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

Albert Lee, Evan Miller, and Kiran Vodrahalli

January 12, 2015

1 Introduction

Things to add: abstract clear definition of problem and motivation adapt proposal -¿ introduction stuff background work: cm-sketch, hokusai (algorithm), specific twitter trending work: ? (this is the closest we could find) (to the best of our knowledge, this work has not been done before)

to note: our goal was to make a real time query algorithm that USES the history (in condensed form) to help understand current time we're remembering the past with less accuracy to save space (realistic model) in order to get good estimates of present trending

this means we are not trying to query the past the way Hokusai is.

how did we approach it overview

data structures description – fib heap – queue – frequency hash table – History (own subsection)

algorithms description - make sure changes in implementation are reflected in algorithm – querying the (History) – updating the data structure —— adding to the data structure ———— frequency hash table update/ heap update / (update history every time block - mention why we can add, etc, some of the theory from the Hokusai paper) —— removing from the data structure (our snapshot of the present) ———— every time unit of the present, we remove outdated frequencies – top k hashtags (real time)

runtime analysis space analysis accuracy analysis

design choices for data structure – data structure modification (History vs Hokusai) – fibonacci heap – heap key function

justification

results

naive algorithm serves as comparison description of naive

the results

future work
references

2 Problem

3 Previous Work

4 Algorithm

4.1 Description

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 these deviations in past data should not greatly effect 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.

4.1.1 Data structure for past tweets

To store data about all occurrences of every hashtag seen in the past, we use a modified version of the time-aggregated Hokusai system[1], which is an extension of the Count-Min sketch. This data structure, which we refer to as the History, is described in full detail in section 4.1.3.

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 with a 2^i day resolution for the past 2^i days. Each of these Count-Min sketches computes m hash functions from $\{0,1\}^*$ to $\{0,1,,n-1\}$.

[1] http://arxiv.org/pdf/1210.4891v1.pdf

To this structure we add another layer that combines these Count-Min sketches into a single Count-Min sketch that depreciates the value of older data. This aggregate Count-Min sketch stores the values $\bar{M} + \sum_{j=0}^{\log(T)} \frac{M^j}{2^j}$ where $\bar{M}$ is the Count-Min sketch storing today's data, M^j is the Count-Min sketch storing the data for the sketch with a 2^j resolution, and T is an upper bound on the number of days stored by the data structure. In section 4.2.3 we will show that this aggregate Count-Min sketch weights the hashtags that occurred $i > 0$ days ago with value approximately $\frac{1}{i}$.

4.1.2 Data structure for current tweets

Our data structure for holding hashtags seen in the last y minutes 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 minutes) / (value in Hokusai data structure) for a specific hashtag, and the value stored in the node is the corresponding hashtag.

4.1.3 Updating Hokusai data structure

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

4.1.4 Updating heap and hash tables

4.1.5 Finding the trending hashtags

4.2 Analysis

4.2.1 Time analysis

Processing the insertion of a hashtag takes the following time. It takes amortized O(m) 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(m) time to compute the new key, and O(1) amortized time to update the key since it will be nondecreasing.

Algorithm 1 Time Aggregation for the Hokusai structure

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 day 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)

Table 1: Variables referenced in this section

Symbol	Meaning
m	Number of tables for each Count-Min sketch
n	Size of each hash table in each Count-Min sketch
s	Number of distinct hashtags in the Current Window
T	Upper bound on the number of days of data stored in History
x	Total number of hashtags in the Current Window
y	Minutes of data contained in the Current Window

Otherwise, it requires O(m) 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(m) 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 minutes 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.

Algorithm 2 Thread to process data input.

1: **while** data arrives **do**
2: **if** the current day is different than that of the last hashtag seen **then**
3: Do all the end-of-day aggregation for the Hokusai structure as detailed in Algorithm 1.
4: **for all** elements in the Fibonacci heap **do**
5: look up the hashtag corresponding to this node in the hash table
6: Update the key of the node to:

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

7: **if** queue is not empty **then**
8: Peek at end of queue.
9: **if** the timestamp + y minutes is before the current time **then**
10: Look up the hashtag in the hash table and decrement the stored frequency.
11: **if** the frequency is now 0 **then**
12: Delete the node in the heap pointed to by this entry in the table.
13: Delete this entry in the hash table.
14: **else**
15: Update the key of this node pointed to by this entry in the table to the proper value given the new frequency.
16: **if** hashtag is in hash table **then**
17: Increment the frequency stored at that entry.
18: Update the key of the node in the Fibonacci heap.
19: **else**
20: Insert a new node into the Fibonacci heap with the appropriate key and value.
21: Insert the hashtag into the hash table with a pointer to this node.

Algorithm 3 Algorithm to find top k trending items.

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

Table 2: Time analysis summary

Case	Amortized Time
Processing the insertion of a new hashtag	$O(m)$
Processing the removal of a hashtag from the Current Window	$O(m + \log(s))$
Updating History and Current Window at the end of a day	$O(nm + ms + s\log(s))$
Querying for the top k trending items	$O(k\log(s))$

Otherwise, it takes O(m) 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(m) 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(nm). Then, for each of the s nodes in the heap, it takes O(m) time to compute each updated key and O(log(s)) amortized time to update the heap given the new key.

Thus, it takes O(nm + slog(s)) amortized time + O(ms) 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 whats trending.

4.2.2 Space analysis

The History requires O(nm) space for each Count-Min sketch, so it requires a total of O(nm*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(nm*log(T) + x) space since s < x.

4.2.3 Correctness

This algorithm finds the k hashtags that have the maximum value of (frequency in last y minutes) / (value in Hokusai data structure).

Claim 1. *The value for hashtag x in the aggregate Count-Min sketch of the Hokusai 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).*

Proof. First, we use Theorem 4 of the Hokusai paper[2] 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 m such that seeing h incremented counters in position b.

Case 1: $m = t$

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

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

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

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 k be the largest j such that $t - \delta - 2^j \geq m$

Then $m \leq t - \delta - 2^k$. Let $\lambda = t \mod 2^{k+1}$. Then $\lambda = \delta$ or $\lambda = \delta + 2^k$.

Since k is the largest j such that $t - \delta - 2^j \geq m$, $m > t - \lambda - 2^{k+1}$.

Also, $m \leq t - \delta - 2^k \leq t - \lambda$, so M^{k+1} contains statistics about h.

For all $j \geq i$, $t - \delta - 2^j < t - 2^j < m$, so $k \leq i - 1$.

Thus, incrementing the counter in M^{k+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 $\frac{1}{t-m}$.

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). □

[2]http://arxiv.org/pdf/1210.4891v1.pdf

This value is approximately (frequency in last y minutes) / (freq today + $\sum_{i=1}^{T} \frac{1}{i}$* (frequency of hashtag i days ago)). This seems to be a desirable function to maximize since it finds hashtags that are common in the last y minutes 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.

5 Experiments

6 Variants

6.1 Heap function

6.2 Modify heap to always know what's trending

(Good for accuracy, bad since forfeits gain from using Fibonacci heap since items will be deleted from tree more frequently.)

6.3 Deamortizing end of day cost