms developed in this paper open the doors to new explorations of topic modeling the Twitter data stream.

8. Future Work

With respect to the algorithms specifically, in the future we would want to adapt our approach to the history-sensitive algorithm and play with the parameters a bit more. We would also like to try different kernel functions to see how they performed. Another approach to utilizing the history information would be instead of weighting by inverse-exponentially-weighted frequency in the past for a given hashtag, attempt to calculate some score of periodicity for the hashtag. If the hashtag has low periodicity over a given time scale, then even if it appears multiple times in the past, our algorithm will avoid discarding it as non-trending despite the potentially high frequency count (because the past is long).

Then, we would like to try adapating our algorithms to real-time data: that is, hook up our algorithms to a significant portion of the Twitter firehose, after setting up the necessary hardware to process that amount of data. In order to aid with the processing, we would also ideally use a distributed system and parallelize our code for better performance.

We also know that an estimation of the top k hashtags in a given time interval could provide a topic model for tweets. It could be interesting to use the top k hashtags to build a time-sensitive topic model and analyze quantities including the speed of topic change, the variety of topics, and so on.

Even though it is perhaps not suited for the specific task of identifying trending hashtags, the approach of the history-sensitive algorithm may be useful in other domains. Future work could also investigate the performance of the history-sensitive algorithm on other data streams of interest, for instance, Wikipedia edits – our goal would be to estimate which Wikipedia topics are being edited the most at any given time interval of some length. Another interesting application of this algorithm to test would be the real-time estimatation of the hottest selling stocks on Wall Street to see if either of these settings results in a significant improvement over the naive algorithm.

9. Appendix I: Code and Visualizations

We provide a link to all our code. In the repository, there are also the results files, Powerpoint slides, and citations for the pictures we used.

We also provide a link to an online hosting of the frequency visualization. The frequency visualization is a histogram of hashtag frequencies, displayed over time. We cherry-picked a set of interesting hashtags based on our observations, and then made this visualization to present the changes in frequency in 3-hour intervals over time of these hashtags.

The fact that these frequencies change over time and spike and fall in turns suggests that when the hashtag frequencies spike, the hashtags are trending. The goal of our algorithms is to identify the hashtags (of a much larger set) which are trending at a given point in time.

So we hope that for a selected 3-hour interval, the hashtags that our algorithms say are trending include the ones that are trending at that time in this visualization.

In our code, we made use of two libraries written by other people, which we modified. One of these was a Count-Min sketch implementation in Python, and another was a Fibonacci heap implementation, also in Python. These program files are in our codebase, with appropriate citations at the top of each document. Furthermore, we used multiple standard Python libraries in our code.

10. Appendix II: Top-*k* Hashtags Results

To see all 56 3-hour intervals for both algorithms, see the results folder of our Github. We display the top 10 hashtags for each 3-hour interval for both the naive algorithm and the history-sensitive algorithm, along with their heap priorities. Note that the heap priorities span a reasonable range from 0 to 1, and are not clustered around 1 or 0, implying the algorithms are actually differentiating the hashtags as trending and not trending.

References

[Burks2014] Burks, L., Miller, M., and Zadeh, R. (2014). RAPID ESTIMATE OF GROUND SHAKING INTENSITY BY COMBINING SIMPLE EARTHQUAKE CHARACTERISTICS WITH TWEETS. *Tenth U.S. National Conference on Earthquake Engineering Frontiers of Earthquake Engineering*. Retrieved from http://www.stanford.edu/ rezab/papers/eqtweets.pdf

[Cormode2005] Cormode, G., and Muthukrishnan, S. (2005). An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms, 55*(1), pp. 58-75.

[MITNews2012] Hardesty, Larry. *"Predicting What Topics Will Trend on Twitter."* MIT News Office. MIT, 1 Nov. 2012. Web. 13 Jan. 2015. http://newsoffice.mit.edu/2012/predicting-twitter-trending-topics-1101.

[Matusevych2012] Matusevych, S., Smola, A., and Ahmed, A. (2012). Hokusai–Sketching Streams in Real Time. *UAI '12: 28th conference on Uncertainty in Artificial Intelligence*, pp. 594-603.

[Twitter2014] Twitter, Inc. (2014). *Archive Team JSON Download of Twitter Stream 2014-09*. Retrieved from https://archive.org/details/twitterstream.