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

Evan Miller

Kiran Vodrahalli

Albert Lee

January 13, 2015

**Abstract**

*Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.*

## 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 the exact frequency history of hashtags in tweets.

More specifically, we will attempt to approximate the $k$ most popular Twitter hashtags in the 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

Some approaches attempt to do this by storing all of the frequency data, and looking at recent spikes while conditioning on all of the past frequency data in order to determine estimates of trending likelihood. We would like to improve the space complexity of this solution. Our approach will differ in that we will not store all the data of the past, but instead use several Count-Min Sketches to approximate the past frequencies.

MIT people played with Tweet data to predict future trends **[MITNews2012]**.

As a preliminary starting point for approximating the past frequencies, we will utilize concepts from Matusevych et al.'s Hokusai paper to generalize the Count-Min Sketch scheme to time-series data **[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 $\{0,1,...,m-1\}$.

## 1.3. Our Approaches

why we don't use raw frequency counts for the past hour: there are some hashtags that are constantly present (for example, #porn, #gamesinsight, #teamfollowback (a way to get Twitter followers fast)) these should not be counted as 'trending', so we need to filter these out with the history. what remains is 'what is trending'

Essentially, we must estimate a probability distribution for a finite set of labels in some moving time window of fixed size. Our labels will be Twitter hashtags, and we will take the window size to be 3 hours.

We will store the exact counts for the present (3 hour window), and continuously update the past and present as new data streams in.

We had two different algorithms for finding out which hashtags were trending in real time, both space efficient. The first algorithm, which we will describe as the naive algorithm, does not take into account the history and simply looks at the three hours before the current 3-hour interval to find the current trending hashtags. 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 baseline comparison for our performance will be the naive version of frequency tallying – we will keep track of the entire history of frequencies, and will use the past to inform the present probability as to whether or not a given hashtag is trending. We also plan to provide a graphic of the top k hashtags, with histogram changing in real time as the estimated frequencies change. Regarding the data, we would ideally gain access to Twitter's firehose of tweets (as only a small subset of the true data is provided for those without access).

# 2. Describing the Algorithm

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 previously 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. Denoting the CM sketches of the Hokusai structure as a vector $\mathbf{M} = \{\overline{M}, M^0, M^1, ..., M^{\lfloor \log(T) \rfloor}\}$, where $\bar{M}$ 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

$$k(\mathbf{M}) = \overline{M} + \sum_{j=0}^{\lfloor \log(T) \rfloor} \frac{M^j}{2^j} \tag{1}$$

For each hashtag, the kernel sketch stores a positive value between 0 and 2 (though typically $\leq 1$) 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 $y$-length time period 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 queue contains (hashtag, timestamp) pairs, each of which is inserted upon seeing a hashtag in the input stream.

The heap has keys equal to (frequency in last $y$-length time period) / (value in Hokusai data structure) for a specific hashtag, and the value stored in the node is the corresponding hashtag.

## 2.2. Algorithm Pseudocode

### 2.2.1 Updating Hokusai data structure

Algorithm 1 describes the necessary steps to maintain the Hokusai data structure as new input is provided. We perform Time Aggregation.

**Algorithm 1** Update History

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

#### 2.2.2 Updating heap and hash tables

This is 2.

#### 2.2.3 Finding the trending hashtags

This is 3.

## 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*

### 3.1. Correctness

This algorithm finds the $k$ hashtags that have the maximum value of (frequency in last $y$-unit) / (value in History data structure).

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

**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 Hokusai 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:

$$\frac{\text{frequency in last } y - \text{length time period found at the table entry}}{\text{new value in Hokusai data structure}}$$

  **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 this node pointed to by this entry in the table to the proper value given 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.

*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 day $p$ such that seeing $h$ incremented counters in position $b$.

Case 1: $p = t$

Then the statistics for $h$ are recorded in $\bar{M}$ 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 $i$ days 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 of 4 of $\frac{1}{t-p}$.

Therefore, summing over all hashtags that increment counters in position $b$ gives $\bar{M}(x)$ for all hashtags that occurred on day $t$, and within a factor of 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 $i > 0$ days ago). □

This value is approximately (frequency in last $y$-unit) / (freq today + $\sum_{i=1}^{T} \frac{1}{i} *$ (frequency of hashtag $i$ $z$-units ago)). 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 days. This ensures that the same items do not stay trending for too long.

## 3.2. Runtime Analysis

Processing the insertion of a hashtag takes the following time. It takes amortized O(d) time to update the History. It takes expected O(1) time to check if in 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 nondecreasing.

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.

Processing the removal of a hashtag from the Current Window takes the following time. It takes O(1) time to verify that the queue is not empty. It takes O(1) time to look 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.

| 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 day | $O(md + ds + s\log(s))$ |
| Querying for the top $k$ trending items | $O(k\log(s))$ |

**Table 2:** *Time analysis summary*

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.

The end-of-day updates to the History and the resulting updates to the heap take the following time. By Lemma 5 of the Hokusai paper, 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 + slog(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 whatâĂŹs 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 still in the current window, 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 $z$

### 4.2. Parameters of the History Data Structure

### 4.3. Choosing the Flavor of Heap

### 4.4. Choosing the Current-Window Heap Function

## 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) format 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. Performance 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.

#### 6.1.1 Naive Algorithm

Space measurements (running on one week's worth of hashtags) (estimate 6 hours worth of hashtags)

Time (running over one week): $\sim$ 15 minutes

#### 6.1.2 History-Sensitive Algorithm

(we had to build a history over the first three weeks of the month in order to determine the History data structure's values for the week we ran the algorithm over)

Space measurements (running on one week's worth of hashtags):

Time (running over one week): $2 - 3$ hours

### 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

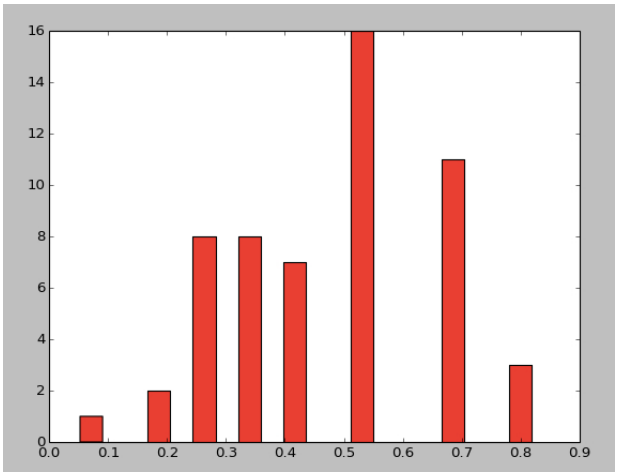


**Figure 1:** *Distribution of Jaccard Similarities*

value in the range $[0, 1]$, given by

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

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 it below 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 History-Sensitive Algorithm

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. We checked that the news was released (via Yahoo! News and Twitter) at around the time our algorithm says the hashtag relating to the news begins to trend.

#### 6.2.3 Naive Algorithm

We consider a few snapshots of duration 3 hours each during the last week of the month.

## 7. Discussion

### 7.1. Comparison of Naive and History-Sensitive Algorithms

### 7.2. Applicability to Real-Time Data

do well to get month's worth of data in a few hours (2-3)


**Figure 2:** *News Visualization of Trending Tweets*

implementation wouldnt run in python, do it in C or something

for real data from firehose (all of it), we would use better infrastructure dedicated to processing all the data would use parallel setup (this scheme is adaptable to parallel setup)

seems reasonable if we have proper data storage setup.

## 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.

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 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 not perhaps irrelevant to 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, as well as to an online hosting of the frequency 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 in 6 Interesting Time Chunks

In [FIGURE WHATEVER] we present side-by-side the top 10 hashtags in six 3-hour intervals for both the naive algorithm and the history-sensitive algorithm, along with their heap priorities. Note that the heap priorities span a fair 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. To see all 56 3-hour intervals for both algorithms, see our Github.

## 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.